ABSTRACT
Existing security-policy-specification languages allow users to specify obligations, but open challenges remain in the composition of complex obligations, including effective approaches for resolving conflicts between policies and obligations and allowing policies to react to the obligations of other policies. This paper presents PoCo, a policy-specification language and enforcement system for the principled composition of atomic-obligation policies. PoCo enables policies to interact meaningfully with other policies’ obligations, thus preventing the unexpected and insecure behaviors that can arise due to partially executed obligations or obligations that execute actions in violation of other policies. This paper also presents and analyzes the PoCo language’s formal semantics and implementation.

KEYWORDS
Policy composition, policy specification, obligations

1 INTRODUCTION
Security-policy composition is a classic problem in software security, due to conflicts that arise when policies have competing requirements. To date, policy composition does not have a complete solution; many languages are domain specific, and the general-purpose solutions may compose obligations in undesirable ways, such as allowing obligations to execute even when they violate the constraints of other policies.

As software becomes more complex, the quantity and severity of security vulnerabilities increases [1]. Managing policies that mitigate these vulnerabilities becomes challenging as the complexity increases; enforcement may devolve into a patchwork of security mechanisms affecting each other in unexpected or hard-to-understand ways, or policies may expand to become complex, monolithic specifications that conflate cross-cutting concerns. As the complexity of policies increases, so does the likelihood of errors within the policies.

Contributions. This paper presents PoCo (short for Policy Composition), a new policy-specification language and enforcement system that:

- Allows for principled (i.e., with provable guarantees) composition of complex atomic obligations
- Supports pre-, post-, and ongoing-obligations
- Allows policies and their obligations to be effectful and specified in a Turing-complete language
- Uses static analysis to enable conflict resolution between policies and other policies’ obligations
- Allows policies to control and react to other policies’ obligations
• Supports custom policy-composition operators

As far as we are aware, PoCo is the first system to provide support for atomic obligations, including conflict resolution and allowing policies to react to obligations. In addition to the design of the PoCo enforcement system, this paper presents and analyzes the language’s formal semantics and implementation.

2 GOALS

For obligation-based policies to be expressive and composable, an ideal general-purpose enforcement system would support 1) pre-, post-, and ongoing obligations, 2) atomic obligations, 3) obligations with side effects, 4) Turing-complete policy specification, 5) conflict resolution between policies and obligations, 6) complete mediation of obligations, and 7) custom composition operators.

2.1 Definition of Goals

The following provides the definition and more details of the aforementioned goals.

Obligation-Type Support. Based on their time of execution, obligations can be partitioned into three categories: pre-, post-, and ongoing [18–20]. A pre-obligation is fulfilled before the decision about a security-relevant event is enforced. For example, in the file-deletion-confirmation policy, the confirmation obligation must be enforced before the decision to permit or deny a deletion because the permit/deny decision depends on the result of the obligated confirmation-pop-up action. A post-obligation is fulfilled after such a decision is carried out, as in a policy that logs all successful bank transactions. An ongoing obligation is carried out asynchronously during the time that decisions are being enforced. For example, a policy responsible for monitoring the usage of system resources might be implemented as an ongoing obligation.

Support for these standard obligation types is essential for maximum expressiveness of an obligation-based policy enforcement mechanism. If any type is absent, there would be a class of obligation policies that cannot be enforced by the system.

Atomic Obligations. An atomic obligation requires that either all or none of the included actions are executed. Atomicity can be extended to include the decision to permit or deny an event after the obligation executes. For many practical policies, obligation atomicity is necessary for correctness. For example, in the policy that prevents accidental file deletion by showing a confirmation dialog before granting a file-deletion request, if the obligated action of displaying the dialog is carried out, then the action’s associated decision, as entered by the user, must also be followed. Otherwise, the policy may incorrectly deny a file deletion that the user already confirmed or permit a file deletion that the user canceled.

Obligations with Side Effects. Related work (e.g., [5]) requires obligations to be side-effect free, which makes some policies unenforceable. For example, any obligation that prompts the user for a decision or makes a call to a remote procedure causes side effect(s) that cannot be undone; any enforcement mechanism that relies on rolling back obligations may be unable to manage such effectful obligations correctly.

Complete Mediation of Obligations. Policies sometimes need to react to other policies’ obligations. For example, the open-file-limiting policy needs access to the number of open files, including those opened while enforcing other policies. Excluding files opened or closed during obligation execution may cause the policy to have an inaccurate count, leading to incorrect enforcement. The ability to monitor all events, including those executed by policy obligations, is called complete mediation [21].

Turing Completeness. Turing-complete policy specification languages ensure expressiveness, at the cost of non-guaranteed enforcement termination (discussed in Section 3.5). Tools, like PoCo, that aim to provide a general-purpose policy specification language, prioritize expressiveness.

Conflict Resolution. Several types of conflicts are possible when enforcing policies. Policies may disagree on a decision regarding a trigger action, an obligation may be disallowed by another policy’s requirements, or multiple policies may wish to enforce obligations in response to the same event.

Disagreement between policies on a permit/deny decision is the simplest type of conflict and can be resolved with Boolean algebra. Allowing users to implement logic to combine permit/deny decisions enables resolution of this type of conflict.

When one policy’s obligation violates the rules of another policy, the resulting behavior can be inconsistent with the behavior of each policy in isolation.

While the order of execution is unimportant for some obligations, for others it is critical for correctness. For example, an obligation to log an event to a file and an obligation to log that same event using a network connection could both be satisfied in either order. However, when the execution of one obligation makes the execution of another unnecessary or incorrect, obligations must include fallback options and be prioritized so that the most important obligations are executed first. Obligations that might cause such conflicts include those that exit the application (and therefore prevent other obligations from executing) or that make changes to the event being processed when other obligations are attempting to do the same.

Custom Composition Operators. There are infinitely many strategies to compose policies. Certain policies need higher priority; some policies may only trigger under certain conditions; the decision of one policy may only matter when another policy agrees with its decision; etc. Each of these examples requires custom composition logic. For the sake of expressiveness, it is therefore desirable to allow policy writers to implement their own custom logic for composing policies.

2.2 Overview of Related Work

Table 1 provides an overview of the goals satisfied by existing policy-composition languages. A more detailed discussion of related work appears in Section 10. Although there has been significant research on the composition of obligation policies,
composition is a known challenge when considering obligations as aspects [22] and, to our knowledge, there has been no work that accomplishes all the goals outlined in Section 2.1.

3 THE POCO MONITOR ARCHITECTURE
PoCo’s enforcement mechanism operates as a monitor that has the ability to observe a target application’s security-relevant actions (e.g., system or method calls) and the results of these actions, as shown in Figure 1. Hence, actions and their results can trigger the monitor to respond, with the response depending on the logic of the policies being enforced.

3.1 Monitor Operation
The PoCo monitor observes all security-relevant events—actions and results—and broadcasts each event to every policy being enforced. Each policy inputs the current trigger event \( e \) (i.e., the security-relevant event triggering policy enforcement) and suggests an obligation to be executed before \( e \) is processed. This obligation, which may be empty, can implement supplemental logic or alter the input event to meet the policy’s goals. In PoCo, security-relevant events are inferred from the logic of enforced policies and can be further defined by the policy author. This ensures that the monitor only broadcasts those events that are required for policy enforcement.

The PoCo monitor can execute any number of obligations before relinquishing control back to the target application by returning a result to it. After relinquishing control, PoCo cannot execute additional obligations until receiving a new event. The monitor therefore operates in a loop, with each iteration performing the following steps:

1. Input security-relevant event \( e \)
2. Collect obligations from policies in response to \( e \)
3. While there are obligations to process
   a. Select an obligation \( o \)
   b. Collect and process policies’ votes on \( o \)
   c. If \( o \) is approved, execute \( o \)
   d. Collect obligations triggered in response to \( o \)
4. If a new output event has been set, execute or return it
5. Otherwise, output the original input event

This repeats until the event that is output is a result that can be returned to the application being monitored. With this design, the monitor maintains control of execution until all approved obligations have executed, that is, the pool of pending obligations is exhausted.

3.2 Monitor Configuration
Before examining PoCo policies in detail, it is useful to understand the configuration options available for composing the policies. This section provides a quick overview of these options; a more detailed discussion appears in Section 6.

Three elements can be supplied to the monitor when specifying a composed policy. The first is a list of policies to be enforced. These base policies are the building blocks used to construct the composed policy.

The second element is a vote combinator function to combine policies’ votes on an obligation into a single permit/deny decision. The monitor uses this decision to determine the obligations to execute. PoCo’s default vote combinator is conjunctive and only permits obligations that are permitted by all base policies. PoCo users can alternatively define and apply custom vote combinators.

The final element is an obligation scheduler function which is used to prioritize obligations for execution. PoCo’s default obligation scheduler orders policies by the order in which they are defined. A custom obligation scheduler allows policies to be prioritized by other features, such as the complexity of the policies’ obligations. For example, PoCo can be configured to always execute simpler obligations before more complex obligations.

Hence, the PoCo monitor can be viewed as a policy scheduler. The monitor decides which obligations to execute and in what order. The monitor’s parameters allow this scheduling to be customized.

3.3 Obligations
Throughout the literature on policy specification and enforcement, there are many definitions of obligation [3–6, 18–20, 23]. Generally, an obligation is one or more actions required to

---

### Table 1: Summary of the extent to which existing policy-specification languages satisfy the goals enumerated in Section 2.

<table>
<thead>
<tr>
<th>Language</th>
<th>Supports obligations that are pre-</th>
<th>Supports obligations that are post-</th>
<th>Supports obligations that are effectful</th>
<th>Supports obligations that are Turing complete</th>
<th>Supports resolving conflicts for obligations that are non-atomic</th>
<th>Supports resolving conflicts for obligations that are atomic</th>
<th>Supports reacting to obligations that are non-atomic</th>
<th>Supports reacting to obligations that are atomic</th>
<th>Supports custom composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>XACML [7]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XACML Extensions [8–12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Polymer [4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ponder [6, 13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPL [5, 14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heimdall [15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rei [16]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aspect Oriented [17]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
execute under certain circumstances with specific timing in relation to events occurring in the application being monitored.

When conflict detection and resolution are introduced to obligation systems, the idea that an obligation is guaranteed to execute must become less strict. When there is a conflict involving an obligation, the only options available are to execute the offending obligation (i.e., ignore the conflict), execute the parts of the obligation that are not in conflict with other policies (i.e., non-atomic obligations), or do not execute the obligation at all (i.e., obligation execution is not guaranteed).

Since one of PoCo’s goals is to dynamically resolve conflicts involving atomic obligations, the only option is to not execute conflicting obligations. Other works have referred to this definition of obligations as “suggestions” since they are not guaranteed to execute [4]. However, even XACML—which does not provide conflict resolution among obligations—suffers from non-guaranteed obligation execution when an intermediate value in the policy/rule hierarchy fails to match the decision of the policy that generated the obligation [24, Section 7.18]. Therefore, we have opted to use the term obligation over a variant such as suggestion with the understanding that the monitoring system is obligated to attempt execution of the obligation. An obligation in PoCo is a series of actions the monitor attempts to execute prior to enforcing a decision on a proposed action or result.

3.4 Complete Mediation of Policies with Atomic Obligations

Complete mediation—the ability to monitor events executed by other policies—is a desirable trait for the enforcement of obligation policies. By default, complete mediation is understood to be implemented such that each security-relevant event can be responded to individually. We refer to this design as event-by-event complete mediation. At least one existing system has provided event-by-event complete mediation but without allowing for atomic obligations [4]. In fact, as Theorem 1 shows, it is impossible to have both event-by-event complete mediation and atomic obligations.

THEOREM 1. Event-by-event complete mediation and atomic obligations cannot both be achieved simultaneously.

PROOF. For all monitors m, if m allows event-by-event complete mediation of policy obligations, then m must allow all policies that it enforces to examine and react to each event in an obligation o as it executes. If any of m’s policies react to or alter any event in o, then o was not executed atomically.

Therefore, PoCo enforces obligation-by-obligation complete mediation, meaning that every policy can monitor and react to every other policy’s atomic obligations (rather than every individual event within those obligations).

3.5 Non-termination of Policy Enforcement

By including branching, looping and variables (Section 4), PoCo is designed to be Turing complete, which introduces possible non-termination in the enforcement code; e.g., policies may contain infinite loops. In addition, allowing policies to react to each other introduces an additional path to non-termination—two policies may generate an infinite sequence of obligations in response to each other’s obligations (e.g., one policy monitors all network connections and logs them to a file while another monitors all file writes and opens a new network connection on each, to log the file write in a database). This non-termination cannot be statically detected in general. This design prioritizes policy expressiveness over guaranteed enforcement termination.

4 POCO LANGUAGE

This section summarizes the formal syntax and semantics that highlight the key features of the PoCo Language and enable formal type-safety reasoning. The primary purpose of these semantics, which includes all of the core features of the PoCo Language, is to express the workings of these features in a precise and unambiguous manner. PoCo is formalized as a functional language due to the inherently simpler specification compared to object-oriented languages such as Java. Using these semantics, the PoCo language is proven to be type safe through standard type-preservation and progress lemmas.

4.1 Syntax

Figure 2 lists the syntactic elements of PoCo. The PoCo language derives from the simply-typed lambda-calculus (STLC) with four base types: Int, Bool, String, and Unit; three composite types: arrow rw → r2, homogeneous list τList, and reference type τ Ref; and two algebraic data types: variant and record (i.e., sums and products with labels). In addition, several algebraic data type aliases are defined to simplify the semantic presentation.

Notably, a Res is a wrapper around a security-relevant result of any type. Since it is dynamically typed (it has a subterm of type TypedVal), it is used primarily in policy code that may not know the return type of trigger events statically. While similar, a τ Res wraps a security-relevant event of type τ. It is used in code that must unconditionally produce a term of type τ (namely, in the dynamic semantics). The same distinction applies to τ Event and Event.

A PoCo policy is a record containing the policy’s name as well as its three components: onTrigger, vote, and onObligation (described in detail in Section 5).

Introduction and elimination expressions are added for each type. It is worth mentioning that, although both call(e1,e2)
Types:
\[
\begin{aligned}
\tau &::= \text{Bool} \mid \text{String} \mid \text{Int} \mid \text{TypedVal} \mid \text{Unit} \mid \tau \text{Ref} \mid \tau \text{List} \\
\text{Act} &::= (\text{name} : \tau \times \text{arg} : \text{TypedVal}) \\
\text{Res} &::= (\text{act} \times \text{result} : \tau) \\
\text{EVENT} &::= (\text{act} \times \text{result} : \text{TypedVal}) \\
\text{OBLIGATION} &::= (\text{act} \times \text{result} : \text{TypedVal}) \\
\text{CFG} &::= (\text{nodes} : \text{Act} \times \text{List} \times \text{edges} : (\text{start} : \tau \times \text{Act} \times \text{end} : \text{Act} \times \text{List} \times \text{obligation} : \text{OBLIGATION}) \\
\text{Pol} &::= (\text{name} : \tau \times \text{onTrigger} : \text{Event} \rightarrow \text{unit} \times \text{onObligation} : \text{Res} \times \text{List} \rightarrow \text{unit} \times \text{vote} : \text{CFG} \rightarrow \text{Bool}) \\
\text{OS} &::= (\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG} \rightarrow (\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG} \rightarrow \text{Bool}) \\
\text{VC} &::= (\text{name} : \tau \\
\text{Option} &::= (\text{some} : \tau \times \text{none} : \text{Unit}) \\
\end{aligned}
\]

Values:
\[
\begin{aligned}
b &::= \text{true} \mid \text{false} \\
f &::= \text{fun} \ x_1 : \tau_1 \rightarrow \ldots \rightarrow x_n : \tau_n \rightarrow \epsilon \\
\varrho &::= \text{let} \ x_1 = \epsilon_1 \ 	ext{in} \ \ldots \ \text{let} \ x_n = \epsilon_n \ 	ext{in} \ \text{epsilon} \\
\text{Expression:} \\
e &::= \text{v} \mid \text{x} \mid \epsilon_1 \mid \epsilon_2 \mid \text{let} \ x_1 = \epsilon_1 \ 	ext{in} \ \ldots \ 	ext{let} \ x_n = \epsilon_n \ 	ext{in} \ \epsilon \\
\text{act}(\epsilon_1, \epsilon_2) &::= (\text{name} = \epsilon_1, \text{arg} = \epsilon_2) \\
\text{res}(\epsilon_1, \epsilon_2) &::= (\text{act} = \epsilon_1, \text{result} = \epsilon_2)
\end{aligned}
\]

Figure 2: Formal syntax for PoCo

and \text{invoke}(\epsilon_1, \epsilon_2) can be used to call functions, the two expressions are designed for different purposes. The expression \text{call}(\epsilon_1, \epsilon_2) is used to call functions that reside within the application or within the policy and is the elimination form for the function type. On the other hand, \text{invoke}(\epsilon_1, \epsilon_2) is used primarily by the PoCo monitor to execute security relevant actions without a direct reference to the action’s function. Its first parameter is a String that specifies a monitored function’s name.

Label elements are added to facilitate the obligation property proofs in Section 7. The two elements, \text{begin}_f and \text{end}_f, mark the beginning and end of a function’s execution and can include parameters and return values. These elements have no effect on a program’s return values.

4.2 Static Semantics

Figure 3 presents some of the static-semantics rules of the PoCo language, a full listing of which is included in Appendix C. In the judgment form \(\Lambda, \Gamma \vdash e : \tau\), the context \(\Gamma\) maps variables to their types while \(\Lambda\) maps memory locations to types. The rule \text{makeCFG} specifies the type of the undefined \text{makeCFG} function. For function calls within target applications and policies, the rule \text{call} specifies that the first parameter is a function type and the second is the type of this function’s parameter. Different from \text{call}, the rule \text{invoke} is used for a PoCo monitor to execute valid actions that are output from the monitor, thus it requires the first parameter to be a \text{String} type which specifies a valid function name (it is assumed that functions will have unique names and signatures). Lastly, since \text{label} elements are merely used for recording start and end points, their type depends only on the type of their subexpression.

A. G \vdash e : \tau

\[\begin{align*}
\Lambda, \Gamma \vdash e : \text{OBLIGATION} &::= (\text{makeCFG}) \\
\Lambda, \Gamma \vdash \text{makeCFG}(\epsilon_1) : \text{CFG} &::= (\text{call}) \\
\Lambda, \Gamma \vdash \epsilon_1 : \tau_1 \rightarrow \tau_2 \quad \Lambda, \Gamma \vdash \epsilon_1 : \tau_2 &::= (\text{call}) \\
\Lambda, \Gamma \vdash \text{call}(\epsilon_1, \epsilon_2) : \tau_2 &::= (\text{call}) \\
\Lambda, \Gamma \vdash \epsilon_1 : \text{String} \quad \Lambda, \Gamma \vdash \epsilon_2 : \text{TypedVal} &::= (\text{invoke}) \\
\Lambda, \Gamma \vdash \text{invoke}(\epsilon_1, \epsilon_2) : \text{TypedVal} &::= (\text{invoke}) \\
\Lambda, \Gamma \vdash e : \tau &::= (\text{end} ; \text{label}) \\
\end{align*}\]

Figure 3: Static semantics rules for CFGs, labels, and function calls.

4.3 Dynamic Semantics

To express the run-time behavior of monitored applications, PoCo’s dynamic semantics are defined using small-step operational semantics with a left-to-right, call-by-value evaluation order. The complete dynamic-semantics rules are presented in Appendix D.

One interesting part of PoCo’s dynamic semantics is the set of rules for function calls. As shown in Figure 4, five rules are included to handle function calls. Aside from all adding a \text{begin} and an \text{end} label to an execution trace before and after the evaluation, the five rules are used to determine the behavior of a function call based on different situations. Specifically, I) for a security-relevant function call, \text{callNonMonitoredFunction} directly evaluates the call and attaches a \text{begin} and an \text{end} label to the function call’s execution trace. 2) for a security-relevant function call that originates from the target application, the rule \text{callMonitorApplication} will invoke the monitor to resolve the security-relevant event; 3) the rule \text{monitorV}, which is directly responsible for invoking the PoCo monitor, sets the flag \text{inOb} to true before evaluation to indicate the current executing context; 4) for \text{onTrigger} and \text{onObligation} function calls, the rules \text{callOnTrigger} and \text{callOnObligation} reset the current \text{result} before evaluation; 5) for a security-relevant function call (i.e., included in the list of monitored functions \(\text{F}\)) that originates from an obligation, the rule \text{callFromObligation} will append the result of evaluating the call to the current \text{result} trace;
Appendix B, illustrate the core features of PoCo policies. The

Theorem 2 (Type-safety).

\( C, e : \tau \land (C, e) \Rightarrow (C', e') \Rightarrow (C'', e'' : (C', e') \Rightarrow (C'', e'')) \).

5 POCO POLICIES

PoCo policies are designed to be granular pieces of logic that execute obligations based on security-relevant input events. Each security-relevant action the target application attempts to execute, and each security-relevant result the underlying system attempts to return, is broadcast to all the policies registered with the monitor.

The following policies will serve as running examples throughout this section.

- \( P_{file} \) disallows users from opening the secret.txt file.
- \( P_{postlog} \) requires every file-open action to be logged after the action has occurred.
- \( P_{prelog} \) requires every file-open action to be logged before the action has occurred.
- \( P_{confirm} \) requires each file-open action attempted by the target to be confirmed through a pop-up window.
- \( P_{time} \) disallows popups unless at least 100 seconds have passed since the last popup.

Taken together, these examples, which are fully defined in Appendix B, illustrate the core features of PoCo policies. The remainder of this section discusses these core features.

5.1 The onTrigger Policy Method

The first component of a PoCo policy, called onTrigger, is an obligation which executes prior to security relevant input events. Thus, policies that wish to respond to actions attempted by the application or results returned from the system must specify an onTrigger obligation. It is worth noting that because onTriggers can respond to both actions and results, they can implement pre-, post-, and ongoing- obligations (see Theorem 7).

onTrigger obligations take a trigger event \( e \) as input and may specify arbitrary logic to run before \( e \) is either executed or returned. onTriggers may also set an output event, which is the final PoCo response to a trigger event. Ultimately, PoCo must relinquish control to enable the application or system to continue executing. If no policy specifies an output event, the PoCo monitor will cede control by allowing the trigger event to be executed, that is, by outputting the input event.

For example, \( P_{file} \)'s onTrigger examines the trigger event \( e \). If \( e \) is fopen(secret.txt) then \( P_{file} \)'s onTrigger sets exit as the output event, meaning that the monitor should cede control to the system to execute the exit action. \( P_{file} \) does not specify an output event when \( e \) is not fopen(secret.txt), thus allowing irrelevant events to execute normally. No additional events are executed prior to the monitor ceding control. Hence, \( P_{file} \)'s onTrigger is defined as follows.

```latex
\textbf{fun} onTrigger(e:Event):Unit = \\
\textbf{case} e \textbf{of} \textbf{act a =>} \\
\textbf{if} a \textbf{name} == "fopen" \textbf{then} \\
\textbf{setOutput}\{\textbf{return} (act("exit", \textbf{makeTypedVal(}\textbf{Unit, unit}))\}; \textbf{unit} \\
\textbf{else} \textbf{unit} \\
| \textbf{res r =>} \textbf{unit} )
```
Output events must be treated specially because the monitoring system must reach agreement in how the transfer of control occurs. Therefore, one of the primary objectives of any system for composing run-time policies must be to determine the singular output event for each trigger event. Output events that are actions cede control to the underlying system, and output events that are results cede control to the application.

Prior work has defined models of monitors that operate in this way, interposing between applications and executing systems and responding to trigger events with output events [26].

As seen in $P_{file}$’s onTrigger, changing the output event is accomplished with the setOutput method. Calling this method commits the monitor to using that event as the output event. Once setOutput has been called for a given trigger event, additional calls by any policy for the same trigger event return false to indicate that the output event cannot be overwritten. To avoid this, a policy may first call getOutput or outputNotSet to confirm that an output has not yet been set; policy logic may then determine what happens if the output event has already been set.

$P_{confirm}$’s onTrigger method tests whether the trigger event, $e$, is a file-open action. If it is and no output event has been set, then onTrigger specifies an obligation to confirm $e$. Based on the result of the confirmation, onTrigger sets the output event to $e$ (indicating that the file open must be executed) or unit (indicating that an empty result must be returned to the application in lieu of opening the file). If $e$ is not a file-open action, or an output event is already set, $P_{confirm}$ inserts no additional logic. Hence, $P_{confirm}$’s onTrigger is defined as follows.

```plaintext
fun onTrigger(e:Event):Unit =
    (case e of act a =>
    if a.name="fopen" ∧ outputNotSet()
    then
      if call(popupConfirm,e)
      then setOutput(e); unit
      else setOutput( inres(res(a,makeTypedVal(UNIT,unit)))) :Event); unit
    else unit
    | res r => unit),
```

The ability to permanently set the output event is required for $P_{confirm}$’s correctness. If it were possible for the output event not to execute due to other policies’ obligations, then the user could opt to allow the file open before the monitor chooses not to allow it, or the user could opt to disallow the file open before the monitor executes it anyway. This level of control also enables policies to self-manage in instances where they conflict with other policies.

### 5.2 The vote Policy Method

The second component of a PoCo policy is the vote method, which votes on whether a given obligation should be executed. To enable static analysis of obligations, PoCo represents them as Control Flow Graphs (CFGs). Hence, the vote method takes the CFG of an obligation $o$ being considered for execution and returns a boolean vote indicating approval or disapproval of $o$. For example, the goal of $P_{file}$ is to prevent the secret.txt file from being opened, even by other policies’ obligations. Therefore, when examining an obligation, $P_{file}$ looks for fopen(secret.txt) in the obligation’s CFG. If $P_{file}$ finds that action, it votes to disallow the obligation. Otherwise, $P_{file}$ votes to allow it. Hence $P_{file}$’s vote is:

```plaintext
fun vote(cfg:CFG):Bool =
  ¬call{containsAct, (cfg=cfg, name="fopen",
    arg=(inArg makeTypedVal(String, "secret.txt"))
    :(arg:TypedVal + none : unit, count=1))
```

To ensure obligation atomicity, PoCo policies analyze obligations before they execute—specifically, policies vote on candidate obligations based on their statically generated CFGs. These CFGs are conservative approximations since computing the exact CFG for an arbitrary program is undesirable. Because it is not always possible to determine the arguments of actions invoked in obligations, it is necessary to allow unresolved arguments, which are parameters to a security-relevant action that could not be determined statically. The CFG of an obligation defines unresolved arguments as such, and policies may specify how to handle unresolved arguments. For example, $P_{file}$ conservatively votes against obligations known to open the secret.txt file and also obligations containing file opens with unresolved arguments. $P_{file}$’s vote is therefore:

```plaintext
fun vote(cfg:CFG):Bool =
  ¬call{containsAct, (cfg=cfg, name="fopen",
    arg=(inArg makeTypedVal(String, "secret.txt"))
    :(arg:TypedVal + none : unit, count=1))
  ∧ ¬call{containsAct, (cfg=cfg, name="fopen",
    arg=(innone unit)
    :(arg:TypedVal + none : unit, count=1))
```

### 5.3 The onObligation Policy Method

The third component of a policy is an obligation that may be executed in response to other obligations, in order to inject additional actions after the triggering obligations. This is necessary to achieve the goal of policies reacting to other policies’ obligations. The onObligation method responds to obligations by analyzing the results of all security-relevant actions performed during an obligation’s execution, that is, a result trace (rt). For example, $P_{postlog}$ proposes an obligation that logs each file open in another obligation. This new obligation is specified in the policy’s onObligation as follows:

```plaintext
fun onObligation(rt:ResList):Unit =
  (let results=ref rt in
    while(¬empty(!results)) {
      let event = head(!results) in
      results := tail(!results);
      if event.act.name == "fopen"
        then call(log,event)
        else unit
    end)
end)
```

PoCo cannot insert obligations before the execution of a triggering obligation because doing so may create inconsistency in the execution. Prior to executing an obligation $o_1$, the monitor decides whether $o_2$ should be executed. Execution of another obligation, $o_2$, prior to $o_1$ may cause policies to vote differently than they did originally, when deciding to permit $o_1$. If PoCo were designed to re-query policies after $o_2$ was
inserted, and the new decision was to not execute \( o_1 \), it is possible that \( o_2 \) should not have been proposed in the first place. To have reliable behavior, the voting on and execution of a given obligation must therefore be treated as an atomic unit. For this reason, it is not possible in \( P_{\text{prelog}} \) to log events in obligations prior to their execution, though it is possible to do so in \( \text{onTrigger} \).

To summarize, there are two ways PoCo policies specify obligations: \( \text{onTrigger} \) specifies an obligation in response to a trigger event, and \( \text{onObligation} \) specifies an obligation in response to other obligations (which may be defined by \( \text{onTrigger} \) or \( \text{onObligation} \)). The \( \text{vote} \) method enables policies to indicate approval or disapproval of obligations.

### 5.4 Parameterized Policies

To aid code reuse, PoCo enables abstraction over common policy patterns. This can be achieved by declaring a function which instantiates different policies based on its argument. For example, there are many policies that might disallow one particular action. This set of policies can be abstracted over with the following function:

```ocaml
fun disallow (x: Act) : Pol = (name = dis,
onTrigger = (fun ot(e:Event):Unit =
case e of
  act a =>
    if a.name == x.name && a.arg == x.arg
    then setOutput(event(
        act("exit", makeTypedVal(Unit,unit)))))
else unit
| res r => unit).
onObligation = (fun oo(rt: ResList):Unit = unit),
vote = (fun vt(cfg: CFG):Unit =
  ¬call(containsActArg,
    (cfg=cfg, name="popup", count=2))
  ∧ ¬call(containsActAnyArg,
    (cfg=cfg, name="popup", count=1))
  ∧ currTime < (1+t100)))
end
```

Other uses of this functionality could be to specify directory paths, port numbers, or any other data that may be relevant to a specific policy.

### 5.5 Local Policy State

Without the ability to keep local state information, any policy that needs to "remember" details about previous events cannot be enforced. It has been noted, in general, that restricting a monitor’s access to state information can have a significant effect on the policies that can be enforced [27]. PoCo policies can utilize let environments and memory references to manage this data.

\( P_{\text{time}} \), shown below, tracks the last time a popup window was displayed. If it was less than 100 seconds ago and the application attempts to open another, the policy attempts to exit the application. The time variable, \( t \), records the last occurrence of the popup event to compare during future attempts to execute the popup action. The let environment initializes this variable.

```ocaml
let t = ref 0 in (name = pol, onTrigger = (fun ot(e:Event):Unit =
case e of act a =>
if a.name=="popup" then
  if currTime<1+t100 ∧ outputNotSet()
  then setOutput(act("exit", makeTypedVal(Unit,unit)))
else t:=currTime
else unit
| res r => unit).
onObligation = (fun oo(rt: ResList):Unit = unit),
vote = (fun vt(cfg: CFG):Unit =
  ¬call(containsActArg,
    (cfg=cfg, name="popup", count=2))
  ∧ ¬call(containsActAnyArg,
    (cfg=cfg, name="popup", count=1))
  ∧ currTime < (1+t100)))
end
```

For the interested reader, Appendix B presents complete specifications of six example policies. Their construction follows directly from the policy components that have been described in this section.

### 6 POLICY COMPOSITION

The PoCo monitor, briefly discussed in Section 3.2, handles composition of policies by scheduling obligations, dispatching the agreed-upon output event, and handing control back to the application or system.

Conflicts produced when composing PoCo policies fall into two categories, obligations that conflict with policies and obligations that conflict with other obligations.

The first type of conflict results from an obligation \( o \) attempting an action that a policy \( p \) specifically disallows. In PoCo, this manifests as \( p \)'s \( \text{vote} \) method returning a deny response on \( o \). This type of conflict is handled in PoCo by using a vote combinator that combines the votes of all policies into a single decision to permit or deny the obligation.

The second type of conflict is a timing issue between obligations. If the execution of an obligation \( o_1 \) would render the execution of obligation \( o_2 \) meaningless or detrimental, the execution of \( o_1 \) should cause \( o_2 \) not to execute. This type of conflict is handled in PoCo by configuring the obligation scheduler to execute the most vital obligations first and writing obligations such that they are able to gracefully handle such changes.

Using the parameters provided allows both types of conflicts to be handled in the manner that the policy architect decides is the best fit for their particular use case. The following sections consider each of these configuration parameters in turn.

### 6.1 Policies

The first parameter of the PoCo monitor is a list of policies to be enforced. Each policy is registered to receive all security-relevant events that the monitor captures and broadcasts. The list of policies does not, necessarily, indicate any sort of priority or ordering of the policies. The order may be more or less important depending on the other parameters supplied to the monitor.

### 6.2 Vote Combinator

The \textbf{Vote Combinator} or \( VC \) is the second parameter that initializes the PoCo monitor. The \( VC \) is a function that is responsible for combining the boolean outputs of the policies' \( \text{vote} \) methods into a boolean output that determines if an obligation will be executed. In addition to the boolean vote of each policy, the \( VC \) may need the policy name to implement combinators that
give preference to specific policies. Therefore, the type of this argument is \((\text{name} : \text{String} \times \text{vote} : \text{Bool})\)_\text{List} \rightarrow \text{Bool}, which will be referred to as simply VC. Some possible VCs include conjunction, disjunction, and majority.

A VC can implement any logic that is desired. For example, one could write a VC that executes an obligation if a specified policy, say Pol1, does not veto it. This VC would look like:

```haskell
fun \text{VCvote}(\text{votes} : (\text{name} : \text{String} \times \text{vote} : \text{Bool})\text{List}) : \text{Bool} =
  \text{let output} = \text{ref true} \text{ in}
  \text{let rvotes} = \text{ref votes} \text{ in}
  \text{while}(!\text{rvotes}) \{ 
    \text{case head(!rvotes)} \text{ of}
    \text{if v.name == "Pol1"}
      \text{then output} := v.vote
      \text{else unit}
    \text{none unit} \Rightarrow \text{unit};
    \text{rvotes} := \text{tail(!rvotes)} \text{ end; !output end}
  \}

The PoCo implementation includes several built-in VCs that can be used in their entirety or as a building block to create other VCs. For example, it would be possible to implement a VC that allows an obligation to execute if either the first policy allows it or all other policies allow it using a combination of the built-in conjunction and disjunction VCs:

```haskell
fun \text{VCoverride}(\text{votes} : (\text{name} : \text{String} \times \text{vote} : \text{Bool})\text{List}) : \text{Bool} =
  \text{call}(\text{VCdisjunction,}
    \text{call}(\text{VCconjunction, tail(votes))} : \text{head(votes)})
```

A convenient side effect of PoCo’s event-broadcasting and voting mechanism is that policy conflicts are obvious during execution of the VC; any votes to disallow an obligation or any vote that gets overruled by the VC are conflicts between policies. It is, therefore, straightforward to detect and act on these conflicts dynamically by adding additional logic to VC. This enables logging information about the conflict so that it can be used to troubleshoot or make improvements to the affected policies.

### 6.3 Obligation Scheduler

The Obligation Scheduler or OS is the last parameter in the PoCo monitor. The OS is a function that orders obligations based on specified criteria. Example OSs include prioritizing simpler obligations, weighting specific actions with more or less priority, or applying specific priorities to the policies generating the obligations. This prioritization is especially important because it determines the single output event for a given input event. Particularly, the obligations of lower priority policies will not be able to set the single output event if an obligation of a higher-priority policy has already set it. Like the vote method for policies, the OS works with CFG representations of obligations, therefore the type for the OS is \((\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})\text{List} \rightarrow (\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})\text{List}\).

The OS allows arbitrary logic to be implemented in order to perform its function. One example OS could be a strict ordering of policies. If the policy writer wanted to prioritize the obligations in the order that the policies were provided to the monitor, they could simply return the same list. The PoCo implementation includes several example OSs including this default ordering:

```haskell
fun \text{OSdefault}(\text{obs} : \text{CFG}\text{List}) : \text{CFG}\text{List} = \text{obs}
```

Another potentially interesting way to order obligations could be based on their complexity (i.e. the number of nodes in their CFG). Essentially, this would allow simple obligations that are less likely to cause conflicts to complete before dealing with more complicated obligations:

```haskell
fun \text{OScomplexity}(\text{obs} : \text{CFG}\text{List}) : \text{CFG}\text{List} =
  \text{call}(\text{sort, (list = obs,}
    \text{comparator = (fun c((o_1 : \text{CFG})\text{List} : \text{CFG})): \text{Int} =
      \text{call}(\text{length, o_1}\text{.nodes}) - \text{call}(\text{length, o_2}\text{.nodes}))})
```

### 6.4 Monitor Operation

Now that all the inputs to the monitor have been described, let us examine how PoCo uses these inputs to provide versatile composition of policies. When a security-relevant event occurs, the monitor collects the CFGs of the policies’ onTrigger obligations. These CFGs are then prioritized using the OS and individually voted on by the policies’ vote methods.

To process an obligation \(\omega\), the monitor collects a vote on \(\omega\) from each policy and sends these votes to the VC to make the final decision on whether the monitor should execute \(\omega\).

Once a current obligation finishes executing, the monitor collects any onObligations that may have been generated and adds them (in order) to the front of the list of obligations to be processed, thus ensuring that any new obligations triggered as a result of the current obligation are voted on and executed prior to moving on to any other obligations that may be waiting.

Once all obligations are processed, the monitor checks if an output event was set by any of the policies. If one has been set, the monitor will dispatch this event in order to cede control back to the target application or the system. If no specific output event has been set, the monitor will, by default, use the trigger event as the output event.

### 7 POCO LANGUAGE PROPERTIES

Using the semantics shown in Section 4 we can prove a number of useful properties about the PoCo language and architecture:

- all obligations are atomic
- obligations always allow for conflict resolution
- policies can always react to other policies’ obligations
- pre-obligations can be used to implement post- and ongoing obligations
- it is possible to design a PoCo monitor such that the order in which the policies are declared does not affect the outcome

Proofs for these theorems can be found in Appendix G. To assist with these proofs, PoCo’s dynamic semantics are designed to output a trace indicating relevant steps taken by the program. This trace is made up of values \(\text{begin}_{f(x)}\) and \(\text{end}_{f(x)}\), where \(\text{begin}_{f(x)}\) indicates the beginning of a step with any applicable parameters and \(\text{end}_{f(x)}\) indicates the end of a step with applicable parameters and output. To simplify theorems’ presentation, the syntax also defines the following values:

```haskell
  \text{fun OSdefault}(\text{obs} : \text{CFG}\text{List}) : \text{CFG}\text{List} = \text{obs}
```

Another potentially interesting way to order obligations could be based on their complexity (i.e. the number of nodes in their CFG). Essentially, this would allow simple obligations that are less likely to cause conflicts to complete before dealing with more complicated obligations:

```haskell
  \text{fun OScomplexity}(\text{obs} : \text{CFG}\text{List}) : \text{CFG}\text{List} =
    \text{call}(\text{sort, (list = obs,}
      \text{comparator = (fun c((o_1 : \text{CFG})\text{List} : \text{CFG})): \text{Int} =
        \text{call}(\text{length, o_1}\text{.nodes}) - \text{call}(\text{length, o_2}\text{.nodes}))})
```
7.1 Atomicity of Obligations

All obligations in PoCo are executed atomically—once an obligation begins executing, no other obligation code executes until that obligation has finished executing. This does not guarantee that the executing obligation will terminate.

Theorem 3 (Atomic Obligations). For all $p$, $t$, and $t'$ if $p$ is well-typed program such that $p \downarrow \omega^+$ $p'$ and $t$ matches the $\omega$-expression $(\langle e_0 \rangle. \langle e_1 \rangle \cdots \langle e_n \rangle)$ then $t'$ matches the $\omega$-expression $(\langle e_0 \rangle. \langle e_1 \rangle \cdots \langle e_n \rangle)$.

Essentially, Theorem 3 states that another obligation will never start if the previous obligation has not completed execution. Note that an $\omega$-expression can generate possibly infinite length strings (i.e., belonging to the union of a regular and an $\omega$ language). A trace, $t$, is considered to match an $\omega$-expression, $e$, if $t$ matches $[e/\omega]e$ or $t$ matches a sub-expression of $e, e_1, \ldots, [e/\omega]e_i$. See Appendix F for a full definition of $\omega$-expression.

7.2 Conflict Resolution

PoCo defines conflict resolution as allowing each policy to vote to approve or deny each obligation immediately prior to its execution. This vote is guaranteed to be provided as input to the vote combinator which may or may not use the value to determine the final vote. Since this vote combinator is specified by the policy architect, policies have as little or as much decision-making power as is desired.

Theorem 4 (Conflict Resolution). For all well-typed programs $p$ such that $p \downarrow \omega^+ p'$, $t$ matches the $\omega$-expression $(\neg \text{beginOb}(e))^{\omega_0} (\neg \text{endOb}(e))^{\omega_0}$, where $\text{beginOb}(e)$ and $\text{endOb}(e)$ are the only types of obligations that are implemented in PoCo.

Theorem 4 shows that no obligation will start without having called the vote method for each policy and getting a true result from the vote combinator.

7.3 Obligation Reaction

PoCo defines obligation reaction as allowing each policy to propose a new obligation in response to an executed obligation that contains security-relevant events.

Theorem 5 (Obligation Reaction Part 1). For all well-typed programs $p_1, p_2, p_3$ such that $p_1 \downarrow \omega^+ p_1'$, $p_2 \downarrow \omega^+ p_2'$, and $p_3 \downarrow \omega^+ p_3'$ such that $t_1, t_2$, and $t_3$ match the $\omega$-expressions $e_{pre}$, $e_{post}$, and $e_{ongoing}$ respectively where $e_{pre} \equiv (\langle e_1 \rangle. \langle e_2 \rangle \cdots \langle e_n \rangle) \downarrow \omega^+$ $\text{beginF}(x) \downarrow \omega^+$ $\text{beginMonitor} \downarrow \omega^+$ $\text{endF}(x) \downarrow \omega^+$ $\text{endMonitor} \downarrow \omega^+$ $\text{ongoing} \downarrow \omega^+$ $\text{pre} \downarrow \omega^+$ $\text{post} \downarrow \omega^+$.

Theorem 5 shows that after each obligation containing security-relevant events ends, a CFG is created based on querying the onObligation function of each policy. Note that policies always propose an obligation with onObligation, but it is possible that it will be an empty obligation which ultimately does nothing. This, by itself, is not sufficient to prove that these obligations are executed once they are retrieved.

Theorem 6 (Obligation Reaction Part 2). For all well-typed programs $p$ where $p \downarrow \omega^+ p'$ and $p$'s monitor is the tuple $(M, f_{\text{monitor}}, p_1 \cdots p_n, \text{eos}, \text{ece})$ where the functions $e_{\text{ec}}, p_1, \text{onTrigger}, \ldots, p_n, \text{onTrigger}, p_1, \text{onObligation}, \ldots, p_n, \text{onObligation}$ terminate, for each event $\text{endMakeCFG}(t_1, t_2) : g$ in $t$ there exist $\argfalse(e)$ or $\argtrue(e)$ in $t$ where:

\[
\begin{align*}
\argtrue(e) & := (\text{beginVote}(e) \langle \text{beginOb}(e) \rangle^{\omega_0} \text{endVote}(e) : v_n) \cdots \langle \text{beginOb}(e) \rangle^{\omega_0} \text{endVote}(e) : v_N) \\
\argfalse(e) & := (\text{beginVote}(e) \langle \text{beginOb}(e) \rangle^{\omega_0} \text{endVote}(e) : v_n) \cdots \langle \text{beginOb}(e) \rangle^{\omega_0} \text{endVote}(e) : v_N) : \text{true}
\end{align*}
\]

Theorem 6 is needed to tie the results of Theorem 5 into the useful result that each of these obligations is ultimately voted on and, if approved, executed. It shows that every obligation that is turned into a CFG is eventually voted on and, if approved, executed provided that all obligations and voting functions terminate.

7.4 Obligation Completeness

Although the categories pre-, post-, and ongoing- are standard, all obligations can be implemented as pre-obligations by expanding the domain of security-relevant events to include both actions and results from actions, as shown in Figure 1. With this expanded definition of events, the obligation types of pre-on-action (i.e. pre-obligations on actions) or pre-on-result (i.e. pre-obligations on results) can be defined. An obligation $o$ is a pre-on-action obligation to an action $a$ if $o$ is fulfilled after $a$ is requested by the monitored application but before the monitor makes a decision regarding $a$. Similarly, an obligation $o$ is a pre-on-result obligation to a result $r$ if $o$ is fulfilled after $r$ is returned from the underlying system but before the monitor makes a decision regarding returning $r$ to the target application.

We refer to this property as pre-obligation completeness (Theorem 7). Similarly, ongoing obligations can be defined in terms of pre- and post-obligations. It is for this reason that Table 1 didn’t have a row for ongoing obligations; any system with pre- and post-obligations in a multi-threaded environment can implement ongoing obligations.

Pre-obligation completeness implies that only pre-on-action and pre-on-result obligations are necessary in order to support all the standard obligation categories and, as such, these are the only types of obligations that are implemented in PoCo.

Theorem 7 (Pre-obligation Completeness). There exists well-typed programs $p_1, p_2$, and $p_3$ such that $p_1 \downarrow \omega^+ p_1'$, $p_2 \downarrow \omega^+ p_2'$, and $p_3 \downarrow \omega^+ p_3'$ such that $t_1, t_2$, and $t_3$ match the $\omega$-expressions $e_{pre}$, $e_{post}$, and $e_{ongoing}$ respectively where $e_{pre} \equiv (\langle e_1 \rangle. \langle e_2 \rangle \cdots \langle e_n \rangle) \downarrow \omega^+$ $\text{beginF}(x) \downarrow \omega^+$ $\text{beginMonitor} \downarrow \omega^+$ $\text{endF}(x) \downarrow \omega^+$ $\text{endMonitor} \downarrow \omega^+$ $\text{ongoing} \downarrow \omega^+$ $\text{pre} \downarrow \omega^+$ $\text{post} \downarrow \omega^+$ $\text{ongoing} \downarrow \omega^+$ $\text{pre} \downarrow \omega^+$ $\text{post} \downarrow \omega^+$.
7.5 Policy Permutability

We have proven that it is possible to design a PoCo monitor (i.e. VC and OS pair) such that the order in which the policies are refined its design. The implementation, written in Java and packaged as a Java library, is 3,299 lines of code and is available online [citation anonymized]. This section provides details of the compiler module.

8 IMPLEMENTATION

We have implemented a prototype of PoCo to evaluate and refine its design. The implementation, written in Java and packaged as a Java library, is 3,299 lines of code and is available online [citation anonymized]. This section provides details of the compiler module.

8.1 PoCo Compiler Architecture

The PoCo compiler builds a trusted application by inlining security-enforcement code into the untrusted application using AspectJ [28], an aspect-oriented extension to Java. The AspectJ compiler inlines code, called advices, that executes before and/or after methods specified with one or more pointcuts [29]. The decision to use AspectJ over manual bytecode re-writing was made largely for simplicity and because bytecode re-writing to enforce run-time policies has already been accomplished by other projects [4] so there is no novelty in creating an additional implementation.

The PoCo compiler is made up of four modules: the pointcut extractor, policy converter, static analyzer, and AspectJ compiler, depicted in Figure 5. Following the flow of code translations, the PoCo compiler takes a list of policies specified in .pol files as input and uses the pointcut extractor to create an AspectJ (.aj) file including security-relevant methods monitored by the policies as the pointcut set. Next, the policy converter reconstructs the .pol files into Java (.java) files and creates a policy-scheduler file using the specified obligation scheduler and vote combinator (in os and .vc files respectively). Then the static analyzer statically creates CFGs that represent the actions that may be invoked for each obligation. Finally, given the generated files and the information gathered for each obligation, the AspectJ compiler inlines the policy-enforcement code into the target application.

8.2 Pointcut Extractor

The pointcut extractor obtains a policy’s events of interest by scanning the onTrigger method, discussed in Section 5.1; it locates all matches calls on the trigger action in the policy’s onTrigger method and extracts the arguments to create AspectJ pointcuts. It is always possible to manually modify the pointcuts after they are determined. This customization may be desirable in cases where complex logic makes it impossible for static analysis to determine the security relevant events or in cases where the writer wishes to manually restrict the events being monitored to improve system performance.

Let’s consider the policy P

11

The decision to use AspectJ over manual bytecode re-writing was made largely for simplicity and because bytecode re-writing to enforce run-time policies has already been accomplished by other projects [4] so there is no novelty in creating an additional implementation.

The PoCo compiler builds a trusted application by inlining security-enforcement code into the untrusted application using AspectJ [28], an aspect-oriented extension to Java. The AspectJ compiler inlines code, called advices, that executes before and/or after methods specified with one or more pointcuts [29]. The decision to use AspectJ over manual bytecode re-writing was made largely for simplicity and because bytecode re-writing to enforce run-time policies has already been accomplished by other projects [4] so there is no novelty in creating an additional implementation.

The PoCo compiler is made up of four modules: the pointcut extractor, policy converter, static analyzer, and AspectJ compiler, depicted in Figure 5. Following the flow of code translations, the PoCo compiler takes a list of policies specified in .pol files as input and uses the pointcut extractor to create an AspectJ (.aj) file including security-relevant methods monitored by the policies as the pointcut set. Next, the policy converter reconstructs the .pol files into Java (.java) files and creates a policy-scheduler file using the specified obligation scheduler and vote combinator (in os and .vc files respectively). Then the static analyzer statically creates CFGs that represent the actions that may be invoked for each obligation. Finally, given the generated files and the information gathered for each obligation, the AspectJ compiler inlines the policy-enforcement code into the target application.

Theorem 8 (Policy Permutability). There exists well-typed programs p1 with monitor (M, f

11

Theorem 8 provides proof that such an OS and VC can be created. Not all monitors will make use of this property, but when true modularity is needed, designing the monitor to display policy permutability will allow policies to be more freely added and removed. A proof for this theorem can be found in Appendix G.

public void onTrigger(Event e) {
    String acts = "java.lang.Runtime.exec(*)";
    if (e.matches(new Action(acts)))
        setOutput(new Result(e, null));
    }

Taking this policy as its input, the pointcut extractor locates the matches method and learns that all of the exec methods are security relevant. The extractor then creates an AspectJ file with a pointcut defined to intercept all of the overloaded java.lang.Runtime.exec methods.
Once the pointcut has been defined, the pointcut extractor defines advice to execute policy-enforcement code whenever the pointcut is triggered.

8.3 Policy Converter
Given a list of policies, the policy converter copies relevant sections of the .pol file into a .java file template. The policy converter is also responsible for creating a policy scheduler Java source file (described further in Section 8.5), using the specified obligation scheduler and vote combinator or the default PoCo scheduler and combinator. The default obligation scheduler, named OrderAsListed, preserves the original order of input policies by directly returning the input list. The default vote combinator, Conjunction, performs a logical AND operation on the policies’ votes to get its result. This combinator is restrictive and can be used for composing unanimous decision-making policies.

8.4 Static Analyzer
The PoCo static analyzer utilizes two libraries, Java Compiler Tree [30] and ASM5 [31], to generate CFG representations of each obligation. First, Java Compiler Tree (included in the com.sun.source package) is used to visit an obligation’s abstract-syntax tree (AST) and obtain information about the method calls. Let’s consider the example policy $P_{confirm}$ from Section 5 which requires every file-open operation attempted by a target application to first be confirmed through a pop-up window.

```java
public void onTrigger(Event trig) {
    if(trig.matches(fileOpenAct) && outputNotSet()) {
        if(JOptionPane.showConfirmDialog(
            null, msg, "Security Question", 0)
            == JOptionPane.YES_OPTION)
            setOutput(trig);
        else
            setOutput(new Result(trig, null));
    }
}
```

By scanning the policy’s onTrigger function with the Java Compiler Tree library, the static analyzer finds three distinct paths (for simplicity, short-circuit expression evaluation is not considered). All paths first invoke matches and outputNotSet methods. After that, one path ends while the other two paths invoke JOptionPane.showConfirmDialog. Depending on the user’s selection in the confirmation dialog each path invokes setOutput with differing parameters and then ends their execution. The CFG representation of the obligation is shown in Figure 6. This information is insufficient to make decisions about an obligation’s relevance to the concerns of the implemented policies as it does not contain type details for variables and methods. For example, the null value of the first argument of JOptionPane.showConfirmDialog is ambiguous because the value null can be assigned for variables of any non-primitive type. Thus, the argument’s type cannot be precisely inferred.

Method signatures and statically initialized argument values are obtained by reading compiled policy code with the ASM5 library, a bytecode manipulation tool that can be used to analyze Java programs. By analyzing $P_{confirm}$’s class file in this way, the signature and parameter of the setOutput method call is determined. In this case, trigger is the trigger event which updates dynamically at run-time. By mapping the detailed method information onto the control flow information, the static analyzer generates a detailed CFG for each obligation.

For this implementation, dynamic analysis of obligations is limited to changes in the trigger event; this could be extended in future work to include additional options for dynamic analysis. This primary reliance on static analysis leads PoCo policies to be more conservative than what might be accomplished by allowing additional dynamic analysis.

Once all policies have been converted into appropriate AspectJ advice and all obligations have been statically analyzed, PoCo relies on the AspectJ compiler to inline the desired policy-enforcement code into the target application.
8.5 Policy Scheduler

As mentioned earlier, the policy converter is responsible for generating a policy scheduler that uses the configured obligation scheduler and vote combinator. To respond to a security-relevant event, the policy scheduler must use the specified obligation scheduler to prioritize the list of policies and generate an ordered list of obligations. It then obtains the statically generated CFG of each policy’s onTrigger method and injects the trigger event into it. If the resulting obligation is non-empty, it is added to an obligation queue. Once all obligations have been added to the queue, the obligation scheduler pushes the queue onto a stack that holds all obligations waiting to execute.

To process an obligation, the policy scheduler removes the first obligation from the first queue on the obligation stack and collects votes from all policies on whether to allow the obligation. These votes are passed to the vote combinator to be composed into a single permit/deny decision. An obligation that is denied will be discarded and the scheduler will continue with the next obligation. An obligation that is permitted is executed, and its result trace is dynamically collected. In order to avoid time-of-check to-time-of-use (TOCTOU) vulnerabilities, the voting on and execution of an obligation needs to happen sequentially in a single thread.

Next, the policy scheduler uses the result trace combined with the onObiligation of each policy to determine if the executed obligation triggers any additional obligations. As with the onTriggers, each new obligation is added to a queue and then the queue is pushed onto the obligation stack. This stack of obligation queues ensures that obligations generated by other obligations are executed as soon as possible after the execution of the triggering obligation. Once the new queue is added to the stack, the scheduler starts the process over with the first obligation in the first queue on the obligation stack. If the obligation stack is empty, the scheduler has completed all obligations. Figure 7 illustrates this process.

Figure 6: The control flow graph of $P_{confirm}$’s onTrigger function.

Figure 7: Policy scheduler flow — obligations are generated by policies based on the trigger event, prioritized by the obligation scheduler and then voted on. Executed obligations can result in additional obligations and this process continues until there are no remaining obligations to prioritize and vote on.

9 CASE STUDY

To demonstrate the expressiveness and analyze the performance of the PoCo system, we replicated the case study that was used to validate Polymer [4], which is the most directly comparable previous work. The case study is made up of ten policies that are designed to prevent unsafe behavior in an email client. Due to differences in the structure of the two systems, there are some differences in how these policies must be written in PoCo, but the goals and results of the policies are the same. All variances from the original case study will be noted for completeness. The policies implemented were:

- IsClientSigned - trusts a cryptographically signed application and ensures that an unsigned application is monitored with additional policies. In Polymer this policy takes two policy parameters; the PoCo version instead trusts the target application by setting the trigger event as the output event. By prioritizing this policy as the first policy via the obligation scheduler the PoCo policy serves the same purpose as Polymer’s version.

- AllowOnlyMIME - prevents connections other than POP and IMAP.

- ConfirmAndAllowOnlyHTTP - disallows non-HTTP connections and opens a popup for user confirmation before allowing HTTP connections.

- IncomingEmail - logs incoming emails and flags emails from unknown addresses as SPAM. This policy includes additional security-relevant methods due to an implementation change in the latest version of Pooka.

- OutgoingMail - confirms recipients, adds a BCC, and logs all outgoing email.

- ClassLoaders - prevents the target application from creating a custom class loader.

- Attachments - warns users about dangerous email attachments before creating them.

- NoOpenClassFiles - ensures that compiled Java code will not be executed by the target application.

- DisSysCalls - prevents the target application from executing system-level calls.
The empirical results demonstrate that the overhead of the PoCo monitor is relatively low. As shown in Table 2, with one trivial policy and ten trivial policies, the average timing overheads for loading Pooka are approximately 0.87% and 1.54%, respectively, and, the overheads for loading a specific email are approximately 0.39% and 5.06%, respectively.

The overhead of PoCo policies, on the other hand, is dominated by policy obligations which can vary significantly from one policy to another. As PoCo obligations are Turing complete and can therefore run for arbitrary amounts of time, the overhead of a composed policy is almost entirely dependent on the complexity of its obligations and how many of these are triggered per security relevant event. The run-time overhead of the entire composed email policy for loading the application is 17788.89ms (292.78%) on average. This may seem high when compared to other scenarios; however, during this time period, the PoCo monitor processes 130 security-relevant events in total and triggers complex obligations like incoming-email logging, spam-email marking, long-email-subject truncation, etc. If excluding the overhead of the PoCo monitor, the overhead per event is approximately 136.43ms.

### 10 RELATED WORK

Composition of obligation policies is a long-standing research problem. This section describes the primary efforts in the area.

#### 10.1 XACML

XACML (eXtensible Access Control Markup Language) [7] allows policies to be specified and composed using XML. XACML allows each policy to return one of four basic result values (permit, deny, indeterminate, or not applicable) and, optionally, an obligation, to express its response to a request. XACML also defines seven rule-combining and eight policy-combining algorithms to combine results from multiple policies. Due to the stateless nature of its policies and relatively simple rule structure, XACML has been widely adopted and has been implemented into commercial and open-source software products. However, XACML has a number of limitations that affect its overall expressiveness.

Even with significant research extending XACML to overcome its limitations [8–12] (e.g., to add conflict resolution by requiring manual specification of which obligations conflict [11]), XACML is still lacking in some areas. Stateless policies are less expressive than stateful policies [27] and cannot express simple policies such as “disallow network-packet sends after file reads” [33]. Table 1 summarizes additional issues with using XACML and its extensions to compose obligation-based policies.

#### 10.2 Polymer

Polymer is an object-oriented policy specification language and run-time monitoring system [4] with well-defined semantics that enables users to compose modularized policies for use on Java programs. Polymer policies issue "suggestions" in response to security-relevant events indicating what they want the monitor to do. By separating policies into an effect-free query method and an effectful accept method, Polymer ensures...
that querying a policy will have no permanent effect when its suggestion is not followed.

Because Polymer implements event-by-event complete mediation, it cannot ensure obligation atomicity (Theorem 1).

10.3 Ponder

Ponder is a policy specification language that can be used to compose access-control and general-purpose policies [6, 13]. With Ponder, users can flexibly compose complex policies based on logical relations between policies and hierarchical relationships between subjects’ policies. Obligation policies are specified in the format of “on triggering-events do obligated actions”.

Complex obligations may be specified in Ponder using its concurrency operators. If any action in an obligation violates an enforced refrain policy (i.e., policies that specify a forbidden subject, action, or object combination), then the target application halts.

Like Polymer, Ponder inspects all actions of an obligation one at a time, so the execution of an obligation can be interrupted if its actions result in a security violation. Also, like Polymer, this characteristic prevents Ponder from ensuring obligation atomicity. Ponder furthermore does not allow policies to react to the obligations of other policies; the only allowed response to a conflict between an obligation and other policies is to halt the application, which may be unacceptable in practice.

10.4 SPL

Security Policy Language (SPL) is a policy specification language that enables users to compose complex authorization policies by using policy combinators [5, 14] to resolve conflicts on permit/deny decisions of composed policies. SPL focuses on policies that make decisions based on actions executed in the past. Obligations are defined as future events that must be carried out after the execution of the current event.

SPL requires all obligations to be atomic, to ensure that future obligations are carried out. In cases where a policy’s obligation violates other enforced policies, SPL resets the application to the state before the execution of the obligation’s trigger action. For this solution to work, obligations must be pure (free of side-effects), because effectful actions generally cannot be rolled back. Excluding effectful obligations significantly limits SPL’s expressiveness.

10.5 Heimdall

Heimdall uses compensatory actions in response to execution failures in obligations [15]. Heimdall builds on the hypothesis that any executed action can be compensated by future actions. However, there may not always exist an effective compensation for security violations; a policy may be able to prevent future leakage of sensitive data but be unable to compensate for data that has already been leaked.

Heimdall also does not support conflict resolution between policies and obligations. If an application’s action triggers any obligations, Heimdall creates instances of those obligations and sends the execution request of these instances directly to the underlying system. Obligations are not validated against other policies before execution. If an obligation is fulfilled, the system sends information about this action to Heimdall, which then deletes the corresponding instance. When an obligation is not fulfilled, Heimdall requests the system to execute the compensatory action of the obligation. Enforced policies are unable to react to the executed obligations.

10.6 Rei

Rei, which modeled the concept of permissions, prohibitions, obligations, and dispensations, is a non-domain specific language that supports specifying pre-on-result obligations [16]. A Rei policy is composed of rules that are each comprised of an entity and a policy object. A policy object specifies allowed or prohibited actions and any applicable obligations; an entity specifies what subject the policy object applies to.

Rei identifies conflicts between obligations and prohibition policies and offers two ways to resolve them. The first is to specify priorities among policies and/or policies’ rules and the second is to set negative/positive-modality precedence on actions, entities, and policies. The authors do not address the issue of conflicts involving complex obligations specifically, but the context suggests that obligated actions are handled individually, and thus complex obligations would not be executed atomically. It is also unclear if Rei is able to react to obligations of other policies since the exact details of how obligations are enforced is not included.

10.7 Aspect Oriented Programming

Aspect-oriented programming (AOP) is another approach that has been used to address modularization of policies, in the context of cross-cutting concerns [17]. AOP allows code that would be distributed throughout an application to be separated into modules of related functionality, called aspects, that are woven into the application at specified locations. Aspect-oriented languages are typically Turing complete. However, we are not aware of any aspect-oriented languages that effectively handle conflicts or allow arbitrary policy combinators over atomic obligations. Composing aspects is a known challenge, as summarized in [22].

11 CONCLUSIONS

PoCo is a policy-specification language and enforcement system that enables principled composition of atomic obligations. It is Turing complete and supports effectful obligations of all types (pre-, post-, and ongoing) (Theorem 7). PoCo employs static analysis of obligations, based on their CFG representations, to allow policies to validate the obligations of other policies before they are executed (Theorem 4). PoCo also allows policies to react to the completed obligations of other policies (Theorems 5 and 6) and enables custom operators to define how policies (i.e., their votes and obligations) should be prioritized and combined. Taken together, these techniques enable versatile composition of security policies.

PoCo has been implemented and evaluated, including by defining the language’s formal semantics, using the semantics to establish important properties of the enforcement system,
and enforcing a case-study composition of ten policies for securing an email client.

REFERENCES

APPENDIX A THE POCO MONITOR ALGORITHM

Figure 8 presents the PoCo monitor algorithm. Situated between an untrusted application and the underlying executing system, the PoCo monitor interposes any attempt of executing security-relevant behaviors. Once an attempt is captured, the monitor performs the following steps:

1. Input security-relevant event e
2. Collect obligations from policies in response to e
3. While there are obligations to process
   a. Select an obligation o
   b. Allow policies to vote on o
   c. If o is approved, execute o
   d. Collect obligations triggered in response to o
4. If a new output event has been set, return it. Otherwise, output the original input event

The monitor is given as an expression parameterized by $\tau$ and config. An instance $\epsilon_{\text{monitor}}(\tau, (\text{evt} = e, \text{pols} = \text{pols}, \text{os} = \text{os}, \text{vc} = \text{vc}))$ monitors the security-relevant event $e$—whose return type is $\tau$—using policies $\text{pols}$, obligation scheduler $\text{os}$, and vote combinator $\text{vc}$.

```plaintext
$\epsilon_{\text{monitor}}(\tau, \text{config}) :=$
let $\text{pols} = \text{ref \hspace{1mm} config.pols}$ in
let $\text{obQueue} = \text{ref \hspace{1mm} []:(\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})_{\text{List}}}$ in
let $\text{obStack} = \text{ref \hspace{1mm} []:(\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})_{\text{ListList}}}$ in
while (¬empty($\text{pols}$)) {
  $\text{obQueue} := \text{!obQueue} @ (\text{pol} = \text{head}(\text{pols}), \text{cfg} =$
  $\text{makeCFG}(\text{inEv}(\text{evt}=\text{config.evt}, \text{onTrig}=\text{head}(\text{pols}).\text{onTrigger}): \text{Obligation})) :: \text{[]}:(\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})_{\text{List};}
  \text{pols} := \text{tail}(\text{pols});
} let $\text{votingPollist} = \text{call}(\text{config.os}, \text{!obQueue})$ in
$\text{obStack} := \text{votingPollist} :: \text{!obStack};$
while (¬empty($\text{obStack}$)) {
  $\text{obQueue} := \text{head}(\text{!obStack});$
  let $\text{ob} = \text{head}((\text{obQueue}).\text{cfg})$ in
  let $\text{votingPols} = \text{ref \hspace{1mm} votingPollist}$ in
  let $\text{votes} = \text{ref \hspace{1mm} []: \text{BoolList}}$ in
  if (¬empty(tail($\text{obQueue}$))) then $\text{obStack} := \text{tail}(\text{!obQueue}) :: \text{tail}(\text{!obStack})$
  else $\text{obStack} := \text{tail}(\text{!obStack});$
  while (¬empty($\text{votingPols}$)) {
    let $\text{pol} = \text{head}(\text{!votingPols}).\text{pol}$ in
    $\text{votingPols} := \text{tail}(\text{!votingPols});$
    $\text{votes} := \text{!votes} @ \text{call}(\text{pol}.\text{vote}, \text{ob}) :: \text{[]}:\text{BoolList}$
  end;
  if $\text{call}(\text{config.vc}, \text{!votes})$ then
    case $\text{ob}.\text{obligation} \text{of \hspace{1mm} ot \hspace{1mm} o1} \Rightarrow \text{call}(\text{ob1}.\text{onTrig}, \text{ob1}.\text{evt})$
    $\mid \text{oo \hspace{1mm} o2} \Rightarrow \text{call}(\text{ob2}.\text{onOblig}, \text{ob2}.\text{rt})$
    else unit;
  if (¬empty($\text{getRT()}$)) then
    $\text{votingPols} := \text{votingPollist};$
    $\text{obQueue} := \text{[]}:(\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})_{\text{List};}$
    while (¬empty($\text{votingPols}$)) {
      $\text{obQueue} := !\text{obQueue} @ (\text{pol} = \text{head}(\text{!votingPols}).\text{pol}, \text{cfg} = \text{makeCFG}(\text{inRt}$(\text{rt}=\text{getRT()}, \text{onOblig}=\text{head}(\text{!votingPols}).\text{onObligation}): \text{Obligation}))$
      :: \text{[]}:(\text{pol} : \text{Pol} \times \text{cfg} : \text{CFG})_{\text{List};}$
      $\text{votingPols} := \text{tail}(\text{!votingPols});$
    }
    $\text{obStack} := !\text{obQueue} :: !\text{obStack}$
  else unit
end end end end;
end end end end

$\text{case getOutput()}$ of some $\circ \Rightarrow \circ$ | none unit $\Rightarrow e$

Figure 8: The PoCo Monitor Algorithm
APPENDIX B  EXAMPLE POLICIES

Figures 9—14 present the example policies used throughout this paper written in the PoCo language.

```plaintext
(name = pol_file,
onTrigger = (fun ot(e:Event):Unit =
  case e of act a =>
    if a.name == "fopen" \&\& tryCast(String,a.arg) == "secret.txt"
    then setOutput(event(act("exit", makeTypedVal(Unit,unit))))
    else unit
  | res r => unit),
onObligation = (fun oo(rt: ResList):Unit = unit),
vote = (fun vt(cfg: CFG):Bool =
  ~call(containsAct, cfg = cfg, name = "fopen",
    arg=in_arg makeTypedVal(String, "secret.txt"):arg:TypedVal + none: unit, count=1) \&
  ~call(containsAct, cfg = cfg, name = "fopen",
    arg=in_name unit:(arg:TypedVal + none: unit, count=1)))
)

Figure 9: \textit{P}_{file} disallows users and obligations from opening the secret.txt file

fun disallow (x:Act) : Pol =
  name = disx,
onTrigger = (fun ot(e:Event):Unit =
    case e of act a =>
      if a.name == x.name \&\& a.arg == x.arg
      then setOutput(event(act("exit", makeTypedVal(Unit,unit))))
      else unit
    | res r => unit),
onObligation = (fun oo(rt: ResList):Unit = unit),
vote = (fun vt(cfg: CFG):Bool =
  ~call(containsAct, cfg=cfg, name=x.name, arg=in_arg(x.arg):(arg:TypedVal+none:Unit),count=1)
)

Figure 10: \textit{fun\_disallow} specifies a family of policies which disallow the action \textit{x}

(name = pol_postlog,
onTrigger = (fun ot(e:Event):Unit =
  case e of act a =>
    act a => unit
  | res r => if r.act.name == "fopen" then call(log,e) else unit),
onObligation = (fun oo(rt: ResList):Unit =
    let results=ref rt in
      while(\~empty(results)) {
        let event = head(results) in
          results := tail(results);
          if event.act.name == "fopen"
            then call(log,event)
          else unit
        end
      }
    end),
vote = (fun vt(cfg: CFG):Bool = true)
)

Figure 11: \textit{P}_{postlog} logs all file-opens after they occur
(name = pol_{prelog},
onTrigger = (fun ot(e: Event): Unit =
    case e of
        act a =>
            if a.name == "fopen" then
                if outputNotSet() then call(log,e); setOutput(e)
            else case getOutput() of
                event o =>
                    case o of
                        act a1 =>
                            if a1.name == e.name
                                then call(log,e) else unit
                        | res r => unit
                        | none n => unit
                end
            end
        onObligation = (fun oo(rt: ResList): Unit =
            let results = ref rt in
                while (¬empty(!results)) {
                    let event = head(!results) in
                        results := tail(!results);
                        if event.act.name == "fopen" then call(log,event) else unit
                }
            end
        vote = (fun vt(cfg: CFG): Bool = true)
    )
)

Figure 12: \(P_{prelog}\) logs file-open actions before they are executed

( name = pol_{confirm},
onTrigger = (fun ot(e: Event): Unit =
    case e of
        act a =>
            if a.name == "fopen" ∧ outputNotSet() then
                if call(popupConfirm,e)
                    then setOutput(e)
                else setOutput(inRes(res(a,makeTypedVal(Unit,unit))):Event)); unit
            else unit
        | res r => unit
    onObligation = (fun oo(rt: ResList): Unit = unit),

time = (fun vt(cfg: CFG): Bool = true)
)

Figure 13: \(P_{confirm}\) requires all file-open attempts to be confirmed by the user through a pop-up window

let t = ref 0 in

(name = pol_{time},
onTrigger = (fun ot(e: Event): Unit =
    case e of
        act a =>
            if a.name == "popup" then
                if currTime < !t+100 ∧ outputNotSet() then
                    setOutput(act("exit", makeTypedVal(Unit,unit)))
                else t := currTime
            else unit
        | res r => unit
    onObligation = (fun oo(rt: ResList): Unit = unit),
vote = fun vt(cfg: CFG): Bool =
    ¬call(containsActAnyArg, (cfg=cfg, name="popup", count=2)) ∧
    ¬{call(containsActAnyArg, (cfg=cfg, name="popup", count=1)) ∧
        currTime < (t+100))

Figure 14: \(P_{time}\) disallows pop-ups unless at least 100 seconds have passed since the last pop-up
### APPENDIX C  STATIC SEMANTICS OF POCO

This section presents the static semantics of the PoCo language. As defined in the syntax (see section 4.1), \( \lambda \) maps locations to values while \( \Gamma \) maps variables to values. It can be said that "under the contexts \( \Lambda \) and \( \Gamma \), the expression \( e \) is of type \( \tau \)" if and only if the judgment \( \Lambda, \Gamma \vdash e : \tau \) is derivable by the following rules:

\[
\begin{align*}
\Lambda, \Gamma \vdash e : \tau \\
\Lambda, \Gamma \vdash n : \text{Int} & \quad \text{(intVal)} \\
\Lambda, \Gamma \vdash \text{unit} : \text{Unit} & \quad \text{(unitVal)} \\
\Lambda, \Gamma \vdash e_1 : \text{Int} \quad \Lambda, \Gamma \vdash e_2 : \text{Int} & \quad \text{(add)} \\
\Lambda, \Gamma \vdash e : \text{Bool} & \quad \text{(negation)} \\
\Lambda, \Gamma \vdash \text{head}(e) : \tau \text{ Option} & \quad \text{(head)} \\
\Lambda, \Gamma \vdash \{ e \}_\text{let}(e) : \tau & \quad \text{(let)} \\
\Lambda, \Gamma \vdash e_1 : \ell \quad \Lambda, \Gamma \vdash e_2 : \ell & \quad \text{(sequence)} \\
\Lambda, \Gamma \vdash \text{let } x \equiv e_1 \text{ in } e_2 \text{ end } : \tau & \quad \text{(projection)} \\
\Lambda, \Gamma \vdash i \in \{1, \ldots, n\} & \quad \text{(projection)} \\
\Lambda, \Gamma \vdash e_1 : \text{Bool} \quad \Lambda, \Gamma \vdash e_2 : \tau \quad \Lambda, \Gamma \vdash e_3 : \tau & \quad \text{(if)} \\
\Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau & \\
\Lambda, \Gamma \vdash e_1 : \ell \quad \Lambda, \Gamma \vdash e_2 : \ell & \quad \text{(call)} \\
\Lambda, \Gamma \vdash e : \tau & \quad \text{(call)} \\
\Lambda, \Gamma \vdash \text{call}(e_1, e_2) : \tau & \\
\Lambda, \Gamma \vdash e : \tau & \quad \text{(cons)} \\
\Lambda, \Gamma \vdash e_1 : \text{Bool} \quad \Lambda, \Gamma \vdash e_2 : \text{Bool} & \\
\Lambda, \Gamma \vdash e : \tau & \quad \text{(createRef)} \\
\Lambda, \Gamma \vdash \text{ref } e : \tau & \\
\Lambda, \Gamma \vdash \text{setOutput}(e) : \text{Bool} & \quad \text{(setOutput)} \\
\Lambda, \Gamma \vdash \text{outputNotSet}() : \text{Bool} & \quad \text{(outputNotSet)} \\
\Lambda, \Gamma \vdash e : \tau & \quad \text{(makeTypedVal)} \\
\Lambda, \Gamma \vdash \text{makeTypedVal}(\tau, e) : \text{TypedVal} & \\
\Lambda, \Gamma \vdash e : \text{Event} & \quad \text{(getEvent)} \\
\Lambda, \Gamma \vdash e : \tau \text { Event } \times \text { pols } : \text { PolList } \times \text { as } : \text { OS } \times \text { uc } : \text { VC} & \quad \text{(monitor)} \\
\Lambda, \Gamma \vdash \text{monitor}(\tau, e) : \tau \text { Event} & \quad \text{(case)} \\
\Lambda, \Gamma \vdash \text{case } e \text { of } \ell_1 x_1 \Rightarrow e_1 \cdots \ell_n x_n \Rightarrow e_n : \tau & \\
\Lambda, \Gamma \vdash e_1 : \text{String} \quad \Lambda, \Gamma \vdash e_2 : \text{TypedVal} & \quad \text{(invoke)} \\
\Lambda, \Gamma \vdash \text{invoke}(e_1, e_2) : \text{TypedVal Option} & \\
\end{align*}
\]
The rules for $\Lambda, \Gamma \vdash e \text{ ok}$ are the same as for $\Lambda, \Gamma \vdash e : \tau$ (that is, $\Lambda, \Gamma \vdash e \text{ ok} \iff \exists \tau. \Lambda, \Gamma \vdash e : \tau$), except there are no equivalents for Rules label and monitor. $\Lambda, \Gamma \vdash e \text{ ok}$ is intended to be a user-level static semantics judgment form: a program is ok if and only if it is well typed and no subexpression of the program is a labeled expression $([e]_{s(v)})$ or a call to the monitor $(\text{monitor}(\tau, e))$. The dynamic semantics will allow an ok program to step to a non-ok (but still well-typed) program. For example, it may be the case that $(C, \text{invoke}("exit", v)) \rightarrow (C, [\cdots]_{\text{exit}(v)})$.

A list of monitored functions $F$ is ok iff for each pair $(s_i, f_i)$, $s_i$ is of type $\text{String}$ and $f_i$ is of type $\tau_1 \rightarrow \tau_2$, for some $\tau_1$ and $\tau_2$. We also assume that each monitored function name (i.e., each $s_i$) is unique in $F$.

**Assumption 1.** $F = (s_n, f_n), F' \Rightarrow s_n \notin \text{dom}(F')$

$$F = \{(s_1, f_1), \ldots, (s_n, f_n)\}$$

$$\forall i \in \{1, \ldots, n\} \quad \Lambda, \cdot \vdash s_i : \text{String}$$

$$\forall i \in \{1, \ldots, n\} \quad \exists \tau_1 \exists \tau_2 \quad \Lambda, \cdot \vdash f_i : \tau_1 \rightarrow \tau_2$$

($F\text{-ok}$)

$$\Lambda \vdash F \text{ ok}$$

To express type preservation, we must ensure the fidelity of configurations. Note how one of the premises in the following rule uses a type similar to $E\text{vent}$, but differs in that the $\text{result}$ field of the $\text{res}$ variant is of type $\tau_\text{out}$, not $\text{TypedVal}$. This is to ensure that obligations can only set the output result to be the return type of the current security-relevant function. See Rule setOutputNotSet.

$$M : \Lambda$$

$$M = \{(\ell_1, v_1), \ldots, (\ell_n, v_n)\}$$

$$\forall i \in \{1, \ldots, n\} \quad \Lambda, \cdot \vdash \ell_i : \tau_i$$

($\text{TMem}$)

$$\Lambda \vdash (M, R, \text{inOb}, \text{rt}, \text{out}) \text{ ok}$$

($\text{C-ok}$)

$$\Lambda \vdash (C, e) : \tau$$

($\text{TConfig}$)
APPENDIX D  DYNAMIC SEMANTICS

This section presents the dynamic semantics rules of the PoCo language using small-step operational semantics (SOS) with a left-to-right, call-by-value evaluation order. Each step creates a sequence of labels that are added to the execution trace. An absence of labels indicates an empty sequence, or no labels, is the result of the current step. We assume the existence of a procedure makeCFG, which, given an arbitrary value v of type Obligation, computes a value g of type CFG such that g represents the control-flow graph of v. Formally:

Assumption 2. ∀Ω, Λ ⊢ v : Obligation ⇒ Λ, ⊢ makeCFG( v ) : CFG

\[
\frac{\text{labels,...,labels} 
\to (C', e')}{(C, e) \to (C', e')} \tag{addE1}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e, e_2) \to (C', e', e'_2)} \tag{seqE2}
\]

\[
\frac{(C, e) \to (C', e')}{(C, t e) \to (C', !e')} \tag{drefE}
\]

\[
\frac{(C, e) \to (C', e')}{(C, ref e) \to (C', ref e')} \tag{refE}
\]

\[
\frac{(C, e) \to (C', e')}{(C, head(e)) \to (C', head(e'))} \tag{listHeadE}
\]

\[
\frac{(C, head(v_1 :: \cdots :: []) : \tauList) 
\to (\text{inSome}(v_1) : \tau \text{ Option})}{(C, head([]) : \tauList) 
\to (\text{inNone}(\text{unit}) : \tau \text{ Option})} \tag{listHeadNil}
\]

\[
\frac{(C, e) \to (C', e')}{(C, empty(e)) \to (C', empty(e'))} \tag{listEmptyE}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e @ e_2) \to (C', e'_1 @ e_2)} \tag{listAppendE}
\]

\[
\frac{(C, e_1 \land e_2) \to (C', e'_1 \land e_2)}{(C, e_1 \lor e_2) \to (C', e'_1 \lor e_2)} \tag{andE, orE}
\]

\[
\frac{(C, true \land e_2) \to (C, e_2)}{(C, true \lor e_2) \to (C, e_2)} \tag{andTrue, orTrue}
\]

\[
\frac{(C, false \land e_2) \to (C, e_2)}{(C, false \lor e_2) \to (C, e_2)} \tag{andFalse, orFalse}
\]

\[
\frac{(C, \neg false) \to (C, true)}{(C, \neg true) \to (C, false)} \tag{notE}
\]

\[
\frac{(C, n_1 + e_2) \to (C', n_1 + e'_2)}{(C, n_1 + n_2) \to (C, n_1 + n_2)} \tag{addValue}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e_2)} \tag{eqE1}
\]

\[
\frac{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e_2)}{(C, n_1 \equiv n_2) \to (C, true)} \tag{eqIntTrue}
\]

\[
\frac{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e_2)}{(C, n_1 \equiv n_2) \to (C, true)} \tag{eqEqTrue}
\]

\[
\frac{(C, s_1 \equiv s_2) \to (C, true)}{(C, s_1 \equiv s_2) \to (C, false)} \tag{eqStrTrue, eqStrFalse}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e_1; e_2) \to (C', e'_1; e_2)} \tag{sequenceE1}
\]

\[
\frac{(C, v_1; e_2) \to (C', v_1; e'_2)}{(C, v_1; e_2) \to (C', e'_2)} \tag{sequenceE2}
\]

\[
\frac{(C, e) \to (C', e')}{(C, t e) \to (C', !e')} \tag{drefE}
\]

\[
\frac{(C, e) \to (C', e')}{(C, ref e) \to (C', ref e')} \tag{refE}
\]

\[
\frac{(C, e) \to (C', e')}{(C, head(e)) \to (C', head(e'))} \tag{listHeadE}
\]

\[
\frac{(C, head(v_1 :: \cdots :: []) : \tauList) 
\to (\text{inSome}(v_1) : \tau \text{ Option})}{(C, head([]) : \tauList) 
\to (\text{inNone}(\text{unit}) : \tau \text{ Option})} \tag{listHeadNil}
\]

\[
\frac{(C, e) \to (C', e')}{(C, empty(e)) \to (C', empty(e'))} \tag{listEmptyE}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e'_2)} \tag{eqE1}
\]

\[
\frac{(C, e_1 \equiv e_2) \to (C', v_1 \equiv e'_1)}{(C, n_1 \equiv n_2) \to (C', v_1 \equiv e'_1)} \tag{eqEqE1}
\]

\[
\frac{(C, v_1 \equiv v_2) \to (C', v_1 \equiv v'_2)}{(C, false \equiv b_2) \to (C, b_2)} \tag{eqEqFalse}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e_2)} \tag{eqE1}
\]

\[
\frac{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e_2)}{(C, n_1 \equiv n_2) \to (C, true)} \tag{eqIntTrue}
\]

\[
\frac{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e_2)}{(C, n_1 \equiv n_2) \to (C, true)} \tag{eqEqTrue}
\]

\[
\frac{(C, s_1 \equiv s_2) \to (C, true)}{(C, s_1 \equiv s_2) \to (C, false)} \tag{eqStrTrue, eqStrFalse}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e_1; e_2) \to (C', e'_1; e_2)} \tag{sequenceE1}
\]

\[
\frac{(C, v_1; e_2) \to (C', v_1; e'_2)}{(C, v_1; e_2) \to (C', e'_2)} \tag{sequenceE2}
\]

\[
\frac{(C, e) \to (C', e')}{(C, t e) \to (C', !e')} \tag{drefE}
\]

\[
\frac{(C, e) \to (C', e')}{(C, ref e) \to (C', ref e')} \tag{refE}
\]

\[
\frac{(C, e) \to (C', e')}{(C, head(e)) \to (C', head(e'))} \tag{listHeadE}
\]

\[
\frac{(C, head(v_1 :: \cdots :: []) : \tauList) 
\to (\text{inSome}(v_1) : \tau \text{ Option})}{(C, head([]) : \tauList) 
\to (\text{inNone}(\text{unit}) : \tau \text{ Option})} \tag{listHeadNil}
\]

\[
\frac{(C, e) \to (C', e')}{(C, empty(e)) \to (C', empty(e'))} \tag{listEmptyE}
\]

\[
\frac{(C, e_1) \to (C', e'_1)}{(C, e_1 \equiv e_2) \to (C', e'_1 \equiv e'_2)} \tag{eqE1}
\]

\[
\frac{(C, e_1 \equiv e_2) \to (C', v_1 \equiv e'_1)}{(C, n_1 \equiv n_2) \to (C', v_1 \equiv e'_1)} \tag{eqEqE1}
\]

\[
\frac{(C, v_1 \equiv v_2) \to (C', v_1 \equiv v'_2)}{(C, false \equiv b_2) \to (C, b_2)} \tag{eqEqFalse}
\]
\[(C, \ell : \tau_\text{List}) @ \nu_3) \rightarrow (C, \nu_3) \quad \text{(appendNil)}\]

\[(C, e_1) \rightarrow (C', e'_1) \quad \text{(listPrependE1)}\]

\[(C, e_1 :: e_2) \rightarrow (C', e'_1 :: e_2) \quad \text{(assignE2)}\]

\[(C, \text{if true then } e_2 \text{ else } e_3) \rightarrow (C, e_2) \quad \text{((ifTrue)}\]

\[(C, e) \rightarrow (C', e') \quad \text{(projectionE)}\]

\[(C, e.\ell) \rightarrow (C', e'.\ell) \quad \text{(projectionV)}\]

\[(C, e_1) \rightarrow (C', e'_1) \quad \text{(assignE1)}\]

\[(C, e_1 :: e_2) \rightarrow (C', e'_1 :: e_2) \quad \text{(assignE2)}\]

\[(C, \text{let } x = e_1 \text{ in } e_2 \text{ end}) \rightarrow (C', \text{let } x = e'_1 \text{ in } e_2 \text{ end}) \quad \text{(letE)}\]

\[(C, \text{tryCast}(\tau, e_1)) \rightarrow (C', \text{tryCast}(\tau, e'_1)) \quad \text{(tryCastE)}\]

\[((\ldots, rt, out, \tau_{out}), \text{getRT}()) \rightarrow ((\ldots, rt, out, \tau_{out}), \text{rt}) \quad \text{(getRTVal)}\]

\[(C, e_1) \rightarrow (C', e'_1) \quad \text{(callE1)}\]

\[(C, \text{call}(e_1, e_2)) \rightarrow (C', \text{call}(e'_1, e_2)) \quad \text{(callE2)}\]

\[(C, \text{invoke}(e_1, e_2)) \rightarrow (C', \text{invoke}(e'_1, e_2)) \quad \text{(invokeE1)}\]

\[(C, e_1) \rightarrow (C', e'_1) \quad \text{(invokeE2)}\]

\[(C, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (C', \text{if } e'_1 \text{ then } e_2 \text{ else } e_3) \quad \text{((ifE)}\]

\[s \neq \text{"monitor"} \quad \text{(endLabelValue)}\]

\[\left((M, R, \text{false}, \ldots), \{ v_1 \}_{\ell(v_1)} \right) \quad \text{endPrepend}_{\nu_3 : v_1} \rightarrow \left((M, R, \text{false}, \ldots), \nu_1 \right) \quad \text{(endLabelValue)}\]

\[\left((\ldots, out, \tau_{out}), \text{outputNotSet}() \right) \rightarrow \left((\ldots, out, \tau_{out}), \text{true} \right) \quad \text{(outputNotSetTrue)}\]

\[\left((\ldots, out, \tau_{out}), \text{outputNotSet}() \right) \rightarrow \left((\ldots, out, \tau_{out}), \text{false} \right) \quad \text{(outputNotSetFalse)}\]

\[\forall (s', f, s) \in F, s \neq s' \quad \text{(invokeValNotExists)}\]

\[\left((M, (\ldots, \ldots), \text{invoke}(s_1, v_2)) \right) \rightarrow \left((M, (\ldots, \ldots), \text{in}_\text{none}(\text{unit}) : \text{TypedVal Option}) \quad \text{(makeTypedValE)}\]

\[(C, e) \rightarrow (C', e') \quad \text{(makeTypedValE)}\]

\[(C, \text{tryCast}(\tau, \text{makeTypedVal}(\tau, e))) \rightarrow (C, \text{in}_\text{some}(e) : \tau \text{ Option}) \quad \text{(tryCastVOk)}\]

\[(C, \text{tryCast}(\tau_1, \text{makeTypedVal}(\tau_2, v))) \rightarrow (C, \text{in}_\text{some}(v) : \tau \text{ Option}) \quad \text{(tryCastVBad)}\]

\[(s_1, \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e) \in F \quad \text{v}_2 = \text{makeTypedVal}(\tau_1, \text{v}_2) \quad \text{invokeValueExistsOk}\]

\[(\left((M, (\ldots, \ldots), \text{invoke}(s_1, \text{v}_2)) \right) \rightarrow \left((M, (\ldots, \ldots), \text{in}_\text{some}(\text{makeTypedVal}(\tau_2, \text{call}(\text{fun } x_1(x_2 : \tau_1) : \tau_2[e], \text{v}_2')) : \text{TypedVal Option}) \quad \text{(invokeValueExistsOk)}\]

\[(s_1, \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e) \in F \quad \text{v}_2 = \text{makeTypedVal}(\tau_3, \text{v}_2') \quad \tau_1 \neq \tau_3 \quad \text{(invokeValueExistsBad)}\]

\[(C, \text{while}(e_1) \{ e_2 \}) \rightarrow (C, \text{if } e_1 \text{ then } e_2; \text{while}(e_2) \{ e_2 \}) \text{ else } \text{false} \quad \text{(whileE)}\]
\[
\begin{align*}
(name = s, onTrigger = f_1, onObligation = f_2, vote = f_3) \in \text{pols} & \quad f_1 = (\text{fun } x_1(x_2 : \text{Event}) : \text{Unit}) = e \\
(M, (F, \text{pols}, os, vc), \text{inOb}, rt, out, \tau_{\text{out}}, \text{call}(f_1, v)) \quad \text{begin}_{f_1(v)} & \\
(M, (F, \text{pols}, os, vc), \text{inOb}, []) : \text{Res}_{\text{List}}. \text{out}, \tau_{\text{out}}). \text{call}(f_1, v) & \\
(name = s, onTrigger = f_1, onObligation = f_2, vote = f_3) \in \text{pols} & \quad f_2 = (\text{fun } x_1(x_2 : \text{Res}_{\text{List}}) : \text{Unit}) = e \\
(M, (F, \text{pols}, os, vc), \text{inOb}, []) : \text{Res}_{\text{List}}. \text{out}, \tau_{\text{out}}). \text{call}(f_2, v) & \\
(s, f) \in F & \quad f = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e) \\
(M, (F, \text{pols}, os, vc), \text{false}, rt, out, \tau_{\text{old}}). \text{call}(f, v) \quad \text{begin}_{v_f} & \\
(M, (F, \text{pols}, os, vc), \text{false}, rt, \text{in}_{\text{none}}(\text{unit}) : \tau_2 \text{ Event Option}, \tau_2). \text{e}_{\text{procEvt}} & \\
\end{align*}
\]

Where \( e_{\text{procEvt}} = \)
\[
\begin{align*}
| \text{let } aux = (\text{fun } aux(\text{event} : \tau_2 \text{ Event}) : \tau_2 \text{ Res} = \\
\quad \text{case } \text{event} \text{ of} & \\
\quad \quad \text{act } a \Rightarrow & \\
\quad \quad \text{case } \text{invoke}(a.\text{name}, a.\text{arg}) \text{ of} & \\
\quad \quad \quad \text{some } r \Rightarrow & \\
\quad \quad \quad \text{case } \text{tryCast}(\tau_2, r_1) \text{ of} & \\
\quad \quad \quad \quad \text{some } v_1 \Rightarrow & \\
\quad \quad \quad \quad \text{let } \text{action_output} = \text{in}_{\text{res}}(a.\text{res}, v_1) : \tau_2 \text{ Event in} & \\
\quad \quad \quad \quad \text{let } \text{mon_output} = \text{monitor}(\tau_2, (\text{ev}t = \text{action_output}, \text{pols} = \text{pols}, os = os, vc = vc)) \text{ in} & \\
\quad \quad \quad \quad \text{call}(aux, \text{mon_output}) & \\
\quad \quad \quad \text{end} & \\
\quad \quad | \text{none } u_1 \Rightarrow \text{call}(aux, \text{event}) & \\
\quad \quad | \text{none } u_2 \Rightarrow \text{call}(aux, \text{event}) & \\
\quad \quad | \text{res } \tau_2 \Rightarrow \tau_2) & \\
\quad \text{end}_1(v_f) & \\
\end{align*}
\]
APPENDIX E PROOF OF TYPE SAFETY

A proof of PoCo’s type-safety is presented below. There are seven main lemmas: Λ-Weakening (page 27), Weakening (page 31), Substitution (page 34), Typing Rule Inversion (page 39), Canonical Forms (page 43), Progress (page 46), and Preservation (page 56) Lemmas. Throughout the proofs, "IH" refers to the inductive hypothesis.

Lemma 1 (C-Inversion).

\[ \Lambda \vdash (M, R, \text{inOb}, \text{rt}, \tau_{\text{out}}) \text{ ok } \Rightarrow M : \Lambda \land \Lambda \vdash R \text{ ok } \land \Lambda, \bullet \vdash \text{inOb} : \text{Bool } \land \Lambda, \bullet \vdash \text{rt} : \text{ResList } \land \Lambda, \bullet \vdash \tau_{\text{out}} : \text{Event Option} \]

Proof.

1. \( \Lambda \vdash (M, R, \text{inOb}, \text{rt}, \tau_{\text{out}}) \text{ ok } \) assumption
2. 1 is only derivable with Rule C-ok Inspection of \( \Lambda \vdash \text{C ok} \) rules
3. \( M : \Lambda \) 2, Inversion of Rule C-ok
4. \( \Lambda \vdash R \text{ ok } \) 2, Inversion of Rule C-ok
5. \( \Lambda, \bullet \vdash \text{inOb} : \text{Bool } \) 2, Inversion of Rule C-ok
6. \( \Lambda, \bullet \vdash \text{rt} : \text{ResList } \) 2, Inversion of Rule C-ok
7. \( \Lambda, \bullet \vdash \tau_{\text{out}} : \text{Event Option } \) 2, Inversion of Rule C-ok

Result is from 3-7

□

Lemma 2 (R-Inversion).

\[ \Lambda \vdash (F, \text{pols}, \text{os}, \text{vc}) \text{ ok } \Rightarrow \Lambda \vdash F \text{ ok } \land \Lambda, \bullet \vdash \text{pols} : \text{PolList } \land \Lambda, \bullet \vdash \text{os} : \text{OS } \land \Lambda, \bullet \vdash \text{vc} : \text{VC} \]

Proof.

1. \( \Lambda \vdash (F, \text{pols}, \text{os}, \text{vc}) \text{ ok } \) assumption
2. 1 is only derivable by Rule R-ok Inspection of \( \Lambda \vdash \text{R ok} \) rules
3. \( \Lambda \vdash F \text{ ok } \) 2, Inversion of Rule R-ok
4. \( \Lambda, \bullet \vdash \text{pols} : \text{PolList } \) 2, Inversion of Rule R-ok
5. \( \Lambda, \bullet \vdash \text{os} : \text{OS } \) 2, Inversion of Rule R-ok
6. \( \Lambda, \bullet \vdash \text{vc} : \text{VC } \) 2, Inversion of Rule R-ok

Result is from 3-6

□

Lemma 3 (C-Weakening).

\[ \Lambda \vdash (M, (F, \text{pols}, \text{os}, \text{vc}), \text{inOb}, \text{rt}, \tau_{\text{out}}) \text{ ok } \land \Lambda' \land \Lambda \subseteq \Lambda' \Rightarrow \Lambda' \vdash (M', (F, \text{pols}, \text{os}, \text{vc}), \text{inOb}, \text{rt}, \tau_{\text{out}}) \text{ ok } \]

Proof.
By induction on the derivation of Case Λ

1. A ⊢ (M, (F, pols, os, vc), inOb, rt) ok assumption
2. M' : A' assumption
3. A ⊆ A' assumption
4. A ⊢ (F, pols, os, vc) ok 1, C-Inversion Lemma
5. A ⊢ F ok 4, R-Inversion Lemma
6. A, • ⊢ pols : PolList 4, R-Inversion Lemma
7. A, • ⊢ os : OS 4, R-Inversion Lemma
8. A, • ⊢ vc : VC 4, R-Inversion Lemma
9. A, • ⊢ inOb : Bool 1, C-Inversion Lemma
10. A, • ⊢ rt : ResList 1, C-Inversion Lemma
11. A, • ⊢ out : τout Event Option 1, C-Inversion Lemma
12. A', • ⊢ pols : PolList 3, 6, Lemma Λ-Weakening
13. A', • ⊢ os : OS 3, 7, Lemma Λ-Weakening
15. A', • ⊢ inOb : Bool 3, 9, Lemma Λ-Weakening
17. A', • ⊢ out : τout Event Option 3, 11, Lemma Λ-Weakening
18. (5) is only derivable with Rule F-ok Inspection of A ⊢ F ok rules
19. F = {(s1, f1), ..., (sn, fn)} 5, 18, Inversion of Rule F-ok
20. Vi ∈ {1, ..., n}, ?r1, r2, A, • ⊢ f1 : r1 → r2 5, 18, Inversion of Rule F-ok
21. Vi ∈ {1, ..., n}, A, • ⊢ s1 : String 5, 18, Inversion of Rule F-ok
22. Vi ∈ {1, ..., n}, ?r1, r2, ?', • ⊢ f1 : r1 → r2 3, 20, Lemma Λ-Weakening
23. Vi ∈ {1, ..., n}, A', • ⊢ s1 : String 3, 21, Lemma Λ-Weakening
24. A' ⊢ F ok 19, 22, 23, Rule F-ok
25. A' + (M, (F, pols, os, vc), inOb, rt, out, τout) ok 12-14, 22, Rule R-ok
26. A' + (M, (F, pols, os, vc), inOb, rt, out, τout) ok 15-17, 23, Rule C-ok

Lemma 4 (Λ-Weakening). (∆1, Γ ⊢ e : τ ∧ ∆1 ⊆ ∆2) ⇒ ∆2, Γ ⊢ e : τ

Proof. By induction on the derivation of ∆1, Γ ⊢ e : τ
Case $\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \text{Bool}$

1. $\Gamma \vdash e_1 : \text{Bool}$ assumption
2. $\Gamma \vdash e_2 : \text{Bool}$ assumption
3. $\Gamma \subseteq \Delta_2$ assumption
4. $\Delta_2, \Gamma \vdash e_1 : \text{Bool}$ 1, 3, IH
5. $\Delta_2, \Gamma \vdash e_2 : \text{Bool}$ 2, 3, IH
6. $\Delta_2, \Gamma \vdash e_1 \vee e_2 : \text{Bool}$ 4, 5, Rule dis

Case $\Gamma \vdash e : \text{Bool}$

1. $\Gamma \vdash e : \text{Bool}$ assumption
2. $\Gamma \subseteq \Delta_2$ assumption
3. $\Delta_2, \Gamma \vdash e : \text{Bool}$ 1, 2, IH
4. $\Delta_2, \Gamma \vdash \neg e : \text{Bool}$ 3, Rule negation

Case $\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\text{Int, Bool, String}\}$

1. $\Gamma \vdash e_1 : \tau$ assumption
2. $\Gamma \vdash e_2 : \tau$ assumption
3. $\tau \in \{\text{Int, Bool, String}\}$ assumption
4. $\Gamma \subseteq \Delta_2$ assumption
5. $\Delta_2, \Gamma \vdash e_1 : \tau$ 1, 4, IH
6. $\Delta_2, \Gamma \vdash e_2 : \tau$ 2, 4, IH
7. $\Delta_2, \Gamma \vdash e_1 = e_2 : \text{Bool}$ 3, 5, 6, Rule equality

Case $\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}$

1. $\Gamma \subseteq \Delta_2$ assumption
2. $\Gamma \vdash e_1 : \text{Int}$ assumption
3. $\Gamma \vdash e_2 : \text{Int}$ assumption
4. $\Gamma \vdash e_1 : \text{Int}$ 1, 2, IH
5. $\Gamma \vdash e_2 : \text{Int}$ 1, 3, IH
6. $\Delta_2, \Gamma \vdash e_1 + e_2 : \text{Int}$ 4, 5, Rule add

Case $\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2$

1. $\Gamma \vdash e_1 : \tau_1$ assumption
2. $\Gamma \vdash e_2 : \tau_2$ assumption
3. $\Gamma \subseteq \Delta_2$ assumption
4. $\Delta_2, \Gamma \vdash e_1 : \tau_1$ 1, 3, IH
5. $\Delta_2, \Gamma \vdash e_2 : \tau_2$ 2, 3, IH
6. $\Delta_2, \Gamma \vdash e_1 \cdot e_2 : \tau_2$ 4, 5, Rule sequence

Case $\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau$

1. $\Gamma \vdash e_1 : \text{Bool}$ assumption
2. $\Gamma \vdash e_2 : \tau$ assumption
3. $\Gamma \vdash e_3 : \tau$ assumption
4. $\Gamma \subseteq \Delta_2$ assumption
5. $\Delta_2, \Gamma \vdash e_1 : \text{Bool}$ 1, 4, IH
6. $\Delta_2, \Gamma \vdash e_2 : \tau$ 2, 4, IH
7. $\Delta_2, \Gamma \vdash e_3 : \tau$ 3, 4, IH
8. $\Delta_2, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau$ 5-7, Rule if

Case $\Gamma \vdash e : \tau$

1. $\Gamma \vdash e : \tau$ assumption
2. $\Gamma \subseteq \Delta_2$ assumption

Case $\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end : } \tau_2$

1. $\Gamma \vdash e_1 : \tau_1$ assumption
2. $\Gamma \vdash e_2 : \tau_2$ assumption
3. $\Gamma \subseteq \Delta_2$ assumption
4. $\Delta_2, \Gamma \vdash e_1 : \tau_1$ 1, 3, IH
5. $\Delta_2, \Gamma \vdash e_2 : \tau_2$ 2, 3, IH
6. $\Delta_2, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end : } \tau_2$ 4, 5, Rule let

Case $\Gamma \vdash e : \tau$

1. $\Gamma \vdash e : \tau$ assumption
2. $\Gamma \subseteq \Delta_2$ assumption
3. $\Delta_2, \Gamma \vdash e : \tau$ 1, 2, IH
4. $\Delta_2, \Gamma \vdash e : \tau$ 3, Rule createRef

Case $\Gamma \vdash e : \tau$

1. $\Gamma \vdash e : \tau$ assumption
2. $\Gamma \subseteq \Delta_2$ assumption
3. $\Delta_2, \Gamma \vdash e : \tau$ 1, 2, IH
4. $\Delta_2, \Gamma \vdash e : \tau$ 3, Rule accessRef

Case $\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2$

1. $\Gamma \vdash e_1 : \tau_1$ assumption
2. $\Gamma \vdash e_2 : \tau_2$ assumption
3. $\Gamma \subseteq \Delta_2$ assumption
4. $\Delta_2, \Gamma \vdash e_1 : \tau_1$ 1, 3, IH
5. $\Delta_2, \Gamma \vdash e_2 : \tau_2$ 2, 3, IH
6. $\Delta_2, \Gamma \vdash e_1 := e_2 : \text{Unit}$ 4, 5, Rule assignment

Case $\Gamma \vdash e : \tau$

1. $\Gamma \vdash e : \tau$ assumption
2. $\Gamma \subseteq \Delta_2$ assumption
3. $\Delta_2, \Gamma \vdash e : \tau$ 1, 3, IH
4. $\Delta_2, \Gamma \vdash e : \tau$ 2, 3, IH
5. $\Delta_2, \Gamma \vdash e : \tau$ 4, 5, Rule assignment

Case $\Gamma \vdash [\ ] : \tau_{\text{List}}$

1. $\Gamma \vdash [\ ] : \tau_{\text{List}}$ Rule listEmptyVal

Case $\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2$

1. $\Gamma \vdash e_1 : \tau_1$ assumption
2. $\Gamma \vdash e_2 : \tau_2$ assumption
3. $\Gamma \subseteq \Delta_2$ assumption
4. $\Delta_2, \Gamma \vdash e_1 : \tau_{\text{List}}$ 1, 3, IH
5. $\Delta_2, \Gamma \vdash e_2 : \tau_{\text{List}}$ 2, 3, IH
6. $\Delta_2, \Gamma \vdash e_1 :: e_2 : \tau_{\text{List}}$ 4, 5, Rule listCons
Case \( \Lambda_1, \Gamma \vdash e_1 : \tau_{\text{List}} \quad \Lambda_1, \Gamma \vdash e_2 : \tau_{\text{List}} \quad \text{(listAppend)} \)

1. \( \Lambda_1, \Gamma \vdash e_1 : \tau_{\text{List}} \quad \text{assumption} \)
2. \( \Lambda_1, \Gamma \vdash e_2 : \tau_{\text{List}} \quad \text{assumption} \)
3. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
4. \( \Lambda_2, \Gamma \vdash e_1 : \tau_{\text{List}} \quad 1, 3, \text{IH} \)
5. \( \Lambda_2, \Gamma \vdash e_2 : \tau_{\text{List}} \quad 2, 3, \text{IH} \)
6. \( \Lambda_2, \Gamma \vdash e_1 \circ e_2 : \tau_{\text{List}} \quad 4, 5, \text{Rule listAppend} \)

Case \( \Lambda_1, \Gamma \vdash e : \tau_{\text{List}} \quad \text{(head)} \)

1. \( \Lambda_1, \Gamma \vdash e : \tau_{\text{List}} \quad \text{assumption} \)
2. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
3. \( \Lambda_2, \Gamma \vdash e : \tau_{\text{List}} \quad 1, 2, \text{IH} \)
4. \( \Lambda_2, \Gamma \vdash \text{head}(e) : \tau_{\text{Option}} \quad 3, \text{Rule head} \)

Case \( \Lambda_1, \Gamma \vdash e : \tau_{\text{List}} \quad \text{(tail)} \)

1. \( \Lambda_1, \Gamma \vdash e : \tau_{\text{List}} \quad \text{assumption} \)
2. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
3. \( \Lambda_2, \Gamma \vdash e : \tau_{\text{List}} \quad 1, 2, \text{IH} \)
4. \( \Lambda_2, \Gamma \vdash \text{tail}(e) : \tau_{\text{List}} \quad 3, \text{Rule tail} \)

Case \( \Lambda_1, \Gamma \vdash e : \tau_{\text{List}} \quad \text{(empty)} \)

1. \( \Lambda_1, \Gamma \vdash e : \tau_{\text{List}} \quad \text{assumption} \)
2. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
3. \( \Lambda_2, \Gamma \vdash e : \tau_{\text{List}} \quad 1, 2, \text{IH} \)
4. \( \Lambda_2, \Gamma \vdash \text{empty}(e) : \text{Bool} \quad 3, \text{Rule empty} \)

Case \( \Lambda_1, \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda_1, \Gamma \vdash e_n : \tau_n \quad \text{(record)} \)

1. \( \Lambda_1, \Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad \text{assumption} \)
2. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
3. \( \Lambda_2, \Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : 1, 2, \text{IH} \)
4. \( \Lambda_2, \Gamma \vdash (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) : 4, \text{Rule record} \)

Case \( \Lambda_1, \Gamma \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\} \quad \text{(projection)} \)

1. \( \Lambda_1, \Gamma \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad \text{assumption} \)
2. \( i \in \{1, \ldots, n\} \quad \text{assumption} \)
3. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
4. \( \Lambda_2, \Gamma \vdash (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) : 1, 3, \text{IH} \)
5. \( \Lambda_2, \Gamma \vdash e_{\ell_i} : \tau_i \quad 2, 4, \text{Rule projection} \)

Case \( \Lambda_1, \Gamma \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\} \quad \text{(variant)} \)

1. \( \Lambda_1, \Gamma \vdash e_i : \tau_i \quad \text{assumption} \)
2. \( i \in \{1, \ldots, n\} \quad \text{assumption} \)
3. \( \Lambda_1 \subseteq \Lambda_2 \quad \text{assumption} \)
4. \( \Lambda_2, \Gamma \vdash e_i : \tau_i \quad 1, 3, \text{IH} \)
5. \( \Lambda_2, \Gamma \vdash (\ell_i, e_i : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)) : 2, 4, \text{Rule variant} \)

\[ (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \]
Case \[
\begin{align*}
\Lambda_1, \Gamma \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) & \quad \Lambda_1, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda_1, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau
\end{align*}
\] (case)

1. \[
\Lambda_1, \Gamma \vdash (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \text{assumption}
\]
2. \[
\Lambda_1, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda_1, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau
\] assumption
3. \[
\Lambda_1 \subseteq \Lambda_2
\] assumption
4. \[
\Lambda_2, \Gamma \vdash (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad 1, 3, \text{IH}
\]
5. \[
\Lambda_2, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda_2, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau \quad 2, 3, \text{IH}
\]
6. \[
\Lambda_2, \Gamma \vdash (\text{case } e \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau
\] 4, 5, Rule case

Case

1. \[
\Lambda_1, \Gamma \vdash e_1 : \tau
\] assumption
2. \[
\Lambda_1 \subseteq \Lambda_2
\] assumption
3. \[
\Lambda_2, \Gamma \vdash \text{makeTypedVal}(\tau, e_1) : \text{TypedVal}
\] 3, Rule makeTypedVal

Case \[
\begin{align*}
\Lambda_1, \Gamma \vdash e' : (ev \tau' \text{ Event} \times \text{Pols} : \text{PolList} \times \text{os} : \text{OS} \times \text{vc} : \text{VC})
\end{align*}
\] (monitor)

1. \[
\Lambda_1, \Gamma \vdash e' : (ev \tau' \text{ Event} \times \text{Pols} : \text{PolList} \times \text{os} : \text{OS} \times \text{vc} : \text{VC}) \quad \text{assumption}
\]
2. \[
\Lambda_1 \subseteq \Lambda_2
\] assumption
3. \[
\Lambda_2, \Gamma \vdash e' : (ev \tau' \text{ Event} \times \text{Pols} : \text{PolList} \times \text{os} : \text{OS} \times \text{vc} : \text{VC}) \quad 1, 2, \text{IH}
\]
4. \[
\Lambda_2, \Gamma \vdash \text{monitor}(\tau', e') : \tau' \text{ Event}
\] 3, Rule monitor
Lemma 5 (Weakening). $\Lambda, \Gamma_1 \vdash e : \tau \land \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Lambda, \Gamma_2 \vdash e : \tau$

Proof. By induction on the derivation of $\Lambda, \Gamma_1 \vdash e : \tau$

Case $\Lambda, \Gamma_1 \vdash n : Int$ \hspace{1cm} (intVal)
1. $\Lambda, \Gamma_2 \vdash n : Int$ \hspace{1cm} Rule intVal

Case $\Lambda, \Gamma_1 \vdash b : Bool$ \hspace{1cm} (boolVal)
1. $\Lambda, \Gamma_2 \vdash b : Bool$ \hspace{1cm} Rule boolValue

Case $\Lambda, \Gamma_1 \vdash s : String$ \hspace{1cm} (stringVal)
1. $\Lambda, \Gamma_2 \vdash s : String$ \hspace{1cm} Rule stringVal

Case $\Lambda, \Gamma_1 \vdash \text{unit} : Unit$ \hspace{1cm} (unitVal)
1. $\Lambda, \Gamma_2 \vdash \text{unit} : Unit$ \hspace{1cm} Rule unitVal

Case $\Lambda, \Gamma_1 \cup \{ x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1 \} \vdash e' : \tau_2$ \hspace{1cm} (fun)
1. $e = \text{fun} x_1 (x_2 : \tau_1) : \tau_2 = e'$ \hspace{1cm} assumption
2. $\Gamma_1 \cup \{ x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1 \} \vdash e' : \tau_2$ \hspace{1cm} assumption
3. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
4. $\Gamma_1 \cup \{ x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1 \} \subseteq$ \hspace{1cm} 3, definition of $\subseteq$
5. $\Gamma_2 \cup \{ x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1 \} \vdash e' : \tau_2$ \hspace{1cm} 2, 4, IH
6. $\Lambda, \Gamma_2 \vdash e : \tau_1 \rightarrow \tau_2$ \hspace{1cm} 1, 5, Rule fun

Case $\Lambda', \Gamma_1' \cup \{ x : \tau \} \vdash x : \tau$ \hspace{1cm} (var)
1. $\Gamma_1 = \Gamma_1' \cup \{ x : \tau \}$ \hspace{1cm} assumption
2. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
3. $\{ x : \tau \} \subseteq \Gamma_1$ \hspace{1cm} 1, definition of $\subseteq$
4. $\{ x : \tau \} \subseteq \Gamma_2$ \hspace{1cm} 2, 3, definition of $\subseteq$
5. $\Gamma_2 = \Gamma_1' \cup \{ x : \tau \}$ \hspace{1cm} 4
6. $\Lambda, \Gamma_2 \vdash x : \tau$ \hspace{1cm} 5, Rule var

Case $\Lambda, \Gamma_1 \vdash e_1 : Bool \hspace{1cm} \Lambda, \Gamma_1 \vdash e_2 : Bool$ \hspace{1cm} (con)
1. $\Lambda, \Gamma_1 \vdash e_1 : Bool$ \hspace{1cm} assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : Bool$ \hspace{1cm} assumption
3. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Bool$ \hspace{1cm} 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : Bool$ \hspace{1cm} 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 \land e_2 : Bool$ \hspace{1cm} 4, 5, rule con

Case $\Lambda, \Gamma_1 \vdash e_1 : Bool \hspace{1cm} \Lambda, \Gamma_1 \vdash e_2 : Bool$ \hspace{1cm} (dis)
1. $\Lambda, \Gamma_1 \vdash e_1 : Bool$ \hspace{1cm} assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : Bool$ \hspace{1cm} assumption
3. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Bool$ \hspace{1cm} 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : Bool$ \hspace{1cm} 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 \lor e_2 : Bool$ \hspace{1cm} 4, 5, Rule dis

Case $\Lambda, \Gamma_1 \vdash e : Bool$ \hspace{1cm} (negation)
1. $\Lambda, \Gamma_1 \vdash e : Bool$ \hspace{1cm} assumption
2. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
3. $\Lambda, \Gamma_2 \vdash e : Bool$ \hspace{1cm} 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash \lnot e : Bool$ \hspace{1cm} 3, Rule negation

Case $\Lambda, \Gamma_1 \vdash e_1 : \tau \hspace{1cm} \Lambda, \Gamma_1 \vdash e_2 : \tau \in \{ Int, Bool, String \}$ \hspace{1cm} (equality)
1. $\Lambda, \Gamma_1 \vdash e_1 : \tau$ \hspace{1cm} assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau$ \hspace{1cm} assumption
3. $\tau \in \{ Int, Bool, String \}$ \hspace{1cm} assumption
4. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
5. $\Lambda, \Gamma_2 \vdash e_1 : \tau$ \hspace{1cm} 1, 4, IH
6. $\Lambda, \Gamma_2 \vdash e_2 : \tau$ \hspace{1cm} 2, 4, IH
7. $\Lambda, \Gamma_2 \vdash e_1 \equiv e_2 : Bool$ \hspace{1cm} 3, 5, 6, Rule equality

Case $\Lambda, \Gamma_1 \vdash e_1 : Int \hspace{1cm} \Lambda, \Gamma_1 \vdash e_2 : Int$ \hspace{1cm} (add)
1. $\Lambda, \Gamma_1 \vdash e_1 : Int$ \hspace{1cm} assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : Int$ \hspace{1cm} assumption
3. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Int$ \hspace{1cm} 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : Int$ \hspace{1cm} 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 + e_2 : Int$ \hspace{1cm} 4, 5, Rule add

Case $\Lambda, \Gamma_1 \vdash e_1 : t_1 \hspace{1cm} \Lambda, \Gamma_1 \vdash e_2 : t_2$ \hspace{1cm} (seq)
1. $\Lambda, \Gamma_1 \vdash e_1 : t_1$ \hspace{1cm} assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : t_2$ \hspace{1cm} assumption
3. $\Gamma_1 \subseteq \Gamma_2$ \hspace{1cm} assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : t_1$ \hspace{1cm} 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : t_2$ \hspace{1cm} 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 ; e_2 : t_2$ \hspace{1cm} 4, 5, Rule sequence
\[
\begin{align*}
\text{Case} & \quad \Lambda, \Gamma \vdash e_1 : \text{Bool} & \Lambda, \Gamma \vdash e_2 : \tau & \Lambda, \Gamma \vdash e_3 : \tau \\
& \quad \text{(if)} \\
& \quad \Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \\
& \quad 1. \quad \Lambda, \Gamma \vdash e_1 : \text{Bool} & \text{assumption} \\
& \quad 2. \quad \Lambda, \Gamma \vdash e_2 : \tau & \text{assumption} \\
& \quad 3. \quad \Gamma \subseteq \Gamma_2 & \text{assumption} \\
& \quad 5. \quad \Lambda, \Gamma \vdash e_1 : \text{Bool} & 1, 4, \text{IH} \\
& \quad 6. \quad \Lambda, \Gamma \vdash e_2 : \tau & 2, 4, \text{IH} \\
& \quad 7. \quad \Lambda, \Gamma \vdash e_3 : \tau & 3, 4, \text{IH} \\
& \quad 8. \quad \Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau & 5, 6, 7, \text{Rule If} \\
\end{align*}
\]
\[
\begin{align*}
\text{Case} & \quad \Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_1 \vdash e_n : \tau_n \\
\hline
1. & \quad \Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_1 \vdash e_n : \tau_n \\
2. & \quad \Gamma_1 \subseteq \Gamma_2 \\
3. & \quad \Lambda, \Gamma_2 \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_2 \vdash e_n : \tau_n \\
4. & \quad (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \\
\end{align*}
\]

\[
\text{Case} \quad \Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in [1, \ldots, n]
\]

\[
\begin{align*}
1. & \quad \Lambda, \Gamma_1 \vdash e_i : \tau_i \\
2. & \quad \Gamma_1 \subseteq \Gamma_2 \\
3. & \quad i \in [1, \ldots, n] \\
4. & \quad \Lambda, \Gamma_2 \vdash e_i : \tau_i \\
5. & \quad \ell_i \vdash \tau_i \\
\end{align*}
\]

\[
\text{Case} \quad \Lambda, \Gamma_1 \vdash \{ e \} \mid \text{label} : \tau 
\]

\[
\begin{align*}
1. & \quad \Lambda, \Gamma_1 \vdash e : \tau \\
2. & \quad \Gamma_1 \subseteq \Gamma_2 \\
3. & \quad \Lambda, \Gamma_2 \vdash e : \tau \\
4. & \quad \ell \vdash e \mid \text{label} : \tau \\
\end{align*}
\]

\[
\text{Case} \quad \Lambda, \Gamma_1 \vdash e : \text{Obligation} 
\]

\[
\begin{align*}
1. & \quad \Lambda, \Gamma_1 \vdash e : \text{Obligation} \\
2. & \quad \Gamma_1 \subseteq \Gamma_2 \\
3. & \quad \Lambda, \Gamma_2 \vdash e : \text{Obligation} \\
4. & \quad \ell \vdash e \mid \text{label} : \tau \\
\end{align*}
\]

\[
\text{Case} \quad \Lambda, \Gamma_1 \vdash \text{getRT}(\ell) : \text{ResList} 
\]

\[
\begin{align*}
1. & \quad \Lambda, \Gamma_2 \vdash \text{getRT}(\ell) : \text{ResList} \\
\end{align*}
\]

\[
\text{Case} \quad \Lambda, \Gamma_1 \vdash \text{makeCFG}(\ell) : \text{CFG} 
\]

\[
\begin{align*}
1. & \quad \Lambda, \Gamma_2 \vdash \text{makeCFG}(\ell) : \text{CFG} \\
\end{align*}
\]
Case \( \Lambda, \Gamma \vdash e \cdot \text{Event} \) (setOutput)

1. \( \Lambda, \Gamma \vdash e \cdot \text{Event} \) assumption
2. \( \Gamma_1 \subseteq \Gamma_2 \) assumption
3. \( \Lambda, \Gamma_2 \vdash e \cdot \text{Event} \) 1, 2, IH
4. \( \Lambda, \Gamma_2 \vdash \text{setOutput}(e) \cdot \text{Bool} \) 3, Rule setOutput

Case \( \Lambda, \Gamma \vdash \text{getOutput}() \cdot \text{Event Option} \) (getOutput)

1. \( \Lambda, \Gamma_2 \vdash \text{getOutput}() \cdot \text{Event Option} \) Rule getOutput

Case \( \Lambda, \Gamma \vdash e_1 : \text{String} \cdot \text{Lambda}, \Gamma \vdash e_2 : \text{TypedVal} \) (invoke)

\[
\begin{align*}
\Lambda, \Gamma_1 \vdash e_1 : \text{String} & \quad \text{assumption} \\
\Lambda, \Gamma_1 \vdash e_2 : \text{TypedVal} & \quad \text{assumption} \\
\Gamma_1 \subseteq \Gamma_2 & \quad \text{assumption} \\
\Gamma_1 \vdash e_1 : \text{String} & \quad 1, 3, \text{IH} \\
\Gamma_1 \vdash e_2 : \text{TypedVal} & \quad 2, 3, \text{IH} \\
\Gamma_1 \vdash \text{invoke}(e_1, e_2) : \text{TypedVal} & \quad 4, 5, \text{Rule invoke} \\
\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 & \quad \text{assumption} \\
\Lambda, \Gamma_1 \vdash e_2 : \tau_1 & \quad \text{assumption} \\
\Gamma_1 \subseteq \Gamma_2 & \quad \text{assumption} \\
\Gamma_1 \vdash e_1 : \tau_1 & \quad 1, 3, \text{IH} \\
\Gamma_1 \vdash e_2 : \tau_1 & \quad 2, 3, \text{IH} \\
\Gamma_1 \vdash \text{call}(e_1, e_2) : \tau_2 & \quad 4, 5, \text{Rule call} \\
\end{align*}
\]

\[\square\]

**Lemma 6 (Substitution).** \( \Lambda, \Gamma \cup \{x : \tau\} \vdash e' : \tau' \land \Lambda, \Gamma \vdash e : \tau \Rightarrow \Lambda, \Gamma \vdash [e/x]e' : \tau' \)**

**Proof.** By induction on the derivation of \( \Lambda, \Gamma \cup \{x : \tau\} \vdash e' : \tau' \)

Case \( \Lambda, \Gamma \vdash \{\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n\} \cdot \Lambda, \Gamma \vdash e_1 : \tau_1 + \cdots + \ell_n \Rightarrow \tau_n : e_n \vdash \tau' \) (case)

\[
\begin{align*}
\Lambda, \Gamma \vdash \{\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n\} & \quad \text{assumption} \\
\Lambda, \Gamma \vdash \{\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n\} & \quad \text{assumption} \\
\Lambda, \Gamma \vdash \{\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n\} & \quad \text{assumption} \\
\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 & \quad 1, 3, \text{IH} \\
\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 & \quad 2, 3, \text{IH} \\
\Lambda, \Gamma \vdash \text{case } [e/x]e_1 \text{ of } \ell_1 x_1 \Rightarrow [e/x]e_1 | \cdots | \ell_n x_n \Rightarrow [e/x]e_n : \tau_1 & \quad 4, 5, \text{Rule case} \\
\Lambda, \Gamma \vdash [e/x](\text{case } e_1 \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau' & \quad 6, \text{definition of } [e/x](\text{case } e_1 \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n)
\end{align*}
\]
\[
\begin{align*}
\text{Case: } & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \text{String} \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \text{TypedVal} \\
& \quad \text{(invoke)} \\
& \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash \text{invoke}(e_1, e_2) : \text{TypedVal Option} \\
1. & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \text{String} \quad \text{assumption} \\
2. & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \text{TypedVal} \quad \text{assumption} \\
3. & \quad \Lambda, \Gamma \vdash e : \tau \quad \text{assumption} \\
4. & \quad \Lambda, \Gamma \vdash [e/x]e_1 : \text{String} \quad 1, 3, \text{IH} \\
5. & \quad \Lambda, \Gamma \vdash [e/x]e_2 : \text{TypedVal} \quad 2, 3, \text{IH} \\
6. & \quad \Lambda, \Gamma \vdash \text{invoke}(e_1, [e/x]e_2) : \text{TypedVal Option} \quad 4, 5, \text{Rule invoke} \\
7. & \quad \Lambda, \Gamma \vdash [e/x](\text{invoke}(e_1, e_2)) : \text{TypedVal Option} \quad 6, \text{definition of } [e/x]\text{invoke}(e_1, e_2)
\end{align*}
\]

\[
\begin{align*}
\text{Case: } & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \text{(makeTypedVal)} \\
1. & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \text{assumption} \\
2. & \quad \Lambda, \Gamma \vdash e : \tau \quad \text{assumption} \\
3. & \quad \Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 \quad 1, 2, \text{IH} \\
4. & \quad \Lambda, \Gamma \vdash \text{makeTypedVal}(\tau_1, [e/x]e_1) : \text{TypedVal} \quad 3, \text{Rule makeTypedVal} \\
5. & \quad \Lambda, \Gamma \vdash [e/x]\text{makeTypedVal}(\tau_1, e_1) : \text{TypedVal} \quad 4, \text{definition of } [e/x]\text{makeTypedVal}(\tau_1, e_1)
\end{align*}
\]

\[
\begin{align*}
\text{Case: } & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \text{TypedVal} \quad \text{(tryCast)} \\
1. & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \text{assumption} \\
2. & \quad \Lambda, \Gamma \vdash e : \tau \quad \text{assumption} \\
3. & \quad \Lambda, \Gamma \vdash [e/x]e_1 : \text{TypedVal} \quad 1, 2, \text{IH} \\
4. & \quad \Lambda, \Gamma \vdash \text{tryCast}(\tau_1, [e/x]e_1) : \tau_1 \quad 3, \text{Rule tryCast} \\
5. & \quad \Lambda, \Gamma \vdash [e/x]\text{tryCast}(\tau_1, e_1) : \tau_1 \quad 4, \text{definition of } [e/x]\text{tryCast}(\tau_1, e_1)
\end{align*}
\]

\[
\begin{align*}
\text{Case: } & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \text{...} \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_n : \tau_n \quad \text{(record)} \\
1. & \quad e' = (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad \text{assumption} \\
2. & \quad \tau' = (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad \text{assumption} \\
3. & \quad \Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 \quad \text{...} \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_n : \tau_n \quad \text{assumption} \\
4. & \quad \Lambda, \Gamma \vdash e : \tau \quad \text{assumption} \\
5. & \quad \Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 \quad \text{...} \quad \Lambda, \Gamma \vdash [e/x]e_n : \tau_n \quad 3, 4, \text{IH} \\
6. & \quad \Lambda, \Gamma \vdash \ell_1 = [e/x]e_1, \ldots, \ell_n = [e/x]e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad 5, \text{Rule record} \\
7. & \quad \Lambda, \Gamma \vdash [e/x]e' : \tau' \quad 1, 2, 6, \text{definition of } [e/x](\ell_1 = e_1, \ldots, \ell_n = e_n)
\end{align*}
\]

\[
\begin{align*}
\text{Case: } & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : (\text{evt} : \tau_1 \text{ Event} \times \text{polList} \times \text{os} \times \text{vc} : \text{VC}) \quad \text{(monitor)} \\
1. & \quad \tau' = (\text{evt} : \tau_1 \text{ Event} \times \text{polList} \times \text{os} \times \text{vc} : \text{VC}) \quad \text{assumption} \\
2. & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau' \quad 1, \text{assumption} \\
3. & \quad \Lambda, \Gamma \vdash e : \tau \quad \text{assumption} \\
4. & \quad \Lambda, \Gamma \vdash [e/x]e_1 : \tau' \quad 2, 3, \text{IH} \\
5. & \quad \Lambda, \Gamma \vdash \text{monitor}(\tau_1, [e/x]e_1) : \tau_1 \quad \text{Event} \quad 4, \text{Rule monitor} \\
6. & \quad \Lambda, \Gamma \vdash [e/x]\text{monitor}(\tau_1, e_1) : \tau_1 \quad \text{Event} \quad 5, \text{definition of } [e/x]\text{monitor}(\tau_1, e_1)
\end{align*}
\]

\[
\begin{align*}
\text{Case: } & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash b : \text{Bool} \quad \text{(boolVal)} \\
1. & \quad \tau' = \text{Bool} \quad \text{assumption} \\
2. & \quad [e/x]b = b \quad \text{definition of } [e/x]b \\
3. & \quad \Lambda, \Gamma \vdash [e/x]b : \tau' \quad 1, 2, \text{Rule boolVal}
\end{align*}
\]

\[
\begin{align*}
\text{Case: } & \quad \Lambda' \cup \{\ell : \tau''\}, \Gamma \cup \{x : \tau\} \vdash e : \tau'' \quad \text{(location)} \\
1. & \quad \tau' = \tau'' \quad \text{Ref} \quad \text{assumption} \\
2. & \quad [e/x]\ell = \ell \quad \text{definition of } [e/x]\ell \\
3. & \quad \Lambda' \cup \{\ell : \tau\}, \Gamma \vdash [e/x]\ell : \tau' \quad 1, 2, \text{Rule location}
\end{align*}
\]
\[
\begin{align*}
\text{Case} & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \text{Bool} \\
\text{Case} & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau' \\
\text{Case} & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \text{Int} \\
\text{Case} & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \text{Int} \\
\end{align*}
\]
\[ \begin{aligned}
\Lambda, \Gamma \cup \{ x : \tau \} &\vdash \text{unit} : \text{Unit} \quad (\text{unitVal}) \\
1. \tau' = \text{Unit} &\quad \text{assumption} \\
2. [e/x]\text{unit} = \text{unit} &\quad \text{definition of } [e/x]\text{unit} \\
3. \Lambda, \Gamma \vdash [e/x]\text{unit} : \tau' &\quad 1, 2, \text{Rule unitVal} \\
\end{aligned} \]

\[ \begin{aligned}
\Lambda, \Gamma \cup \{ x : \tau \} &\vdash ([] : \tau_{\text{List}}) : \tau_{\text{List}} \quad (\text{listEmptyVal}) \\
1. \tau' = \tau_{\text{List}} &\quad \text{assumption} \\
2. [e/x][()] : \tau_{\text{List}} = ([] : \tau_{\text{List}}) &\quad \text{definition of } [e/x][()] : \tau_{\text{List}} \\
3. \Lambda, \Gamma \vdash [e/x][()] : \tau_{\text{List}} : \tau' &\quad 1, 2, \text{Rule listEmptyVal} \\
\end{aligned} \]

\[ \begin{aligned}
\Lambda, \Gamma \cup \{ x : \tau \} &\vdash e_1 : \tau_{\text{List}} \quad (\text{head}) \\
1. \Lambda, \Gamma \cup \{ x : \tau \} &\vdash \text{head}(e_1) : \tau_1 \text{ Option} \\
2. \Lambda, \Gamma \vdash e : \tau &\quad \text{assumption} \\
3. \Lambda, \Gamma \vdash [e/x]e_1 : \tau_{\text{List}} &\quad 1, 2, \text{IH} \\
4. \Lambda, \Gamma \vdash \text{head}([e/x]e_1) : \tau_1 \text{ Option} &\quad 3, \text{Rule head} \\
5. \Lambda, \Gamma \vdash [e/x](\text{head}(e_1)) : \tau_1 \text{ Option} &\quad 4, \text{definition of } [e/x](\text{head}(e_1)) \\
\end{aligned} \]

\[ \begin{aligned}
\Lambda, \Gamma \cup \{ x : \tau \} &\vdash \text{tail}(e_1) : \tau_{\text{List}} \quad (\text{tail}) \\
1. \Lambda, \Gamma \cup \{ x : \tau \} &\vdash e_1 : \tau_{\text{List}} \quad \text{assumption} \\
2. \Lambda, \Gamma \vdash e : \tau &\quad \text{assumption} \\
3. \Lambda, \Gamma \vdash [e/x]e_1 : \tau_{\text{List}} &\quad 1, 2, \text{IH} \\
4. \Lambda, \Gamma \vdash \text{tail}([e/x]e_1) : \tau_{\text{List}} &\quad 3, \text{Rule tail} \\
5. \Lambda, \Gamma \vdash [e/x](\text{tail}(e_1)) : \tau_{\text{List}} &\quad 4, \text{definition of } [e/x](\text{tail}(e_1)) \\
\end{aligned} \]

\[ \begin{aligned}
\Lambda, \Gamma \cup \{ x : \tau \} &\vdash \text{empty}(e') : \text{bool} \quad (\text{empty}) \\
1. \Lambda, \Gamma \cup \{ x : \tau \} &\vdash e' : \tau_{\text{List}} \quad \text{assumption} \\
2. \Lambda, \Gamma \vdash e : \tau &\quad \text{assumption} \\
3. \Lambda, \Gamma \vdash [e/x]\text{empty}(e') = \text{empty}([e/x]e') &\quad \text{definition of } [e/x]\text{empty}(e') \\
4. \Lambda, \Gamma \vdash [e/x]\text{empty}(e') : \tau'_{\text{List}} &\quad 1, 2, \text{IH} \\
5. \Lambda, \Gamma \vdash [e/x]\text{empty}(e') : \text{bool} &\quad 3, 4, \text{Rule empty} \\
\end{aligned} \]

\[ \begin{aligned}
\Lambda, \Gamma \cup \{ x : \tau \} &\vdash e_1 : \tau_1 \quad (\text{listCons}) \\
1. \Lambda, \Gamma \cup \{ x : \tau \} &\vdash e_1 : \tau_2 &\quad \text{assumption} \\
2. \Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 &\quad \text{assumption} \\
3. \Lambda, \Gamma \vdash e : \tau &\quad \text{assumption} \\
4. \Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 &\quad 1, 3, \text{IH} \\
5. \Lambda, \Gamma \vdash [e/x]e_1 : \tau_2 &\quad 2, 3, \text{IH} \\
6. \Lambda, \Gamma \vdash [e/x][e_1 : [e/x]e_2] : \tau_{\text{List}} &\quad 4, 5, \text{Rule listCons} \\
7. \Lambda, \Gamma \vdash [e/x][e_1 : e_2] : \tau_{\text{List}} &\quad 6, \text{definition of } [e/x][e_1 : e_2] \\
\end{aligned} \]
Case $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e'' : \text{Event}$ (setOutput)
1. $\Lambda, \Gamma \cup \{ x : \tau \} \vdash \text{setOutput}(e'') : \text{Bool}$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \text{Event}$ 1, 2, IH
4. $\Lambda, \Gamma \vdash \text{setOutput}([e/x]e'') : \text{Bool}$ 3, Rule setOutput
5. $\Lambda, \Gamma \vdash [e/x](\text{setOutput}(e'')) : \text{Bool}$ 4, definition of $[e/x](\text{setOutput}(e''))$

Case $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e' : \text{Event}$ (let)
1. $\Lambda, \Gamma \cup \{ x : \tau \} \vdash [e/x]e' : \text{Event}$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e' : \text{Event}$ 1, 2, IH
4. $\Lambda, \Gamma \vdash \text{setOutput}([e/x]e'') : \text{Bool}$ 3, Rule setOutput
5. $\Lambda, \Gamma \vdash [e/x](\text{setOutput}(e'')) : \text{Bool}$ 4, definition of $[e/x](\text{setOutput}(e''))$

Case $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e' : \text{Obligation}$ (makeCFG)
1. $\Lambda, \Gamma \cup \{ x : \tau \} \vdash \text{makeCFG}(e'') : \text{CFG}$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \text{Obligation}$ 1, 2, IH
4. $\Lambda, \Gamma \vdash \text{makeCFG}([e/x]e'') : \text{CFG}$ 5, Rule makeCFG
5. $\Lambda, \Gamma \vdash [e/x]e'' : \text{Obligation}$ 3, 4, IH
6. $\Lambda, \Gamma \vdash \text{makeCFG}([e/x]e'') : \text{CFG}$ 5, Rule makeCFG
7. $\Lambda, \Gamma \vdash [e/x]e' : \tau'$ 1, 2, 6, definition of $[e/x](\text{makeCFG}(e''))$

Case $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e' : \text{Obligation}$ (while)
1. $\Lambda, \Gamma \vdash e : \text{Bool}$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash \text{while}(e_1) \{ e_2 \} : \text{Bool}$ assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \text{Bool}$ 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau''$ 2, 3, IH
6. $\Lambda, \Gamma \vdash \text{while}([e/x]e_1) \{ [e/x]e_2 \} : \text{Bool}$ 4, 5, Rule while
7. $\Lambda, \Gamma \vdash [e/x](\text{while}(e_1) \{ e_2 \}) : \text{Bool}$ 6, definition of $[e/x](\text{while}(e_1) \{ e_2 \})$

Case $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e' : \text{Event}$ (endLabel)
1. $\Lambda, \Gamma \vdash e : \tau$ assumption
2. $\Lambda, \Gamma \vdash e : \text{Event}$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e' : \tau'$ 1, 2, IH
4. $\Lambda, \Gamma \vdash \text{setOutput}([e/x]e'') : \text{Bool}$ 3, Rule setOutput
5. $\Lambda, \Gamma \vdash [e/x](\text{setOutput}(e'')) : \text{Bool}$ 4, definition of $[e/x](\text{setOutput}(e''))$

Case $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e' : \text{Event}$ (projection)
1. $\Lambda, \Gamma \cup \{ x : \tau \} \vdash e'' \cdot (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \in \{ 1, \ldots, n \}$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \ell_1 : \tau_1$ 1, 3, IH
4. $\Lambda, \Gamma \vdash ([e/x]e''), \ell_i : \tau_i$ 2, 4, Rule projection
5. $\Lambda, \Gamma \vdash [e/x](e'', \ell_i) : \tau_i$ 5, definition of $[e/x](e'', \ell_i)$
Lemma 7 (Typing Rule Inversion). All of the typing rules are invertible.

Proof. By case analysis of rules deriving \( \Delta, \Gamma \vdash e : \tau \)

Case \( \Delta, \Gamma \vdash n : \text{Int} \) (intVal)
1. \( \Delta, \Gamma \vdash n : \tau \) assumption
2. \( \Delta, \Gamma \vdash n : \tau \) is only derivable by Rule intVal inspection of typing rules
3. \( \tau = \text{Int} \) 1, 2, rule intVal

Case \( \Delta, \Gamma \vdash b : \text{Bool} \) (boolVal)
1. \( \Delta, \Gamma \vdash b : \tau \) assumption
2. \( \Delta, \Gamma \vdash b : \tau \) is only derivable by Rule boolVal inspection of typing rules
3. \( \tau = \text{Bool} \) 1, 2, rule boolVal

Case \( \Delta, \Gamma \vdash s : \text{String} \) (stringVal)
1. \( \Delta, \Gamma \vdash s : \tau \) assumption
2. \( \Delta, \Gamma \vdash s : \tau \) is only derivable by Rule stringVal inspection of typing rules
3. \( \tau = \text{String} \) 1, 2, rule stringVal

Case \( \Delta, \Gamma \vdash \text{unit} : \text{Unit} \) (unitVal)
1. \( \Delta, \Gamma \vdash \text{unit} : \tau \) assumption
2. \( \Delta, \Gamma \vdash \text{unit} : \tau \) is only derivable by Rule unitVal inspection of typing rules
3. \( \tau = \text{Unit} \) 1, 2, rule unitVal

Case \( \Delta, \Gamma \vdash \text{fun} x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e' : \tau \) (fun)
1. \( \Delta, \Gamma \vdash \text{fun} x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e' : \tau \) assumption
2. \( \Delta, \Gamma \vdash \text{fun} x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e' : \tau \) is only derivable by rule fun inspection of typing rules
3. \( \tau = \tau_1 \rightarrow \tau_2, \Delta, \Gamma \vdash \text{fun} x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e' : \tau_2 \) 1, 2, Rule fun

Case \( \Delta' \cup [\ell : \tau'], \Delta \vdash \ell : \tau' \) (location)
1. \( \Delta' \cup [\ell : \tau'], \Delta \vdash \ell : \tau \) assumption
2. \( 1 \) is only derivable by Rule location inspection of typing rules
3. \( \tau = \tau' \) 1, 2, Rule location

Case \( \Delta, \Gamma \vdash x : \tau \) (var)
1. \( \Delta, \Gamma \vdash x : \tau \) assumption
2. \( 1 \) is only derivable by Rule var inspection of typing rules
3. \( \tau = \tau' \) 1, 2, Rule var

\( \square \)
Case\[\Lambda, \Gamma \vdash e_1 : \text{Bool}, \Lambda, \Gamma \vdash e_2 : \text{Bool}\]

1. \(\Lambda, \Gamma \vdash e_1 \land e_2 : \tau\) assumption
2. \(\Lambda, \Gamma \vdash e_1 \land e_2 : \tau\) is only derivable by rule \(\land\) inspection of typing rules
3. \(\tau = \text{Bool}, \Lambda, \Gamma \vdash e_1 : \text{Bool}, \Lambda, \Gamma \vdash e_2 : \text{Bool}\) 1, 2, Rule \(\land\)

Case\[\Lambda, \Gamma \vdash e_1 : \text{Bool}, \Lambda, \Gamma \vdash e_2 : \text{Bool}\]

1. \(\Lambda, \Gamma \vdash e_1 \lor e_2 : \tau\) assumption
2. \(\Lambda, \Gamma \vdash e_1 \lor e_2 : \tau\) is only derivable by Rule \(\lor\) inspection of typing rules
3. \(\tau = \text{Bool}, \Lambda, \Gamma \vdash e_1 : \text{Bool}, \Lambda, \Gamma \vdash e_2 : \text{Bool}\) 1, 2, Rule \(\lor\)

Case\[\Lambda, \Gamma \vdash e : \text{Bool}\]

1. \(\Lambda, \Gamma \vdash \neg e : \tau\) assumption
2. \(\Lambda, \Gamma \vdash \neg e : \tau\) is only derivable by rule \(\neg\) inspection of typing rules
3. \(\tau = \text{Bool}, \Lambda, \Gamma \vdash e : \text{Bool}\) 1, 2, Rule \(\neg\)

Case\[\Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \vdash e_2 : \tau_2\]

1. \(\Lambda, \Gamma \vdash e_1 = e_2 : \tau\) assumption
2. \(\Lambda, \Gamma \vdash e_1 = e_2 : \tau\) is only derivable by rule \(=\) inspection of typing rules
3. \(\tau = \text{Bool}, \Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \vdash e_2 : \tau_1\) 1, 2, Rule \(=\)

Case\[\Lambda, \Gamma \vdash e_1 : \text{Int}, \Lambda, \Gamma \vdash e_2 : \text{Int}\]

1. \(\Lambda, \Gamma \vdash e_1 + e_2 : \tau\) assumption
2. \(\Lambda, \Gamma \vdash e_1 + e_2 : \tau\) is only derivable by rule \(+\) inspection of typing rules
3. \(\tau = \text{Int}, \Lambda, \Gamma \vdash e_1 : \text{Int}, \Lambda, \Gamma \vdash e_2 : \text{Int}\) 1, 2, Rule \(+\)

Case\[\Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \vdash e_2 : \tau_2\]

1. \(\Lambda, \Gamma \vdash e_1 : e_2 : \tau\) assumption
2. \(\Lambda, \Gamma \vdash e_1 : e_2 : \tau\) is only derivable by rule \(\tau\) inspection of typing rules
3. \(\tau = \tau_2, \Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \vdash e_2 : \tau_2\) 1, 2, Rule \(\tau\)

Case\[\Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_1\]

1. \(\Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau\) assumption
2. \(\Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau\) is only derivable by rule \text{if} inspection of typing rules
3. \(\tau = \tau_1, \Lambda, \Gamma \vdash e_1 : \text{Bool}, \Lambda, \Gamma \vdash e_2 : \tau_1, \Lambda, \Gamma \vdash e_3 : \tau_1\) 1, 2, Rule \text{if}

Case\[\Lambda, \Gamma \vdash \text{while}(e_1) \{e_2\} : \text{Bool}\]

1. \(\Lambda, \Gamma \vdash \text{while}(e_1) \{e_2\} : \text{Bool}\) assumption
2. \(\Lambda, \Gamma \vdash \text{while}(e_1) \{e_2\} : \text{Bool}\) is only derivable by rule \text{while} inspection of typing rules
3. \(\tau = \text{Bool}, \Lambda, \Gamma \vdash e_1 : \text{Bool}, \Lambda, \Gamma \vdash e_2 : \tau_1\) 1, 2, Rule \text{while}

Case\[\Lambda, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2\]

1. \(\Lambda, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau\) assumption
2. \(\Lambda, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau\) is only derivable by rule \text{let} inspection of typing rules
3. \(\tau = \tau_2, \Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2\) 1, 2, Rule \text{let}

Case\[\Lambda, \Gamma \vdash \text{ref } e : \tau_1\]

1. \(\Lambda, \Gamma \vdash \text{ref } e : \tau\) assumption
2. \(\Lambda, \Gamma \vdash \text{ref } e : \tau\) is only derivable by rule \text{createRef} inspection of typing rules
3. \(\tau = \tau_1 \text{Ref, } \Lambda, \Gamma \vdash e : \tau_1\) 1, 2, Rule \text{createRef}
Case $\Lambda, \Gamma \vdash e : \tau \quad \text{(accessRef)}$
1. $\Lambda, \Gamma \vdash e : \tau$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash e : \tau$ is only derivable by rule accessRef inspection of typing rules
3. $\tau = \tau_1$, $\Lambda, \Gamma \vdash e : \tau_1$ \hspace{2cm} 1, 2, Rule accessRef

Case $\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_1 \quad \text{(assignment)}$
1. $\Lambda, \Gamma \vdash e_1 : \tau_2$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash e_1 : \tau_2$ is only derivable by rule assignment inspection of typing rules
3. $\tau = \text{unit}$, $\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_1$ \hspace{2cm} 1, 2, Rule assignment

Case $\Lambda, \Gamma \vdash ([] : \tau'_\text{List}) : \tau'_\text{List} \quad \text{(listEmptyVal)}$
1. $\Lambda, \Gamma \vdash ([] : \tau'_\text{List}) : \tau$ \hspace{2cm} assumption
2. $\tau$ is only derivable by Rule listEmptyVal inspection of typing rules
3. $\tau = \tau'_\text{List}$ \hspace{2cm} 1, 2, Rule listEmptyVal

Case $\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_\text{List} \quad \text{(listCons)}$
1. $\Lambda, \Gamma \vdash e_1 : \tau_2$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash e_1 : \tau_2$ is only derivable by rule listCons inspection of typing rules
3. $\tau = \tau_1\ell, \Lambda, \Gamma \vdash e_1 : \tau_1\ell, \Lambda, \Gamma \vdash e_2 : \tau_\text{List}$ \hspace{2cm} 1, 2, Rule listCons

Case $\Lambda, \Gamma \vdash e @ e_2 : \tau_\text{List} \quad \text{(listAppend)}$
1. $\Lambda, \Gamma \vdash e_1 @ e_2 : \tau$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash e_1 @ e_2 : \tau$ is only derivable by rule listAppend inspection of typing rules
3. $\tau = \tau_\text{List}, \Lambda, \Gamma \vdash e_1 : \tau_\text{List}, \Lambda, \Gamma \vdash e_2 : \tau_\text{List}$ \hspace{2cm} 1, 2, Rule listAppend

Case $\Lambda, \Gamma \vdash e : \tau_\text{List} \quad \text{(head)}$
1. $\Lambda, \Gamma \vdash \text{head}(e) : \tau$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash \text{head}(e) : \tau$ is only derivable by rule head inspection of typing rules
3. $\tau = \tau_1\ell, \Lambda, \Gamma \vdash e : \tau_1\ell$ \hspace{2cm} 1, 2, Rule head

Case $\Lambda, \Gamma \vdash \text{tail}(e) : \tau_\text{List} \quad \text{(tail)}$
1. $\Lambda, \Gamma \vdash \text{tail}(e) : \tau$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash \text{tail}(e) : \tau$ is only derivable by rule tail inspection of typing rules
3. $\tau = \tau_\text{List}, \Lambda, \Gamma \vdash e : \tau_\text{List}$ \hspace{2cm} 1, 2, Rule tail

Case $\Lambda, \Gamma \vdash \text{empty}(e) : \text{Bool} \quad \text{(empty)}$
1. $\Lambda, \Gamma \vdash \text{empty}(e) : \tau$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash \text{empty}(e) : \tau$ is only derivable by rule empty inspection of typing rules
3. $\tau = \text{Bool}$, $\Lambda, \Gamma \vdash e : \tau_\text{List}$ \hspace{2cm} 1, 2, Rule empty

Case $\Lambda, \Gamma \vdash e_1 : \tau_1, \ldots, \Lambda, \Gamma \vdash e_n : \tau_n \quad \text{(record)}$
1. $\Lambda, \Gamma \vdash (e_1, \ldots, e_n) : (e_1 : \tau_1 \times \cdots \times e_n : \tau_n)$ \hspace{2cm} 1, 2, Rule record
2. $\Lambda, \Gamma \vdash (e_1, \ldots, e_n) : \tau$ is only derivable by rule record inspection of typing rules
3. $\tau = (e_1 : \tau_1 \times \cdots \times e_n : \tau_n)$, $\Lambda, \Gamma \vdash v_1 : \tau_1, \ldots, \Lambda, \Gamma \vdash v_n : \tau_n$ \hspace{2cm} 1, 2, Rule record

Case $\Lambda, \Gamma \vdash e : (e_1 : \tau_1 \times \cdots \times e_n : \tau_n) \; i \in [1, \ldots, n] \quad \text{(projection)}$
1. $\Lambda, \Gamma \vdash e_1 : \tau$ \hspace{2cm} assumption
2. $\Lambda, \Gamma \vdash e_1 : \tau$ is only derivable by rule projection inspection of typing rules
3. $\tau = \tau_1$, $\Lambda, \Gamma \vdash e : (e_1 : \tau_1 \times \cdots \times e_n : \tau_n)$ \hspace{2cm} 1, 2, Rule projection
\[
\text{Case } \Lambda, \Gamma \vdash e_i : \tau_i, \ i \in \{1, \ldots, n\} \quad \text{(variant)}
\]

1. \(\Lambda, \Gamma \vdash (\pi_{\ell_i}, e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : \tau\) \quad \text{assumption}

2. \(\Lambda, \Gamma \vdash (\pi_{\ell_i}, e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : \tau\) \quad \text{is only derivable by rule \text{variant} \ \text{inspection of typing rules}

3. \(\tau = (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n), \ \Lambda, \Gamma \vdash e_i : \tau_i, \ i \in \{1, \ldots, n\}\) \quad \text{1, 2, Rule \text{variant}}

\[
\text{Case } \Lambda, \Gamma \vdash e' : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \ \Lambda, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau_x \ldots \Lambda, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau_x \quad \text{\ (case)}
\]

1. \(\Lambda, \Gamma \vdash (\text{case } e' \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau\) \quad \text{assumption}

2. \(\Lambda, \Gamma \vdash (\text{case } e' \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau\) \quad \text{is only derivable by rule case \ \text{inspection of typing rules}

3. \(\tau = \tau_x, \ \Lambda, \Gamma \vdash e' : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n), \ \Lambda, \Gamma \vdash e_i : \tau_i \ldots \Lambda, \Gamma \vdash e_n : \tau_n \quad \text{1, 2, Rule case}

\[
\text{Case } \Lambda, \Gamma \vdash \text{makeTypedVal}(\tau_1, e_1) : \text{TypedVal}
\]

1. \(\Lambda, \Gamma \vdash \text{makeCfg}(\tau_1, e_1) : \text{Cfg}\) \quad \text{assumption}

2. \(\Lambda, \Gamma \vdash \text{makeCfg}(\tau_1, e_1) : \text{Cfg}\) \quad \text{is only derivable by rule \text{makeCfg} \ \text{inspection of typing rules}

3. \(\tau = \text{Cfg}, \ \Lambda, \Gamma \vdash e_1 : \tau_i \ldots \Lambda, \Gamma \vdash e_n : \tau_i \quad \text{1, 2, Rule \text{makeCfg}}

\[
\text{Case } \Lambda, \Gamma \vdash e : \tau_i \quad \text{\ (endLabel)}
\]

1. \(\Lambda, \Gamma \vdash \{e\}_{\text{label}} : \tau_i\) \quad \text{assumption}

2. \(\Lambda, \Gamma \vdash \{e\}_{\text{label}} : \tau\) \quad \text{is only derivable by rule \text{endLabel} \ \text{inspection of typing rules}

3. \(\tau = \tau_i, \ \Lambda, \Gamma \vdash e : \tau_i \quad \text{1, 2, Rule \text{endLabel}}

\[
\text{Case } \Lambda, \Gamma \vdash \text{getRT()} : \text{ResList}
\]

1. \(\Lambda, \Gamma \vdash \text{getRT()} : \tau\) \quad \text{assumption}

2. \(\text{1 is only derivable by Rule \text{getRT} \ \text{inspection of typing rules}

3. \(\tau = \text{ResList} \quad 1, 2, \text{Rule \text{getRT}}

\[
\text{Case } \Lambda, \Gamma \vdash e : \text{Obligation}
\]

1. \(\Lambda, \Gamma \vdash \text{makeCfg}(e_1) : \text{Cfg}\) \quad \text{assumption}

2. \(\Lambda, \Gamma \vdash \text{makeCfg}(e_1) : \tau\) \quad \text{is only derivable by rule \text{makeCfg} \ \text{inspection of typing rules}

3. \(\tau = \text{Cfg}, \ \Lambda, \Gamma \vdash e_1 : \text{Obligation} \quad 1, 2, \text{Rule \text{makeCfg}}

\[
\text{Case } \Lambda, \Gamma \vdash \text{setOutput}(e_1) : \text{unit}
\]

1. \(\Lambda, \Gamma \vdash \text{setOutput}(e_1) : \tau\) \quad \text{assumption}

2. \(\Lambda, \Gamma \vdash \text{setOutput}(e_1) : \tau\) \quad \text{is only derivable by rule \text{setOutput} \ \text{inspection of typing rules}

3. \(\tau = \text{unit}, \ \Lambda, \Gamma \vdash e_1 : \text{Event} \quad 1, 2, \text{Rule \text{setOutput}}

\[
\text{Case } \Lambda, \Gamma \vdash \text{outputNotSet()} : \text{Bool}
\]

1. \(\Lambda, \Gamma \vdash \text{outputNotSet()} : \tau\) \quad \text{assumption}

2. \(\text{1 is only derivable by Rule \text{outputNotSet} \ \text{inspection of typing rules}

3. \(\tau = \text{Bool} \quad 1, 2, \text{Rule \text{outputNotSet}}

\[
\text{Case } \Lambda, \Gamma \vdash \text{outputNotSet()} : \text{Event Option}
\]

1. \(\Lambda, \Gamma \vdash \text{outputNotSet()} : \tau\) \quad \text{assumption}

2. \(\text{1 is only derivable by Rule \text{outputNotSet} \ \text{inspection of typing rules}

3. \(\tau = \text{Event Option} \quad 1, 2, \text{Rule \text{outputNotSet}}

\]
Proof. By induction on the derivation of Lemma 8 (Canonical Forms).

1. \( \Lambda, \Gamma \vdash monitor(\tau', e) : \tau' \) assumption
2. \( \tau \) is only derivable by Rule monitor Inspection of typing rules
3. \( \tau = \tau' \) Event 2, Inversion of Rule monitor
4. \( \Lambda, \Gamma \vdash e : (evt : \tau' \text{ Event} \times pols : \text{PolList} \times os : \text{OS} \times vc : \text{VC}) \) 2, Inversion of Rule monitor

Result is from 3, 4

Case \( \Lambda, \Gamma \vdash e_1 : \text{String} \quad \Lambda, \Gamma \vdash e_2 : \text{TypedVal} \) (invoke)

1. \( \Lambda, \Gamma \vdash invoke(e_1, e_2) : \tau \) assumption
2. \( \tau \) is only derivable by Rule invoke Inspection of typing rules
3. \( \tau = \text{TypedVal Option}, \Lambda, \Gamma \vdash e_1 : \text{String}, \Lambda, \Gamma \vdash e_2 : \text{TypedVal} \) 1, 2, Rule invoke

Case \( \Lambda, \Gamma \vdash call(e_1, e_2) : \tau \) (call)

1. \( \Lambda, \Gamma \vdash \text{call}(e_1, e_2) : \tau \) assumption
2. \( \tau \) is only derivable by rule call Inspection of typing rules
3. \( \tau = \tau_2, \Lambda, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2, \Lambda, \Gamma \vdash e_2 : \tau_1 \) 1, 2, Rule call

\[\square\]

Lemma 8 (Canonical Forms). If \( \Lambda, \bullet \vdash v : \tau \) then

1. \( \tau = \text{Bool} \Rightarrow v = b \)
2. \( \tau = \text{String} \Rightarrow v = s \)
3. \( \tau = \text{Int} \Rightarrow v = n \)
4. \( \tau = \text{Unit} \Rightarrow v = \text{unit} \)
5. \( \tau = \text{Ref} \Rightarrow v = \ell \)
6. \( \tau = \text{TypedVal} \Rightarrow \exists v_1 : v = \text{makeTypedVal}(\tau, v_1) \)
7. \( \tau = \text{List} \Rightarrow (\exists v_1, v_2 : v = v_1 :: v_2) \lor v = [] : \text{List} \)
8. \( \tau = (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \Rightarrow \exists v_{\ell_1}, \ldots, v_{\ell_n} : v = (v_{\ell_1} \times \cdots \times v_{\ell_n} = v_n) \)
9. \( \tau = e_1 \rightarrow e_2 \Rightarrow \exists x_1, x_2, e : v = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e) \)
10. \( \tau = (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \Rightarrow \exists i, v_1 : i \in \{1, \ldots, n\} \land v = in_{\ell_i} v_1 : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \)

Proof. By induction on the derivation of \( \Lambda, \bullet \vdash v : \tau \)

Case \( \Lambda, \bullet \vdash n : \text{Int} \) (intVal)

1. \( v = n \) assumption

Case \( \Lambda, \bullet \vdash b : \text{Bool} \) (boolVal)

1. \( v = b \) assumption

Case \( \Lambda, \bullet \vdash s : \text{String} \) (stringVal)

1. \( v = s \) assumption

Case \( \Lambda, \bullet \vdash \text{unit} : \text{Unit} \) (unitVal)

1. \( v = \text{unit} \) assumption

Case \( \Lambda, \bullet \vdash \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e : \tau_2 \) (fun)

1. \( v = \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e \) assumption
\[
\begin{array}{l}
\text{Case } (\Lambda') \cup \{x : \tau\}, \Gamma \vdash \ell : \tau \text{ Ref} \\
1. \quad v = \ell \quad \text{assumption} \\
\hline
\text{Case } \Lambda, \bullet \cup \{x : \tau\} \vdash x : \tau \text{ (var)} \\
1. \quad x \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \text{Bool} \quad \Lambda, \bullet \vdash e_2 : \text{Bool} \text{ (con)} \\
1. \quad e_1 \land e_2 \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \text{Bool} \quad \Lambda, \bullet \vdash e_2 : \text{Bool} \text{ (dis)} \\
1. \quad e_1 \lor e_2 \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e : \text{Bool} \quad \Lambda, \bullet \vdash \neg e : \text{Bool} \text{ (negation)} \\
1. \quad \neg e \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \tau \quad \Lambda, \bullet \vdash e_2 : \tau \quad \tau \in \{\text{Int}, \text{Bool}, \text{String}\} \text{ (equality)} \\
1. \quad e_1 \equiv e_2 \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \text{Int} \quad \Lambda, \bullet \vdash e_2 : \text{Int} \text{ (add)} \\
1. \quad e_1 + e_2 \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \tau_1 \quad \Lambda, \bullet \vdash e_2 : \tau_2 \text{ (sequence)} \\
1. \quad e_1; e_2 \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \text{Bool} \quad \Lambda, \bullet \vdash e_2 : \tau \quad \Lambda, \bullet \vdash e_3 : \tau \text{ (if)} \\
1. \quad if \ e_1 \ then \ e_2 \ else \ e_3 \ is not a value so the lemma holds vacuously in this case \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \text{Bool} \quad \Lambda, \bullet \vdash \text{while}(e_1) \{e_2\} : \text{Bool} \text{ (while)} \\
1. \quad \text{while}(e_1) \{e_2\} \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e_1 : \tau_1 \quad \Lambda, \bullet \cup \{x : \tau_1\} \vdash e_2 : \tau_2 \text{ (let)} \\
1. \quad let \ x = e_1 \ in \ e_2 \ end : \tau_2 \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash \text{ref } e : \tau \text{ Ref} \text{ (createRef)} \\
1. \quad \text{ref } e \text{ is not a value so the lemma holds vacuously in this case} \\
\hline
\text{Case } \Lambda, \bullet \vdash e : \tau \text{ Ref} \text{ (accessRef)} \\
1. \quad !e \text{ is not a value so the lemma holds vacuously in this case} \\
\end{array}
\]
\[
\begin{array}{c}
\text{Case} \\
\lambda \cdot e_2 \vdash e_1 : \tau \rightarrow e_2 : \tau \\
\lambda \cdot e_1 := e_2 : \text{unit} \quad \text{(assignment)}
\end{array}
\]

1. \(e_1 := e_2\) is not a value so the lemma holds vacuously in this case

\[
\begin{array}{c}
\text{Case} \\
\lambda \cdot v = \text{listEmptyVal} \quad \text{(listEmptyVal)}
\end{array}
\]

1. \(v = \text{listEmptyVal}\) assumption

\[
\begin{array}{c}
\text{Case} \\
\lambda \cdot e_1 : \tau, \lambda \cdot e_2 : \text{listList} \quad \text{(listCons)}
\end{array}
\]

1. \(e_1 = v_1 \text{ and } e_2 = v_2 \Rightarrow e_1 :: e_2 = v_1 :: v_2\) assumption
2. \(e_1 \neq v_1 \text{ or } e_2 \neq v_2 \Rightarrow e_1 :: e_2\) is not a value the lemma holds vacuously in this case

Result from 1, 2

\[
\begin{array}{c}
\text{Case} \\
\lambda \cdot e_1 : \text{listList}, \lambda \cdot e_2 : \text{listList} \quad \text{(listAppend)}
\end{array}
\]

1. \(e_1 @ e_2\) is not a value so the lemma holds vacuously in this case

\[
\begin{array}{c}
\text{Case} \\
\lambda \cdot e_1 : \tau \quad \text{(head)}
\end{array}
\]

1. \(\text{head}(e_1)\) is not a value so the lemma holds vacuously in this case

\[
\begin{array}{c}
\text{Case} \\
\lambda \cdot e_1 : \text{listList} \quad \text{(tail)}
\end{array}
\]

1. \(\text{tail}(e_1)\) is not a value so the lemma holds vacuously in this case

\[
\begin{array}{c}
\lambda, \Gamma \vdash e : \text{listList} \quad \text{(empty)}
\end{array}
\]

1. \(\text{empty}(e_2)\) is not a value so the lemma holds vacuously in this case

\[
\begin{array}{c}
\lambda \cdot e_1 : \tau_1, \ldots, \lambda \cdot e_n : \tau_n \quad \text{(record)}
\end{array}
\]

1. \(e_1 = v_1, \ldots, e_n = v_n \Rightarrow \ell_1 = e_1, \ldots, \ell_n = e_n\) assumption
2. \(e_1 \neq v_1 \lor \ldots \lor e_n \neq v_n \Rightarrow \ell_1 = e_1, \ldots, \ell_n = e_n\) is not a value the lemma holds vacuously in this case

Result from 1, 2

\[
\begin{array}{c}
\lambda \cdot e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad \text{(projection)}
\end{array}
\]

1. \(e . \ell_i\) is not a value so the lemma holds vacuously in this case

\[
\begin{array}{c}
\lambda \cdot e_1 : \tau_i \quad i \in [1, \ldots, n] \quad \text{(variant)}
\end{array}
\]

1. \(\tau = (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)\) assumption
2. \(u = \text{in}_{\ell_i} e_1 : \tau\) assumption, 1
3. \(i \in [1, \ldots, n]\) assumption
4. \(e_1 = v_1 \Rightarrow u = \text{in}_{\ell_i} v_1 : \tau\) assumption, 2
5. \(e_1 \neq v_1 \Rightarrow u\) is not a value the lemma holds vacuously in this case

Result from 4, 5

\[
\begin{array}{c}
\lambda \cdot e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \lambda, [x_1 : \tau_1] : e_1 : \tau, \ldots, \lambda, [x_n : \tau_n] : e_n : \tau \quad \text{(case)}
\end{array}
\]

1. \((\text{case } e \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau\) is not a value so the lemma holds vacuously in this case
We will instead prove the equivalent statement

\[ \Lambda \vdash (C, e) : \tau \Rightarrow e \text{ value } \lor \exists C', e' : (C, e) \rightarrow (C', e') \]

We will instead prove the equivalent statement \( \Lambda \vdash C \text{ ok} \land \Lambda \vdash e : \tau \Rightarrow e \text{ value } \lor \exists C', e' : (C, e) \rightarrow (C', e') \). It can be shown that these two statements are equivalent by inversion of rule TConfig.
Proof. By induction on the derivation of \( \Lambda, \bullet \vdash e : \tau \)

Case \( \Lambda, \bullet \vdash n : Int \) (intVal)
1. \( n \) \( \text{value} \) definition of values

Case \( \Lambda, \bullet \vdash b : \text{Bool} \) (boolVal)
1. \( b \) \( \text{value} \) definition of values

Case \( \Lambda, \bullet \vdash s : \text{String} \) (stringVal)
1. \( s \) \( \text{value} \) definition of values

Case \( \Lambda, \bullet \vdash \text{unit} : \text{Unit} \) (unitVal)
1. \( \text{unit} \) \( \text{value} \) definition of values

Case \( \Lambda, \bullet \vdash e_1 : \tau_2 \)
\( \Lambda, \bullet \vdash \text{fun} \ x_1(x_2 : \tau_1) : \tau_2 = e_2 : \tau_1 \to \tau_2 \) (fun)
1. \( \text{(fun} x_1(x_2 : \tau_1) : \tau_2 = e_2) \) \( \text{value} \) definition of values

Case \( \Lambda \cup \{ \ell : \tau \} \vdash \ell : \tau \) Ref (location)
1. \( \ell \) \( \text{value} \) definition of values

Case \( \Lambda, \bullet \vdash x : \tau \) (var)
1. \( \Lambda, \bullet \vdash x : \tau \) assumption
2. \( \Lambda, \bullet \vdash x : \tau \) is not derivable Inspection of typing rules
3. This case holds vacuously 1, 2

Case \( \Lambda, \bullet \vdash e_1 : \text{Bool} \) \( \Lambda, \bullet \vdash e_2 : \text{Bool} \) (con)
1. \( \Lambda, \bullet \vdash \text{C ok} \) assumption
2. \( \Lambda, \bullet \vdash e_1 : \text{Bool} \) assumption
3. \( \Lambda, \bullet \vdash e_2 : \text{Bool} \) assumption
4. \( e_1 \equiv v_1 \) or \( (C, e_1) \to (C', e'_1) \)
5. \( e_2 \equiv v_2 \) or \( (C, e_2) \to (C', e'_2) \)
6. \( e_1 = v_1 \Rightarrow e_1 \in \{ \text{true, false} \} \)
7. \( (C, e_1) \to (C', e'_1) \Rightarrow (C, e_1 \land e_2) \to (C', e'_1 \land e_2) \)
8. \( (e_1 = v_1) \) and \( (C, e_2) \to (C', e'_2) \Rightarrow (C, e_1 \land e_2) \to (C', v_1 \land e'_2) \)
9. \( (e_1 = \text{true and } e_2 = v_2) \Rightarrow (C, e_1 \land e_2) \to (C, v_2) \)
10. \( (e_1 = \text{false and } e_2 = v_2) \Rightarrow (C, e_1 \land e_2) \to (C, \text{false}) \)
11. \( \exists C', e' : (C, e_1 \land e_2) \to (C', e') \)

Case \( \Lambda, \bullet \vdash e_1 : \text{Bool} \) \( \Lambda, \bullet \vdash e_2 : \text{Bool} \) (dis)
1. This case is analogous to case con.

Case \( \Lambda, \bullet \vdash e_1 : \text{Bool} \)
\( \Lambda, \bullet \vdash \neg e_1 : \text{Bool} \) (negation)
1. \( \Lambda, \bullet \vdash \text{C ok} \) assumption
2. \( \Lambda, \bullet \vdash e_1 : \text{Bool} \) assumption
3. \( e_1 = v_1 \lor (C, e_1) \to (C', e'_1) \)
4. \( e_1 = v_1 \Rightarrow e_1 \in \{ \text{true, false} \} \)
5. \( e_1 = \text{true} \Rightarrow (C, \neg e_1) \to (C, \text{false}) \)
6. \( e_1 = \text{false} \Rightarrow (C, \neg e_1) \to (C, \text{true}) \)
7. \( (C, e_1) \to (C', e'_1) \Rightarrow (C, \neg e_1) \to (C', \neg e'_1) \)
8. \( \exists C', e' : (C, \neg e_1) \to (C', e') \)

5-7
Case

\[ \Lambda, \bullet \vdash e_1 : \tau \quad \Lambda, \bullet \vdash e_2 : \tau \quad \tau \in \{\text{Int}, \text{Bool}, \text{String}\} \]

\[
\Lambda, \bullet \vdash e_1 == e_2 : \text{Bool}
\]

1. \( \Lambda \vdash C \text{ ok} \)
   
   assumption
2. \( \Lambda, \bullet \vdash e_1 : \tau \)
   
   assumption
3. \( \Lambda, \bullet \vdash e_2 : \tau \)
   
   assumption
4. \( \tau \in \{\text{Int}, \text{Bool}, \text{String}\} \)
   
   assumption
5. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
   
   1, 2, IH
6. \( e_2 = v_2 \) or \( (C, e_2) \rightarrow (C', e'_2) \)
   
   1, 3, IH
7. \( e_1 \rightarrow e'_1 \Rightarrow (C, e_1 == e_2) \rightarrow (C', e'_1 == e_2) \)
   
   Rule eqE1
8. \( e_1 = v_1 \) and \( e_2 \rightarrow e'_2 \Rightarrow (C, e_1 == e_2) \rightarrow (C', v_1 == e'_2) \)
   
   Rule eqE2
9. \( e_1 = v_1 \) and \( e_2 = v_2 \Rightarrow \)
   
   a. \( \tau = \text{Int} \Rightarrow \)
      
      Lemma Canonical Forms, 2, 9, 9a
   
   b. \( \tau = \text{Bool} \Rightarrow \)
      
      Lemma Canonical Forms, 2, 9, 9b
   
   c. \( \tau = \text{String} \Rightarrow \)
      
      Lemma Canonical Forms, 2, 9, 9c
10. \( \Lambda, \bullet \vdash e_1 : \text{Int} \quad \Lambda, \bullet \vdash e_2 : \text{Int} \quad (\text{add}) \)

\[
\Lambda, \bullet \vdash e_1 + e_2 : \text{Int}
\]

1. \( \Lambda \vdash C \text{ ok} \)
   
   assumption
2. \( \Lambda, \bullet \vdash e_1 : \text{Int} \)
   
   assumption
3. \( \Lambda, \bullet \vdash e_2 : \text{Int} \)
   
   assumption
4. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
   
   1, 2, IH
5. \( e_2 = v_2 \) or \( (C, e_2) \rightarrow (C', e'_2) \)
   
   1, 3, IH
6. \( e_1 = e_1 \Rightarrow e_1 = n_1 \)
   
   2, Lemma Canonical Forms
7. \( e_2 = e_2 \Rightarrow e_2 = n_2 \)
   
   3, Lemma Canonical Forms
8. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 + e_2) \rightarrow (C', e'_1 + e_2) \)
   
   Rule addE1
9. \( (e_1 = n_1 \text{ and } (C, e_2) \rightarrow (C', e'_2)) \Rightarrow (C, e_1 + e_2) \rightarrow (C', n_1 + e'_2) \)
   
   Rule addE2
10. \( (e_1 = n_1 \text{ and } e_2 = n_2) \Rightarrow (C, e_1 + e_2) \rightarrow (C, n_1 + n_2) \)
    
    6, 7, Rule addValue
11. \( e = v \text{ or } (C', e') \rightarrow (C', e') \)
    
    8-10
12. \( \Lambda, \bullet \vdash e_1 : t_1 \quad \Lambda, \bullet \vdash e_2 : t_2 \quad (\text{sequence}) \)

\[
\Lambda, \bullet \vdash e_1; e_2 : t_2
\]

1. \( \Lambda \vdash C \text{ ok} \)
   
   assumption
2. \( \Lambda, \bullet \vdash e_1 : t_1 \)
   
   assumption
3. \( \Lambda, \bullet \vdash e_2 : t_2 \)
   
   assumption
4. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
   
   1, 2, IH
5. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, (e_1; e_2)) \rightarrow (C', (e'_1; e_2)) \)
   
   Rule sequenceE1
6. \( e_1 = v_1 \Rightarrow (C, (e_1; e_2)) \rightarrow (C, e_2) \)
   
   Rule sequenceE2
7. \( \exists C', e' : (C, e) \rightarrow (C', e') \)
   
   3-5
Case \( \Lambda \cdot \vdash e_1 : \text{Bool} \) \( \Lambda \cdot \vdash e_2 : \tau \) \( \Lambda \cdot \vdash e_3 : \tau \) (d)

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda \cdot \vdash e_1 : \text{Bool} \)
3. \( \Lambda \cdot \vdash e_2 : \tau \)
4. \( \Lambda \cdot \vdash e_3 : \tau \)
5. \( e_1 = v_1 \text{ or } (C, e_1) \rightarrow (C', e'_1) \)
6. \( e_1 \equiv v_1 \Rightarrow e_1 \in \{ \text{true}, \text{false} \} \)
7. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (C', \text{if } e'_1 \text{ then } e_2 \text{ else } e_3) \) Rule ifE
8. \( e_1 \equiv \text{true} \Rightarrow (C, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (C, e_2) \) Rule ifTrue
9. \( e_1 \equiv \text{false} \Rightarrow (C, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (C, e_3) \) Rule ifFalse
10. \( \exists C', e' : (C, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (C', e') \) 7-9

Case \( \Lambda \cdot \vdash e_1 : \text{Bool} \) \( \Lambda \cdot \vdash e_2 : \tau \)

1. \( \Lambda \vdash C \text{ ok} \)
2. \( (C, \text{while}(e_1)|e_2)) \Rightarrow (C, \text{if } e_1 \text{ then while}(e_1)|e_2) \) 1, Rule whileE

Case \( \Lambda \cdot \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \) (let)

1. \( e = \text{(let } x = e_1 \text{ in } e_2 \text{ end)} \)
2. \( \Lambda \vdash C \text{ ok} \)
3. \( \Lambda \cdot \vdash e_1 : \tau_1 \)
4. \( \Lambda \cdot \vdash e_2 : \tau_2 \)
5. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
6. \( e_1 \equiv v_1 \Rightarrow (C, e) \rightarrow (C, v_1[e]\mid e_2) \) Rule letValue
7. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow \text{let } x = e_1 \text{ in } e_2 \text{ end} \rightarrow \text{let } x = e'_1 \text{ in } e_2 \text{ end} \) Rule letE
8. \( \exists C', e' : (C, e) \rightarrow (C', e') \) 5-7

Case \( \Lambda \cdot \vdash \text{ref } e_1 : \tau \) (createRef)

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda \cdot \vdash e_1 : \tau \)
3. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
4. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{ref } e_1) \rightarrow (C', \text{ref } e'_1) \) Rule refE
5. \( \Lambda \vdash C \text{ ok} \) is only derivable by Rule C-Ok Inspection of Rule C-Ok
6. \( C = (M, R, \text{inOb}, rt, out, \tau_{\text{out}}) \)
7. \( e_1 = v \Rightarrow (C, \text{ref } e_1) \rightarrow ((M \cup \{ \ell, v \}, R, \text{inOb}, rt, out, \tau_{\text{out}}), \ell) \) 6, Rule refValue
8. \( \exists C', e' : (C, e) \rightarrow (C', e') \) 3, 4, 7

Case \( \Lambda \cdot \vdash \text{le } e_1 : \tau \) (accessRef)

1. \( \Lambda \vdash C \text{ ok} \)
2. \( e_1 : \tau \text{ Ref} \)
3. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
4. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{le } e_1) \rightarrow (C', \text{le } e'_1) \) Rule derefE
5. \( e_1 = v \Rightarrow e_1 = e_1 \)
6. \( \Lambda = \Lambda' \cup \{ \ell : \tau \} \) 2, 5, Rule location
7. \( \Lambda \vdash C \text{ ok} \) is only derivable by Rule C-Ok Inspection of Rule C-Ok
8. \( C = (M, R, \text{inOb}, rt, out, \tau_{\text{out}}) \land M : \Lambda \) 1, 7, Inspection of Rule C-Ok
9. \( M : \Lambda \) is only derivable by Rule TMem Inspection of Rule M:A
10. \( M = M' \cup \{ (\ell, v) \} \) 6, 8, 9, Rule TMem
11. \( e_1 = v_1 \Rightarrow (C, \text{! } e_1) \rightarrow (C, v) \) 8, 10, Rule derefV
12. \( \exists C', e' : (C, e) \rightarrow (C', e') \) 3, 4, 11
Case \( \Lambda \bullet \vdash e_1 : \tau \) Ref
\[ \frac{\Lambda \bullet \vdash e_2 : \tau}{\Lambda \bullet \vdash e_1 := e_2 : \tau} \]  
1. \( \Lambda \vdash C \) ok  
2. \( \Lambda \bullet \vdash e_1 : \tau \) Ref  
3. \( \Lambda \bullet \vdash e_2 : \tau \)  
4. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)  
5. \( e_2 = v_2 \) or \( (C, e_2) \rightarrow (C', e'_2) \)  
6. \( e_1 = v_1 \Rightarrow e_1 = e' \)  
7. \( \Lambda \bullet \vdash \ell_1 : \tau \) Ref is only derivable by Rule location  
8. \( \Lambda = \Delta \cup \{ \ell_1 : \tau \} \)  
9. \( C = (M, R, inOb, rt, out, \tau_{out}) \land M : \Lambda \)  
10. \( M : \Lambda \) is only derivable by TMem  
11. \( M' = M' \cup \{ (\ell_1, \tau_{old}) \} \)  
12. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 := e_2) \rightarrow (C', e'_1 := e_2) \)  
13. \( e_1 = v_1 \land (C, e_2) \rightarrow (C', e'_2) \Rightarrow (C, e_1 := e_2) \rightarrow (C', v_1 := e'_2) \)  
14. \( e_1 = v_1 \land e_2 = v_2 \Rightarrow (C, e_1 := e_2) \rightarrow (M' \cup \{ v_1, v_2 \}, R, inOb, rt, out, \tau_{out}), \text{unit} \)  
15. \( \exists C', e', (C, e) \rightarrow (C', e') \)  

Case \( \Lambda \bullet \vdash e_1 : \tau \) Ref
\[ \frac{\Lambda \bullet \vdash e_2 : \tau}{\Lambda \bullet \vdash e_1 :: e_2 : \tau} \]  
1. \( \Lambda \vdash C \) ok  
2. \( \Lambda \bullet \vdash e_1 : \tau \) Ref  
3. \( \Lambda \bullet \vdash e_2 : \tau \)  
4. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)  
5. \( e_2 = v_2 \) or \( (C, e_2) \rightarrow (C', e'_2) \)  
6. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 :: e_2) \rightarrow (C', e'_1 :: e_2) \)  
7. \( (C, e_2) \rightarrow (C', e'_2) \Rightarrow (C, e_1 :: e_2) \rightarrow (C', v_1 :: e'_2) \)  
8. \( e_1 = v_1 \land e_2 = v_2 \land v_1 :: v_2 = v) \Rightarrow (C, e_1 :: e_2) \rightarrow (C, v) \)  
9. \( e = v \lor v' \) or \( \exists C', e', (C, e) \rightarrow (C', e') \)  

Case \( \Lambda \bullet \vdash e_1 : \tau \) Ref
\[ \frac{\Lambda \bullet \vdash e_1 @ e_2 : \tau}{\Lambda \bullet \vdash e_1 :: e_2 : \tau} \]  
1. \( \Lambda \vdash C \) ok  
2. \( \Lambda \bullet \vdash e_1 : \tau \) Ref  
3. \( \Lambda \bullet \vdash e_2 : \tau \)  
4. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)  
5. \( e_2 = v_2 \) or \( (C, e_2) \rightarrow (C', e'_2) \)  
6. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 @ e_2) \rightarrow (C', e'_1 @ e_2) \)  
7. \( e_1 = e_1 \land (C, e_2) \rightarrow (C', e'_2) \Rightarrow (C, e_1 @ e_2) \rightarrow (C', v_1 @ e'_2) \)  
8. \( e_1 = v_1 \Rightarrow e_1 = e'_1 \) or \( v_1 = v_1 \) or \( e_1 = [] : \tau_{List} \)  
9. \( e_2 = v_2 \Rightarrow e_2 = e'_{n+1} \) or \( e_2 = [] : \tau_{List} \)  
10. \( (e_1 = [] : \tau_{List} \and e_2 = v_2) \Rightarrow (C, (e_1 @ e_2)) \rightarrow (C, v_2) \)  
11. \( e_1 = e'_{n} \) or \( e_2 = [] : \tau_{List} \)  
12. \( e_1 = e'_{n} \) or \( e_2 = e'_{m} \)  
13. \( e = v \lor \exists C', e', (C, e) \rightarrow (C', e') \)
Case \( \Lambda \vdash e_1 : \tau_{\text{List}} \) (head)

1. \( \Lambda \vdash \text{C ok} \) assumption
2. \( \Lambda, \bullet \vdash e_1 : \tau_{\text{List}} \) assumption
3. \( e_1 = \nu \lor (C, e_1) \rightarrow (C', e'_1) \) 1, 2, IH
4. \( e_1 = \nu \Rightarrow e_1 = \nu_1 :: \nu_2 \lor e_1 = [] : \tau_{\text{List}} \) 1, Lemma Canonical Forms
5. \( e_1 = [] : \tau_{\text{List}} \Rightarrow (C, \text{head}(e_1)) \rightarrow (C, \text{in}_\text{none}(\text{unit}) : \tau \text{ Option}) \) listHeadEmpty
6. \( e_1 = \nu_1 :: \nu_2 \Rightarrow (C, \text{head}(e_1)) \rightarrow (C, \text{in}_\text{some}(\nu_1) : \tau \text{ Option}) \) listHeadValue
7. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{head}(e_1)) \rightarrow (C', \text{head}(e'_1)) \) listHeadE
8. \( \exists C', e', (C, e) \rightarrow (C', e') \) 5-7

Case \( \Lambda \vdash e_1 : \tau_{\text{List}} \) (tail)

1. \( \Lambda \vdash \text{C ok} \) assumption
2. \( \Lambda, \bullet \vdash e_1 : \tau_{\text{List}} \) assumption
3. \( e_1 = \nu \lor (C, e_1) \rightarrow (C', e'_1) \) 1, 2, IH
4. \( e_1 = \nu \Rightarrow e_1 = \nu_1 :: \nu_2 \lor e_1 = [] : \tau_{\text{List}} \) 2, Lemma Canonical Forms
5. \( e_1 = [] : \tau_{\text{List}} \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C, [] : \tau_{\text{List}}) \) Rule listTailNil
6. \( e_1 = \nu_1 :: \nu_2 \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C, \nu_2) \) Rule listTailCons
7. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C', \text{tail}(e'_1)) \) Rule listTailE
8. \( \exists C', e', (C, e) \rightarrow (C', e') \) 5-7

Case \( \Lambda, \bullet \vdash e_1 : \tau_{\text{List}} \) (empty)

1. \( \Lambda \vdash \text{C ok} \) assumption
2. \( \Lambda, \bullet \vdash e_1 : \tau_{\text{List}} \) assumption
3. \( e_1 = \nu \lor (C, e_1) \rightarrow (C', e'_1) \) 1, 2, IH
4. \( e_1 = \nu \Rightarrow e_1 = \nu_1 :: \nu_2 \lor e_1 = [] : \tau_{\text{List}} \) 2, Lemma Canonical Forms
5. \( e_1 = [] : \tau_{\text{List}} \Rightarrow (C, \text{empty}(e_1) \rightarrow (C, \text{true}) \) Rule listTailNil
6. \( e_1 = \nu_1 :: \nu_2 \Rightarrow (C, \text{empty}(e_1) \rightarrow (C, \text{false}) \) Rule listTailCons
7. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C', \text{tail}(e'_1)) \) Rule listTailE
8. \( \exists C', e', (C, e) \rightarrow (C', e') \) 5-7

Case \( \Lambda, \bullet \vdash e_1 : \tau_1, \ldots, \Lambda, \bullet \vdash e_n : \tau_n \) (record)

1. \( \Lambda \vdash \text{C ok} \) assumption
2. \( e = (\ell_1 = e_1, \ldots, \ell_n = e_n) \) assumption
3. \( \Lambda, \bullet \vdash e_1 : \tau_1, \ldots, \Lambda, \bullet \vdash e_n : \tau_n \) assumption
4. \( e_1 = \nu_1 \lor (C, e_1) \rightarrow (C', e'_1), \ldots, e_n = \nu_n \lor (C, e_n) \rightarrow (C', e'_n) \) 3, IH
5. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e) \rightarrow (C', (\ell_1 = e'_1, \ldots, \ell_n = e_n)) \) 1, 2, variantE
6. \( \vdots \)
7. \( e_1 = \nu_1, \ldots, e_{i-1} = \nu_{i-1}, (C, e_i) \rightarrow (C', e'_i) \Rightarrow (C, e) \rightarrow (C', (\ell_1 = e_1, \ldots, \ell_i = e'_i, \ldots, \ell_n = e_n)) \) 1, 2, variantE
8. \( \vdots \)
9. \( e_1 = \nu_1, \ldots, e_{n-1} = \nu_{n-1}, (C, e_n) \rightarrow (C', e'_n) \Rightarrow (C, e) \rightarrow (C', (\ell_1 = e_1, \ldots, \ell_n = e'_n)) \) 1, 2, variantE
10. \( \vdots \)
11. \( e = \nu \lor \exists C', e', (C, e) \rightarrow (C', e') \) 1, definition of values
12. \( \vdots \) 5-6
Case \( \Lambda \vdash e_1 : (\ell_1 : \tau_1, \ldots, \ell_n : \tau_n) \quad i \in \{1, \ldots, n\} \) (projection)

1. \( \Lambda \vdash C \quad \text{ok} \)
2. \( \Lambda \vdash e_1 : (\ell_1 : \tau_1, \ldots, \ell_n : \tau_n) \)
3. \( i \in \{1, \ldots, n\} \)
4. \( e_1 = v_i \) or \( (C, e_1) \rightarrow (C', e_i') \)
5. \( e_1 = v_i \Rightarrow e_1 = (\ell_1 : v_{i_1}, \ldots, \ell_n : v_{i_n}) \)
6. \( (C, e_1) \rightarrow (C', e_i) \Rightarrow (C, e_1) \rightarrow (C', e_i') \)
7. \( e_1 = (\ell_1 : v_{i_1}, \ldots, \ell_n : v_{i_n}) \Rightarrow (C, e_1, \ell_1) \rightarrow (C, v_i) \)
8. \( e = v \) or \( \exists \forall C', e', (C, e) \rightarrow (C', e') \)

Case \( \Lambda \vdash (i \in \{1, \ldots, n\}) \) (variant)

1. \( e = \text{in}_{\ell_i} e_1 : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n \quad \text{assumption} \)
2. \( \Lambda \vdash C \quad \text{ok} \)
3. \( \Lambda \vdash e_i : \tau_i \quad \text{assumption} \)
4. \( (C, e_i) \rightarrow (C', e_i') \Rightarrow (C, e) \rightarrow (C', e_i') \) (1, Rule variantE)
5. \( e = v \) or \( \forall C', e', (C, e) \rightarrow (C', e') \)
6. \( \Lambda \vdash (i \in \{1, \ldots, n\}) (\Lambda \vdash \{x_1 : \tau_1\} \vdash e_1 : \tau \cdots \Lambda \vdash \{x_n : \tau_n\} \vdash e_n : \tau) \) (case)

Case \( \Lambda \vdash \text{makeTypedVal}(\tau_1, e_1) : \text{TypedVal} \) (makeTypedVal)

1. \( \Lambda \vdash C \quad \text{ok} \)
2. \( \Lambda \vdash e_1 : \tau_1 \quad \text{assumption} \)
3. \( e_1 = v_i \) or \( (C, e_1) \rightarrow (C', e_i') \) (1, 2, IH)
4. \( (C, e_1) \rightarrow (C', e_i') \Rightarrow (C, \text{makeTypedVal}(\tau_1, e_1)) \rightarrow (C', \text{makeTypedVal}(\tau_1, e_i')) \)
5. \( e_1 = v_i \Rightarrow \text{makeTypedVal}(\tau_1, e_1) = v \quad \text{Definition of values} \)
6. \( e = v \) or \( \exists C', e', (C, e) \rightarrow (C', e') \)

Case \( \Lambda \vdash \text{tryCast}(\tau_1, e_1) : \tau_1 \quad \text{Option} \) (tryCast)

1. \( \Lambda \vdash C \quad \text{ok} \)
2. \( \Lambda \vdash e_1 : \text{TypedVal} \quad \text{assumption} \)
3. \( e_1 = v_i \) or \( (C, e_1) \rightarrow (C', e_i') \) (1, 2, IH)
4. \( (C, e_1) \rightarrow (C', e_i') \Rightarrow (C, \text{tryCast}(\tau_1, e_1)) \rightarrow (C', \text{tryCast}(\tau_1, e_i')) \)
5. \( e_1 = v_i \Rightarrow \text{makeTypedVal}(\tau_1, e_2) \)
6. \( e_1 = \text{makeTypedVal}(\tau_2, e_2) \land \tau_1 = \tau_2 \Rightarrow (C, \text{tryCast}(\tau_1, e_1)) \rightarrow (C, \text{inSome} v_2 : \tau_1 \quad \text{Option}) \)
7. \( e_1 = \text{makeTypedVal}(\tau_2, e_2) \land \tau_1 = \tau_2 \Rightarrow (C, \text{tryCast}(\tau_1, e_1)) \rightarrow (C, \text{inSome} \quad \text{unit} : \tau_1 \quad \text{Option}) \)
8. \( \exists C', e', (C, e) \rightarrow (C', e') \)
Case \( \Lambda \cdot e_1 : \tau \) (endLabel)

1. \( \Lambda + C \) ok
2. \( \Lambda + C \) ok is only derivable by Rule C-Ok
3. \( C = (M, R, inOb, rt, out, \tau_{out}) \)
4. \( \Lambda \cdot e_1 : \tau \)
5. \( e_1 = \nu_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
6. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \{ e \}_{\xi(v_2)}) \rightarrow (C', \{ e' \}_{\xi(v_2)}) \)
7. \( e_1 = \nu_1 \land s \neq "monitor'" \Rightarrow (C, \{ e \}_{\xi(v_2)}) \rightarrow (C, \nu_1) \)
8. \( e_1 = \nu_1 \land s = "monitor" \Rightarrow (C, \{ e \}_{\xi(v_2)}) \rightarrow (M, R, false, rt, out, \tau_{out}, \nu_1) \)
9. \( \exists C', e' : (C, e) \rightarrow (C', e') \)

Case \( \Lambda \cdot \nu_1 : \tau \) (getRT)

1. \( \Lambda + C \) ok
2. \( \Lambda + C \) ok is only derivable by Rule C-Ok
3. \( C = (M, R, inOb, rt, out, \tau_{out}) \)
4. \( (C, getRT()) \rightarrow (C, rt) \)

Case \( \Lambda \cdot e_1 : Obligation \) (makeCFG)

1. \( \Lambda \cdot e_1 : Obligation \)
2. \( e_1 = \nu_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
3. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, makeCFG(e_1)) \rightarrow (C', makeCFG(e'_1)) \)
4. \( e_1 = \nu_1 \Rightarrow (C, makeCFG(e_1)) \rightarrow (C, \{ makeCFG_{\xi(v_2)}(v_1) \}) \)
5. \( \exists C', e' : (C, e) \rightarrow (C', e') \)

Case \( \Lambda \cdot e_1 : Event \) (setOutput)

1. \( \Lambda + C \) ok
2. \( e_1 : Event \)
3. \( \Lambda + C \) ok is only derivable by Rule C-Ok
4. \( C = (M, R, inOb, rt, out, \tau_{out}) \)
5. \( \Lambda \cdot e_1 : \tau \) Event Option
6. \( out \) is a value
7. \( e_1 = \nu_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)
8. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, setOutput(e_1)) \rightarrow (C', setOutput(e'_1)) \)
9. \( e_1 = \nu_1 \Rightarrow \)
   a. \( out = in_{\text{none}} \) unit : \( \tau \) Event Option \( \lor \exists \nu_2 : out = in_{\text{some}} \) \( \nu_2 : \tau \) Event Option
   b. \( out = in_{\text{some}} \) \( \nu_2 : \tau \) Event Option \( \Rightarrow (C, setOutput(e_1)) \rightarrow (C, false) \)
   c. \( out = in_{\text{none}} \) unit : \( \tau \) Event Option
      i. \( e_1 = in_{\text{act}} v_3 : Event \lor e_1 = in_{\text{res}} res(v_4, makeTypedVal(r, v_5)) : Event \)
      ii. \( e_1 = in_{\text{act}} v_3 : Event \Rightarrow (C, setOutput(e_1)) \rightarrow ((M, R, inOb, rt, in_{\text{some}}(in_{\text{act}} v : \tau \text{ Event Option}, \tau_{out})), true) \)
      iii. \( e_1 = in_{\text{res}} res(v_4, makeTypedVal(r, v_5)) \land \tau = \tau_{out} \Rightarrow (C, setOutput(e_1)) \rightarrow ((M, R, inOb, rt, in_{\text{some}}(in_{\text{res}} res(v_1, v_2) : \tau_{out} \text{ Event Option}, \tau_{out})), true) \)
      iv. \( e_1 = in_{\text{res}} res(v_4, makeTypedVal(r, v_5)) \land \tau \neq \tau_{out} \Rightarrow (C, setOutput(e_1)) \rightarrow (C, false) \)
   d. \( out = in_{\text{none}} \) unit : \( \tau \) Event Option \( \Rightarrow \exists C', e' : (C, setOutput(e_1)) \rightarrow (C, e') \)
   e. \( \exists C', e' : (C, setOutput(e_1)) \rightarrow (C, e') \)
10. \( e_1 = \nu_1 \Rightarrow \exists C', e' : (C, setOutput(e_1)) \rightarrow (C, e') \)
11. \( \exists C', e' : (C, setOutput(e_1)) \rightarrow (C, e') \)

53
Case \( \Lambda \cdot \vdash \text{outputNotSet}() : \text{Bool} \) (outputNotSet)

1. \( \Lambda \vdash \text{C-ok} \)
2. \( \Lambda \vdash \text{C-ok} \) is only derivable by Rule C-Ok
3. \( C = (M, R, \text{inOb}, \text{rt}, \text{out}, \tau_{\text{out}}) \)
4. \( \Lambda \cdot \vdash \text{out} : \tau_{\text{out}} \text{ Event Option} \)
5. \( \text{out} \) is a value
6. \( \text{out} = \text{in}_{\text{none}} \text{ unit} : \tau_{\text{out}} \text{ Event Option} \lor \exists v_2 : \text{out} = \text{in}_{\text{some}} v_2 : \tau \text{ Event Option} \)
7. \( \text{out} = \text{in}_{\text{none}} \text{ unit} : \tau_{\text{out}} \text{ Event Option} \Rightarrow (C, \text{outputNotSet}(\cdot)) \rightarrow (C, \text{false}) \)
8. \( \exists v_2 : \tau_{\text{out}} \text{ Event Option} \Rightarrow (C, \text{outputNotSet}(\cdot)) \rightarrow (C, e') \)

Case \( \Lambda \cdot \vdash \text{getOutput}() : \text{Event Option} \) (getOutput)

1. \( \Lambda \vdash \text{C-ok} \)
2. \( \Lambda \vdash \text{C-ok} \) is only derivable by Rule C-Ok
3. \( C = (M, R, \text{inOb}, \text{rt}, \text{out}, \tau_{\text{out}}) \)
4. \( \Lambda \cdot \vdash \text{out} : \tau_{\text{out}} \text{ Event Option} \)
5. \( \text{out} \) is a value
6. \( \text{out} = \text{in}_{\text{none}} \text{ unit} : \tau_{\text{out}} \text{ Event Option} \lor \exists v_2 : \text{out} = \text{in}_{\text{some}} v_2 : \tau \text{ Event Option} \)
7. \( \text{out} = \text{in}_{\text{none}} \text{ unit} : \tau_{\text{out}} \text{ Event Option} \Rightarrow (C, \text{getOutput}(\cdot)) \rightarrow (C, \text{in}_{\text{none}} \text{ unit} : \tau_{\text{out}} \text{ Event Option}) \)
8. \( \text{out} = \text{in}_{\text{some}} v_2 : \tau_{\text{out}} \text{ Event Option} \Rightarrow \)
   a. \( v_2 = \text{in}_{\text{act}} v_3 : \tau_{\text{out}} \text{ Event Option} \lor v_2 = \text{in}_{\text{res}} \text{res}(v_4, v_5) : \tau_{\text{out}} \text{ Event Option} \)
   b. \( v_2 = \text{in}_{\text{act}} v_3 : \tau_{\text{out}} \text{ Event Option} \Rightarrow (C, \text{getOutput}(\cdot)) \rightarrow (C, \text{in}_{\text{act}} (v_3 : \text{Event}) : \tau_{\text{out}} \text{ Event Option}) \)
   c. \( \exists v_2 : \tau_{\text{out}} \text{ Event Option} \Rightarrow (C, \text{getOutput}(\cdot)) \rightarrow (C, e') \)
9. \( \text{out} = \text{in}_{\text{some}} v_2 : \tau_{\text{out}} \text{ Event Option} \Rightarrow \exists v_3 : \text{out} = \text{makeTypedVal}(v_3, v_3) : \text{Event} \text{ Event Option} \)
10. \( \exists v_3 : \text{out} = \text{makeTypedVal}(v_3, v_3) : \text{Event} \text{ Event Option} \Rightarrow (C, e') \)

Case \( \Lambda \cdot \vdash \text{monitor}((\tau_1, e_1)) : \tau_1 \text{ Event} \) (monitor)

1. \( \Lambda \vdash \text{C-ok} \)
2. \( \Lambda \vdash \text{C-ok} \) is only derivable by Rule C-Ok
3. \( e_1 = v_1 \text{ or } (C, e_1) \rightarrow (C', e'_1) \)
4. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{monitor}(\tau_1, e_1)) \rightarrow (C', \text{monitor}(\tau_1, e'_1)) \)
5. \( \Lambda \vdash \text{C-ok} \) is only derivable by Rule C-Ok
6. \( C = (M, R, \text{inOb}, \text{rt}, \text{out}, \tau_{\text{out}}) \)
7. \( e_1 = v_1 \Rightarrow (C, \text{monitor}(\tau_1, v_1)) \rightarrow (C, v_1) \)
   \( ((M, R, \text{true}, \text{inOb}, \text{rt}, \text{out}, \tau_{\text{out}}), \{e_{\text{monitor}}(\tau_1, v_1)\}_{\text{monitor}(v_1)}) \) \( \rightarrow (C, v_1) \)
8. \( \exists v_2 : (C, e) \rightarrow (C', e') \)
Case \( \Lambda \bullet \vdash e_1 : \text{String} \quad \Lambda \bullet \vdash e_2 : \text{TypedVal} \)

\( \Lambda \bullet \vdash \text{invoke}(e_1, e_2) : \text{TypedVal Option} \)

1. \( \Lambda \vdash \text{C ok} \)  
   assumption

2. \( \Lambda \bullet \vdash e_1 : \text{String} \)  
   assumption

3. \( \Lambda \bullet \vdash e_2 : \text{TypedVal} \)  
   assumption

4. \( e_1 = v_1 \) or \( (C, e_1) \rightarrow (C', e'_1) \)

5. \( e_1 = v_1 \Rightarrow e_1 = s \)

6. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C', \text{invoke}(e'_1, e_2)) \)

7. \( e_2 = v_2 \) or \( (C, e_2) \rightarrow (C', e'_2) \)

8. \( e_1 = s \land e_2 \rightarrow e'_2 \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C', \text{invoke}(e'_1, e'_2)) \)

9. \( e_1 = s \land e_2 = v_2 \Rightarrow \)

   a. \( \forall s, f : (s, f) \notin F \lor (\exists x_1, x_2, \tau_1, \tau_2, e : (s, \text{fun} x_1(x_2 : \tau_1) : \tau_2 = e) \in F) \)

   b. \( \forall s, f : (s, f) \notin F \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C, \text{in} \text{none} \text{ unit} : \text{TypedVal Option}) \)

   c. \( (\exists x_1, x_2, \tau_1, \tau_2, e : (s, \text{fun} x_1(x_2 : \tau_1) : \tau_2 = e) \in F) \Rightarrow \)

      i. \( v_2 = \text{makeTypedVal}(\tau_3, v'_2) \)

      ii. \( \tau_1 = \tau_3 \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C, \text{in} \text{some} \text{ makeTypedVal}(\tau_2, \text{call}((\text{fun} x_1(x_2 : \tau_1) : \tau_2 = e), v'_2)) : \text{TypedVal Option}) \)

     iii. \( \tau_1 \neq \tau_3 \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C, \text{in} \text{none} \text{ unit} : \text{TypedVal Option}) \)

   d. \( (\exists x_1, x_2, \tau_1, \tau_2, e : (s, \text{fun} x_1(x_2 : \tau_1) : \tau_2 = e) \in F) \Rightarrow \exists C', e', (C, e) \rightarrow (C', e') \)

   e. \( \exists C', e', (C, e) \rightarrow (C', e') \)

10. \( e_1 = s \land e_2 = v_2 \Rightarrow \exists C', e', (C, e) \rightarrow (C', e') \)

11. \( \exists C', e', (C, e) \rightarrow (C', e') \)

   4, 6, 7, 8, 10

2, Lemma Canonical Forms
1, 2, IH
1, 3, IH
5, Rule invokeE2
5, 9, invokeValueNotExists
3, 9, Lemma Canonical Forms
9, c, i, Rule invokeValueExistsOk
9, c, i, Rule invokeValueExistsBad
ii, iii
a, b, d
9, 9(e)
The proof of this lemma is trivial though uninteresting. The proof technique is to derive the proof tree of \( \Lambda \). It is assumed throughout this proof that any \( \Lambda \) is only derivable by rule C-ok.

**Lemma 10 (Monitor Type).** \( \Lambda \vdash C \text{ ok} \) is only derivable by rule C-ok.

**Lemma 11 (Preservation).** \( \Lambda \vdash (C, e) : \tau \land (C, e) \rightarrow (C', e') \rightarrow \exists \Lambda' : \Lambda \vdash C \text{ ok} \land \Lambda' \vdash (C', e') : \tau \land \Lambda \subseteq \Lambda' \).
Case \((C, e_1) \rightarrow (C', e_1')\) (andE)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash e_1 \land e_2 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e_1')\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \text{ Bool}\) \(2\), Inversion Lemma
5. \(\Lambda, \bullet \vdash e_2 : \text{ Bool}\) \(2\), Inversion Lemma
6. \(\tau = \text{ Bool}\) \(2\), Inversion Lemma
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e_1' : \text{ Bool} \land \Lambda \subseteq \Lambda')\) \(1, 3, 4\), IH
8. \(\Lambda', \bullet \vdash e_2 : \text{ Bool}\) \(5, 7\), A-weakening Lemma
9. \(\Lambda', \bullet \vdash e_1' \land e_2 : \tau\) Result is from 7, 9

Case \((C, \text{ true} \land e_2) \rightarrow (C, e_2)\) (andTrue)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{ true} \land e_2 : \tau\) assumption
3. \(\tau = \text{ Bool}\) \(2\), Inversion Lemma
4. \(\Lambda, \bullet \vdash e_2 : \tau\) \(2, 3\), Inversion Lemma
Result is from 1, 4

Case \((C, \text{ false} \land e_2) \rightarrow (C, \text{ false})\) (andFalse)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{ false} \land e_2 : \tau\) assumption
3. \(\tau = \text{ Bool}\) \(2\), Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{ false} : \tau\) \(3\), Rule boolVal
Result is from 1, 4

Case \((C, e_1) \rightarrow (C', e_1')\) (orE)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash e_1 \lor e_2 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e_1')\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \text{ Bool}\) \(2\), Inversion Lemma
5. \(\Lambda, \bullet \vdash e_2 : \text{ Bool}\) \(2\), Inversion Lemma
6. \(\tau = \text{ Bool}\) \(2\), Inversion Lemma
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e_1' : \text{ Bool} \land \Lambda \subseteq \Lambda')\) \(1, 3, 4\), IH
8. \(\Lambda', \bullet \vdash e_2 : \text{ Bool}\) \(5, 7\), A-weakening Lemma
9. \(\Lambda', \bullet \vdash e_1' \lor e_2 : \tau\) Result is from 7, 9

Case \((C, \text{ true} \lor e_2) \rightarrow (C, \text{ true})\) (orTrue)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{ true} \lor e_2 : \tau\) assumption
3. \(\tau = \text{ Bool}\) \(2\), Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{ true} : \tau\) \(3\), Rule boolVal
Result is from 1, 4

Case \((C, \text{ false} \lor e_2) \rightarrow (C, e_2)\) (orFalse)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{ false} \lor e_2 : \tau\) assumption
3. \(\tau = \text{ Bool}\) \(2\), Inversion Lemma
4. \(\Lambda, \bullet \vdash e_2 : \tau\) \(2, 3\), Inversion Lemma
Result is from 1, 4
Case (C, e₁) \rightarrow (C', e'_1) \quad (\text{notE})
1. \Lambda \vdash \text{notFalse} 
2. \Lambda, \bullet \vdash \text{notFalse} : \tau
3. (C, e₁) \rightarrow (C', e'_1)
4. \Lambda, \bullet \vdash e₁ : \text{Bool}
5. \tau = \text{Bool}
6. \exists \Lambda' : (\Lambda' \vdash \text{notFalse} \land \Lambda' \vdash e'_1 : \text{bool} \land \Lambda \subseteq \Lambda')
7. \Lambda', \bullet \vdash -e'_1 : \tau
Result is from 6, 7

Case (C, \text{notFalse}) \rightarrow (C, \text{true}) \quad (\text{notFalse})
1. \Lambda \vdash \text{true}
2. \Lambda, \bullet \vdash \text{false} : \tau
3. \tau = \text{bool}
4. \Lambda, \bullet \vdash \text{true} : \tau
Result is from 1, 4

Case (C, \text{nottrue}) \rightarrow (C, \text{false}) \quad (\text{notTrue})
1. This case is analogous to case notFalse

Case (C, e₁) \rightarrow (C', e'_1) \quad (\text{eqE1})
1. \Lambda \vdash \text{e₁} == e₂
2. \Lambda, \bullet \vdash \text{e₁} == e₂ : \tau
3. (C, e₁) \rightarrow (C', e'_1)
4. \Lambda, \bullet \vdash e₁ : \tau₁
5. \Lambda, \bullet \vdash e₂ : \tau₁
6. \tau₁ = \{\text{Int, Bool, String}\}
7. \tau = \text{Bool}
8. \exists \Lambda' : (\Lambda' \vdash \text{true} \land \Lambda' \vdash e'_1 : \tau₁ \land \Lambda \subseteq \Lambda')
9. \Lambda', \bullet \vdash e₂ : \tau₁
10. \Lambda', \bullet \vdash e'_1 == e₂ : \tau
Result is from 8, 10

Case (C, v₁ == e₂) \rightarrow (C', v₁ == e'_2) \quad (\text{eqE2})
1. \Lambda \vdash \text{v₁} == e₂
2. \Lambda, \bullet \vdash \text{v₁} == e₂ : \tau
3. (C, e₂) \rightarrow (C', e'_2)
4. \Lambda, \bullet \vdash \text{v₁} : \tau₁
5. \Lambda, \bullet \vdash \text{e₂} : \tau₁
6. \tau₁ = \{\text{Int, Bool, String}\}
7. \tau = \text{Bool}
8. \exists \Lambda' : (\Lambda' \vdash \text{true} \land \Lambda' \vdash e'_2 : \tau₁ \land \Lambda \subseteq \Lambda')
9. \Lambda', \bullet \vdash \text{v₁} : \tau₁
10. \Lambda', \bullet \vdash \text{v₁} == e'_2 : \tau
Result is from 8, 10
Case \( n_1 = n_2 \) (eqIntTrue)

1. \( \Lambda \vdash C \text{ ok} \) assumption
2. \( \Lambda \cdot \vdash n_1 = n_2 : \tau \) assumption
3. \( \tau = \text{Bool} \) 2, Inversion Lemma
4. \( \Lambda \cdot \vdash \text{true} : \tau \) 3, Rule boolVal

Result is from 1, 4

Case \( n_1 \neq n_2 \) (eqIntFalse)

1. \( \Lambda \vdash C \text{ ok} \) assumption
2. \( \Lambda \cdot \vdash n_1 = n_2 : \tau \) assumption
3. \( \tau = \text{Bool} \) 2, Inversion Lemma
4. \( \Lambda \cdot \vdash \text{false} : \tau \) 3, Rule boolVal

Result is from 1, 4

Case \( s_1 = s_2 \) (eqStrTrue)

1. This case is analogous to case eqIntTrue

Case \( s_1 \neq s_2 \) (eqStrFalse)

1. This case is analogous to case eqIntFalse

Case \( (C, \text{true} == b_2) \) (eqBoolTrue)

1. \( \Lambda \vdash C \text{ ok} \) assumption
2. \( \Lambda \cdot \vdash \text{true} == b_2 : \tau \) assumption
3. \( \tau = \text{Bool} \) 2, Inversion Lemma
4. \( \Lambda \cdot \vdash b_2 : \tau \) 2, 3, Inversion Lemma

Result is from 1, 4

Case \( (C, \text{false} == b_2) \) (eqBoolFalse)

1. \( \Lambda \vdash C \text{ ok} \) assumption
2. \( \Lambda \cdot \vdash \text{false} == b_2 : \tau \) assumption
3. \( \tau = \text{Bool} \) 2, Inversion Lemma
4. \( \Lambda \cdot \vdash \neg b_2 : \tau \) 3, 4, Rule negation

Result is from 1, 5

Case \( (C, e_1) \) (addE1)

1. \( \Lambda \vdash C \text{ ok} \) assumption
2. \( \Lambda \cdot \vdash e_1 + e_2 : \tau \) assumption
3. \( (C, e_1) \) \( \rightarrow \) (eqIntTrue)
4. \( \Lambda \cdot \vdash e_1 : \text{Int} \) 2, Inversion Lemma
5. \( \Lambda \cdot \vdash e_2 : \text{Int} \) 2, Inversion Lemma
6. \( \tau = \text{Int} \) 2, Inversion Lemma
7. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \cdot \vdash e'_1 : \text{Int} \land \Lambda \subseteq \Lambda') \)
8. \( \Lambda' \cdot \vdash e_2 : \text{Int} \) 1, 3, 4, IH
9. \( \Lambda' \cdot \vdash e'_1 + e_2 : \text{Int} \) 5, 7, A-weakening Lemma

Result is from 7, 9
Case \[\frac{(C, e_2) \rightarrow (C', e'_2)}{(C, n_1 + e_2) \rightarrow (C', n_1 + e'_2)}\] (addE2)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash n_1 + e_2 : \tau\) assumption
3. \((C, e_2) \rightarrow (C', e'_2)\) assumption
4. \(\Lambda, \bullet \vdash e_2 : \text{Int}\) 2, Inversion Lemma
5. \(\tau = \text{Int}\) 2, Inversion Lemma
6. \(\exists \Lambda' : (\Lambda' + C' \text{ ok} \land \Lambda', \bullet \vdash e'_2 : \text{Int} \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
7. \(\Lambda', \bullet \vdash n_1 : \text{Int}\) Rule IntVal
8. \(\Lambda', \bullet \vdash n_1 + e'_2 : \text{Int}\) 5-7, Rule add

Result is from 6, 8

Case \[\frac{n_1 + n_2 = n}{(C, n_1 + n_2) \rightarrow (C, n)}\] (addValue)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash n_1 + n_2 : \tau\) assumption
3. \(\tau = \text{Int}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash n : \tau\) 3, Rule IntVal

Result is from 1, 4

Case \[\frac{(C, e_1) \rightarrow (C', e'_1)}{(C, e_1; e_2) \rightarrow (C', e'_1; e_2)}\] (sequenceE1)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash e_1; e_2 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau'\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash e_2 : \tau\) 2, Inversion Lemma
6. \(\exists \Lambda' : (\Lambda' + C' \text{ ok} \land \Lambda', \bullet \vdash e'_1 : \tau' \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
7. \(\Lambda', \bullet \vdash e_2 : \tau\) 5, 6, A-weakening Lemma
8. \(\Lambda', \bullet \vdash e'_1; e_2 : \tau\) 6, 7, Rule sequence

Result is from 6, 8

Case \[\frac{(C, e_1; e_2) \rightarrow (C, e_2)}{(C, e_1; e_2)}\] (sequenceE2)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash e_1; e_2 : \tau\) assumption
3. \(\Lambda, \bullet \vdash e_2 : \tau\) 2, Inversion Lemma

Result is from 1, 3

Case \[\frac{(C, e_1) \rightarrow (C', e'_1)}{(C, \text{ if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (C', \text{ if } e'_1 \text{ then } e_2 \text{ else } e_3)}\] (IE)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{ if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \text{Bool}\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash e_2 : \tau\) 2, Inversion Lemma
6. \(\Lambda, \bullet \vdash e_3 : \tau\) 2, Inversion Lemma
7. \(\exists \Lambda' : (\Lambda' + C' \text{ ok} \land \Lambda', \bullet \vdash e'_1 : \text{Bool} \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
8. \(\Lambda', \bullet \vdash e_2 : \tau\) 5, 7, A-weakening Lemma
9. \(\Lambda', \bullet \vdash e_3 : \tau\) 6, 7, A-weakening Lemma
10. \(\Lambda', \bullet \vdash \text{ if } e'_1 \text{ then } e_2 \text{ else } e_3 : \tau\) 7-9, Rule if

Result is from 7, 10
Case \((C, \text{if true then } e_2 \text{ else } e_3) \rightarrow (C, e_2)\) (\text{ifTrue})

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \• \vdash \text{if true then } e_2 \text{ else } e_3 : \tau\) assumption
3. \(\Lambda, \• \vdash e_2 : \tau\) 2, Inversion Lemma
   Result is from 1, 3

Case \((C, \text{if false then } e_2 \text{ else } e_3) \rightarrow (C, e_3)\) (\text{ifFalse})

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \• \vdash \text{if false then } e_2 \text{ else } e_3 : \tau\) assumption
3. \(\Lambda, \• \vdash e_3 : \tau\) 2, Inversion Lemma
   Result is from 1, 3

Case \((C, \text{while}(e_1) \{ e_2 \}) \rightarrow (C, \text{if } e_1 \text{ then } (e_2; \text{while}(e_1) \{ e_2 \}) \text{ else } \text{false})\) (\text{while})

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \• \vdash \text{while}(e_1) \{ e_2 \} : \tau\) assumption
3. \(\Lambda, \• \vdash e_1 : \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda, \• \vdash e_2 : \tau_1\) 2, Inversion Lemma
5. \(\tau = \text{Bool}\) 2, Inversion Lemma
6. \(\Lambda, \• \vdash \text{false} : \text{Bool}\) Rule boolVal
7. \(\Lambda, \• \vdash (e_2; \text{while}(e_1) \{ e_2 \}) : \text{Bool}\) 2, 4, 5, Rule sequence
8. \(\Lambda, \• \vdash \text{if } e_1 \text{ then } (e_2; \text{while}(e_1) \{ e_2 \}) \text{ else } \text{false} : \tau\) 3, 5-7, Rule if
   Result is from 1, 8

Case \((C, e_1) \rightarrow (C', e'_1)\) (\text{let})

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \• \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \• \vdash e_1 : \tau_1\) 2, Inversion Lemma
5. \(\Lambda, \{ x : \tau_1 \} \vdash e_2 : \tau_2\) 2, Inversion Lemma
6. \(\tau = \tau_2\) 2, Inversion Lemma
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \land \Lambda', \• \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
8. \(\Lambda', \{ x : \tau_1 \} \vdash e_2 : \tau_2\) 5, 7, \(A\)-weakening Lemma
9. \(\Lambda', \• \vdash \text{let } x = e'_1 \text{ in } e_2 \text{ end} : \tau\) 6, 7, 8, Rule let
   Result is from 7, 9

Case \((C, \text{let } x = u \text{ in } e_2 \text{ end}) \rightarrow (C, [u/x]e_2)\) (\text{letValue})

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \• \vdash \text{let } x = u \text{ in } e_2 \text{ end} : \tau\) assumption
3. \(\Lambda, \• \vdash u : \tau_1\) 2, Inversion Lemma
4. \(\Lambda, \{ x : \tau_1 \} \vdash e_2 : \tau_2\) 2, Inversion Lemma
5. \(\tau = \tau_2\) 2, Inversion Lemma
6. \(\Lambda, \• \vdash [u/x]e_2 : \tau\) 3-5, Substitution Lemma
   Result is from 1, 6
Case  \[(C, e_1) \rightarrow (C', e'_1)\]
\[(C, \text{ref } e_1) \rightarrow (C', \text{ref } e'_1)\] (refVal)

1. \(\Lambda \vdash C \text{ ok}\) \hspace{10cm} assumption
2. \(\Lambda, \bullet \vdash \text{ref } e_1 : \tau\) \hspace{10cm} assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) \hspace{10cm} assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau_1\) \hspace{10cm} assumption
5. \(\tau = \tau_1 \text{ Ref}\) \hspace{10cm} 2, Inversion Lemma
6. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \wedge \Lambda', \bullet \vdash e'_1 : \tau_1 \wedge \Lambda \subseteq \Lambda')\)
7. \(\Lambda', \bullet \vdash \text{ref } e'_1 : \tau\)

Case  \((M, R, \text{inOb}, rt, \text{out}, \tau_{\text{out}}, \text{ref } v) \rightarrow ((M \cup \{(\ell, v)\}, R, \text{inOb}, rt, \text{out}, \tau_{\text{out}}), \ell)\) (refValue)

1. let \((M, R, \text{inOb}, rt, \text{out}, \tau_{\text{out}}) = C\) \hspace{10cm} assumption
2. let \((M \cup \{(\ell, v)\}, R, \text{inOb}, rt, \text{out}, \tau_{\text{out}}) = C'\) \hspace{10cm} assumption
3. \(\Lambda \vdash C \text{ ok}\) \hspace{10cm} assumption
4. \(\Lambda, \bullet \vdash v : \tau\) \hspace{10cm} assumption
5. \(\Lambda, \bullet \vdash v : \tau'\) \hspace{10cm} 4, Inversion Lemma
6. \(\tau = \tau' \text{ Ref}\) \hspace{10cm} 4, Inversion Lemma
7. \(M : \Lambda\) \hspace{10cm} 1, 3, C-Inversion Lemma
8. \(s\) is only derivable by Rule TMem \hspace{10cm} Inspection of \(M : \Lambda\) rules
9. \(M = \{(\ell_1, \nu_1), \ldots, (\ell_n, \nu_n)\}\) \hspace{10cm} 7, 8, Inversion of Rule TMem
10. \(\Lambda = \{(\ell : \tau_1), \ldots, (\ell : \tau_n)\}\) \hspace{10cm} 7, 8, Inversion of Rule TMem
11. \(\nu_i \in \{1, \ldots, n\}, \Lambda, \bullet \vdash \nu_i : \tau_i\) \hspace{10cm} 7, 8, Inversion of Rule TMem
12. let \(M' = M \cup \{(\ell, v)\}\)
13. let \(\Lambda' = \Lambda \cup \{(\ell : \tau')\}\)
14. \(\Lambda \subseteq \Lambda'\) \hspace{10cm} Definition of \(\subseteq\)
15. \(\nu_i \in \{1, \ldots, n\}, \Lambda', \bullet \vdash \nu_i : \tau_i\)
16. \(\Lambda', \bullet \vdash v : \tau'\)
17. \(M' : \Lambda'\)
18. \(\Lambda', \bullet \vdash C' \text{ ok}\)
19. \(\Lambda', \bullet \vdash \ell : \tau\)

Case  \[(C, e_1) \rightarrow (C', e'_1)\]
\[(C, \text{!e}_1) \rightarrow (C', \text{!e}'_1)\] (derVal)

1. \(\Lambda \vdash C \text{ ok}\) \hspace{10cm} assumption
2. \(\Lambda, \bullet \vdash \text{!e}_1 : \tau\) \hspace{10cm} assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) \hspace{10cm} assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau \text{ Ref}\) \hspace{10cm} 2, Inversion Lemma
5. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \wedge \Lambda', \bullet \vdash e'_1 : \tau \text{ Ref} \wedge \Lambda \subseteq \Lambda')\)
6. \(\Lambda', \bullet \vdash \text{!e}'_1 : \tau\)

Case  \((M' \cup \{(\ell, v)\}, \ldots, !\ell) \rightarrow (M' \cup \{(\ell, v)\}, \ldots, v)\) (derVal)

1. \(\Lambda \vdash (M' \cup \{(\ell, v)\}, \ldots) \text{ ok}\) \hspace{10cm} assumption
2. \(\Lambda, \bullet \vdash \ell : \tau\) \hspace{10cm} assumption
3. \(\Lambda, \bullet \vdash \ell : \tau \text{ Ref}\) \hspace{10cm} 2, Inversion Lemma
4. \(s\) is only derivable with Rule location \hspace{10cm} Inspection of typing rules
5. \(\Lambda = \Lambda' \cup \{(\ell : \tau)\}\)
6. \(M' \cup \{(\ell, v)\} : \Lambda' \cup \{(\ell : \tau)\}\)
7. \(s\) is only derivable by Rule TMem \hspace{10cm} 3, 4, Inversion of Rule location
8. \(\Lambda, \bullet \vdash v : \tau\)

Result is from 14, 18, 19

Result is from 6, 7, Inversion of Rule TMem
Case \[
(C, e_1) \rightarrow (C, e'_1)
\] (assignE1)
\[
(C, e_1 : e_2) \rightarrow (C', e'_1 : e_2)
\]

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash e_1 : e_2 : \tau\)
3. \((C, e_1) \rightarrow (C', e'_1)\)
4. \(\Lambda, \bullet \vdash e_1 : \tau_i \text{ Ref}\)
5. \(\Lambda, \bullet \vdash e_2 : \tau_i\)
6. \(\tau = \text{ Unit}\)
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_1 : \tau_1 \text{ Ref} \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash e_2 : \tau_i\)
9. \(\Lambda', \bullet \vdash e'_1 : e_2 : \tau\)
   Result is from 7, 9

Case \[
(C, e_2) \rightarrow (C, e'_2)
\] (assignE2)
\[
(C, \ell_1 : e_2) \rightarrow (C', \ell_1 : e'_2)
\]

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash \ell_1 : e_2 : \tau\)
3. \((C, e_2) \rightarrow (C', e'_2)\)
4. \(\Lambda, \bullet \vdash \ell_1 : \tau_i \text{ Ref}\)
5. \(\Lambda, \bullet \vdash e_2 : \tau_i\)
6. \(\tau = \text{ Unit}\)
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash \ell_1 : \tau_i \text{ Ref}\)
9. \(\Lambda', \bullet \vdash \ell_1 : e'_2 : \tau\)
   Result is from 7, 9

Case \[
(M' \cup \{\ell, v\}, R, \text{ inOb, rt, out, } \tau_{\text{out}}), \ell := v'\) \rightarrow ((M' \cup \{\ell, v'\}, R, \text{ inOb, rt, out, } \tau_{\text{out}}), \text{ unit})
\] (assignValue)

1. \(\Lambda \vdash (M' \cup \{\ell, v\}, R, \text{ inOb, rt, out, } \tau_{\text{out}}) \text{ ok}\)
2. \(\Lambda, \bullet \vdash \ell := v' : \tau\)
3. \(M' \cup \{\ell, v\} : \Lambda\)
4. \(\Lambda \vdash R \text{ ok}\)
5. \(\Lambda, \bullet \vdash \text{ inOb} : \text{ Bool}\)
6. \(\Lambda, \bullet \vdash \text{ rt} : \text{ Ref} L_{\text{List}}\)
7. \(\Lambda, \bullet \vdash \text{ out} : \tau_{\text{out}} \text{ Event Option}\)
8. \(\Lambda, \bullet \vdash \ell : \tau' \text{ Ref}\)
9. \(\Lambda, \bullet \vdash v' := t'\)
10. \(\tau = \text{ Unit}\)
11. \(\Delta \text{ is only derivable by Rule TMem}\)
12. \(\Lambda = \Lambda' \cup \{\ell : \tau'\}\)
13. \(\forall i \in \{1, \ldots, n\}, \Lambda, \bullet \vdash v_i : \tau_i\)
14. \(\forall i \in \{1, \ldots, n - 1\}, \Lambda, \bullet \vdash v_i : \tau_i\)
15. \(M' \cup \{\ell, v'\} : \Lambda\)
16. \(\Lambda \vdash (M' \cup \{\ell, v'\}, R, \text{ inOb, rt, out, } \tau_{\text{out}}) \text{ ok}\)
17. \(\Lambda, \bullet \vdash \text{ unit} : \tau\)
   Result is from 16, 17

\text{Unit}
Case \[
\frac{(C, e_1) \rightarrow (C', e'_1)}{(C, e_1 :: e_2) \rightarrow (C', e'_1 :: e_2)} \quad \text{(listPrependE1)}
\]

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash e_1 :: e_2 : \tau\)
3. \((C, e_1) \rightarrow (C', e'_1)\)
4. \(\tau = \tau_{\text{List}}\)
5. \(\Lambda, \bullet \vdash e_1 : \tau_1\)
6. \(\Lambda, \bullet \vdash e_2 : \tau_{\text{List}}\)
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash e_2 : \tau_{\text{List}}\)
9. \(\Lambda', \bullet \vdash e'_2 :: e_2 : \tau\)

Result is from 7, 9

Case \[
\frac{(C, e_2) \rightarrow (C', e'_2)}{(C, v_1 :: e_2) \rightarrow (C', v'_1 :: e'_2)} \quad \text{(listPrependE2)}
\]

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash v_1 :: e_2 : \tau\)
3. \((C, e_2) \rightarrow (C', e'_2)\)
4. \(\tau = \tau_{\text{List}}\)
5. \(\Lambda, \bullet \vdash v_1 : \tau_1\)
6. \(\Lambda, \bullet \vdash e_2 : \tau_{\text{List}}\)
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_2 : \tau_{\text{List}} \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash e_2 : \tau_{\text{List}}\)
9. \(\Lambda', \bullet \vdash v'_1 :: e'_2 : \tau\)

Result is from 7, 8

Case \[
\frac{(C, e_1) \rightarrow (C', e'_1)}{(C, e_1 @ e_2) \rightarrow (C', e'_1 @ e_2)} \quad \text{(listAppendE1)}
\]

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash e_1 @ e_2 : \tau\)
3. \((C, e_1) \rightarrow (C', e'_1)\)
4. \(\Lambda, \bullet \vdash e_1 : \tau'_{\text{List}}\)
5. \(\Lambda, \bullet \vdash e_2 : \tau'_{\text{List}}\)
6. \(\tau = \tau'_{\text{List}}\)
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_1 : \tau'_{\text{List}} \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash e_2 : \tau'_{\text{List}}\)
9. \(\Lambda', \bullet \vdash e'_1 @ e_2 : \tau\)

Result is from 7, 8

Case \[
\frac{(C, e_2) \rightarrow (C', e'_2)}{(C, v_1 @ e_2) \rightarrow (C', v'_1 @ e'_2)} \quad \text{(listAppendE2)}
\]

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash v_1 @ e_2 : \tau\)
3. \((C, e_2) \rightarrow (C', e'_2)\)
4. \(\Lambda, \bullet \vdash v_1 : \tau'_{\text{List}}\)
5. \(\Lambda, \bullet \vdash e_2 : \tau'_{\text{List}}\)
6. \(\tau = \tau'_{\text{List}}\)
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_2 : \tau'_{\text{List}} \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash e_2 : \tau'_{\text{List}}\)
9. \(\Lambda', \bullet \vdash v'_1 @ e'_2 : \tau\)

Result is from 7, 9
Case \((\lambda, ([]:\tau'_\text{List}) \circ \varphi_3) \rightarrow (\lambda, \varphi_3)\) (appendNil)

1. \(\lambda \vdash C \text{ ok}\) assumption
2. \(\lambda \cdot \vdash ([]:\tau'_\text{List} \circ \varphi_3) : \tau\) assumption
3. \(\tau = \tau'_\text{List}\) 2, Inversion Lemma
4. \(\lambda \cdot \vdash \varphi_3 : \tau\) 2, 3, Inversion Lemma

Result is from 1, 4

Case \((\lambda, (\varphi_1 : \varphi_2) \circ \varphi_3) \rightarrow (\lambda, (\varphi_1 \circ \varphi_2 \circ \varphi_3))\) (appendCons)

1. \(\lambda \vdash C \text{ ok}\) assumption
2. \(\lambda \cdot \vdash (\varphi_1 : \varphi_2) \circ \varphi_3 : \tau\) assumption
3. \(\lambda \cdot \vdash (\varphi_1 : \varphi_2) : \tau'_\text{List}\) 2, Inversion Lemma
4. \(\lambda \cdot \vdash \varphi_3 : \tau'_\text{List}\) 2, Inversion Lemma
5. \(\tau = \tau'_\text{List}\) 2, Inversion Lemma
6. \(\lambda \cdot \vdash \varphi_1 : \tau'\) 3, Inversion Lemma
7. \(\lambda \cdot \vdash \varphi_2 : \tau'_\text{List}\) 3, Inversion Lemma
8. \(\lambda \cdot \vdash (\varphi_2 \circ \varphi_3) : \tau'_\text{List}\) 4, 7, Rule listAppend
9. \(\lambda \cdot \vdash (\varphi_2 \circ \varphi_3) : \tau\) 5, 6, 8, Rule listCons

Result is from 1, 9

Case \((\lambda, \text{head}(e_1)) \rightarrow (\lambda, \text{head}(e'_1))\) (listHeadE)

1. \(\lambda \vdash C \text{ ok}\) assumption
2. \(\lambda \cdot \vdash \text{head}(e_1) : \tau\) assumption
3. \((\lambda, e_1) \rightarrow (\lambda, e'_1)\) assumption
4. \(\lambda \cdot \vdash e_1 : \tau'_\text{List}\) 2, Inversion Lemma
5. \(\tau = \tau'\) Option
6. \(\exists \lambda' : (\lambda' \vdash C' \text{ ok} \land \lambda' \cdot \vdash e'_1 : \tau'_\text{List} \land \lambda \subseteq \lambda')\) 1, 3, 4, IH
7. \(\lambda' \cdot \vdash \text{head}(e'_1) : \tau\) 5, 6, Rule head

Result is from 6, 7

Case \((\lambda, \text{head}(\varphi_1 : \cdots : [\cdot] : \tau'_\text{List})) \rightarrow (\lambda, \text{in}\text{some}(\varphi_1) : \tau'\text{Option})\) (listHeadCons)

1. \(\lambda \vdash C \text{ ok}\) assumption
2. \(\lambda \cdot \vdash \text{head}(\varphi_1 : \cdots : [\cdot] : \tau'_\text{List}) : \tau'\text{Option}\) assumption
3. \(\lambda \cdot \vdash (\varphi_1 : \cdots : [\cdot] : \tau'_\text{List}) : \tau'_\text{List}\) 2, Inversion Lemma
4. \(\tau = \tau'\) Option
5. \(\lambda \cdot \vdash \varphi_1 : \tau'\) 3, Inversion Lemma
6. \(\lambda \cdot \vdash \text{in}\text{some}(\varphi_1) : \tau'\text{Option}\) 4, 5, Def of types, Rule variant

Result is from 1, 6

Case \((\lambda, \text{head}([\cdot] : \tau'_\text{List})) \rightarrow (\lambda, \text{in}\text{n}\text{one}(\text{unit}) : \tau'\text{Option})\) (listHeadNil)

1. \(\lambda \vdash C \text{ ok}\) assumption
2. \(\lambda \cdot \vdash \text{head}([\cdot] : \tau'_\text{List}) : \tau\) assumption
3. \(\tau = \tau'\) Option
4. \(\lambda \cdot \vdash \text{unit} : \text{Unit}\) Rule unitVal
5. \(\lambda \cdot \vdash \text{in}\text{n}\text{one}(\text{unit}) : \tau'\text{Option}\) 3, 4, Def of types, Rule variant

Result is from 1, 5
Case \[(C, e_1) \rightarrow (C', e'_1)\] \(\text{(listTailE)}\)  
1. \(\Lambda \vdash C\ ok\)\hspace{2cm} assumption  
2. \(\Lambda; \bullet \vdash \text{tail}(e_1) : \tau\)\hspace{2cm} assumption  
3. \((C, e_1) \rightarrow (C', e'_1)\)\hspace{2cm} assumption  
4. \(\Lambda; \bullet \vdash e_1 : \tau'_{\text{List}}\)\hspace{2cm} 2, Inversion Lemma  
5. \(\tau = \tau'_{\text{List}}\)\hspace{2cm} 2, Inversion Lemma  
6. \(\exists \Lambda' : (\Lambda' \vdash C'\ ok \land \Lambda' \vdash e'_1 : \tau'_{\text{List}} \land \Lambda \subseteq \Lambda')\)\hspace{2cm} 1, 3, 4, IH  
7. \(\Lambda' ; \bullet \vdash \text{tail}(e'_1) : \tau\)\hspace{2cm} 5, 6, Rule tail  
Result is from 6, 7  

Case \[(C, \text{tail}(v_1 :: v_2)) \rightarrow (C, v_2)\] \(\text{(listTailCons)}\)  
1. \(\Lambda \vdash C\ ok\)\hspace{2cm} assumption  
2. \(\Lambda; \bullet \vdash \text{tail}(v_1 :: v_2) : \tau\)\hspace{2cm} assumption  
3. \(\Lambda; \bullet \vdash (v_1 :: v_2) : \tau'_{\text{List}}\)\hspace{2cm} 2, Inversion Lemma  
4. \(\tau = \tau'_{\text{List}}\)\hspace{2cm} 2, Inversion Lemma  
5. \(\Lambda; \bullet \vdash v_2 : \tau\)\hspace{2cm} 3, 4, Inversion Lemma  
Result is from 1, 5  

Case \[(C, \text{tail}([] : \tau'_{\text{List}})) \rightarrow (C, []) : \tau'_{\text{List}}\)\] \(\text{(listTailNil)}\)  
1. \(\Lambda \vdash C\ ok\)\hspace{2cm} assumption  
2. \(\Lambda; \bullet \vdash \text{tail}([] : \tau'_{\text{List}}) : \tau\)\hspace{2cm} assumption  
3. \(\tau = \tau'_{\text{List}}\)\hspace{2cm} 2, Inversion Lemma  
4. \(\Lambda; \bullet \vdash ([]) : \tau'_{\text{List}} : \tau\)\hspace{2cm} 2, 3, Inversion Lemma  
Result is from 1, 4  

Case \[(C, e_1) \rightarrow (C', e'_1)\] \(\text{(listEmptyE)}\)  
1. \(\Lambda \vdash C\ ok\)\hspace{2cm} assumption  
2. \(\Lambda; \bullet \vdash \text{empty}(e_1) : \tau\)\hspace{2cm} assumption  
3. \((C, e_1) \rightarrow (C', e'_1)\)\hspace{2cm} assumption  
4. \(\Lambda; \bullet \vdash e_1 : \tau'_{\text{List}}\)\hspace{2cm} 2, Inversion Lemma  
5. \(\tau = \text{Bool}\)\hspace{2cm} 2, Inversion Lemma  
6. \(\exists \Lambda' : (\Lambda' \vdash C'\ ok \land \Lambda' \vdash e'_1 : \tau'_{\text{List}} \land \Lambda \subseteq \Lambda')\)\hspace{2cm} 1, 3, 4, IH  
7. \(\Lambda' ; \bullet \vdash \text{empty}(e'_1) : \tau\)\hspace{2cm} 5, 6, Rule empty  
Result is from 6, 7  

Case \[(C, \text{empty}([] : \tau'_{\text{List}})) \rightarrow (C, \text{true})\] \(\text{(listEmptyNil)}\)  
1. \(\Lambda \vdash C\ ok\)\hspace{2cm} assumption  
2. \(\Lambda; \bullet \vdash \text{empty}([] : \tau'_{\text{List}}) : \tau\)\hspace{2cm} assumption  
3. \(\tau = \text{Bool}\)\hspace{2cm} 2, Inversion Lemma  
4. \(\Lambda; \bullet \vdash \text{true} : \tau\)\hspace{2cm} 3, Rule boolVal  
Result is from 1, 4  

Case \[(C, \text{empty}(v_1 :: v_2)) \rightarrow (C, \text{false})\] \(\text{(listEmptyCons)}\)  
1. \(\Lambda \vdash C\ ok\)\hspace{2cm} assumption  
2. \(\Lambda; \bullet \vdash \text{empty}(v_1 :: v_2) : \tau\)\hspace{2cm} assumption  
3. \(\tau = \text{Bool}\)\hspace{2cm} 2, Inversion Lemma  
4. \(\Lambda; \bullet \vdash \text{false} : \tau\)\hspace{2cm} 3, Rule boolVal  
Result is from 1, 4
Case \( \forall j(1 \leq j \leq n), e_j = v_j \) \( \vdash (C, e_i) \rightarrow (C', e'_i) \) \( i \in \{1, \ldots, n\} \) (recordE)

1. \( \vdash C \text{ ok} \) assumption
2. \( \vdash \ell_i \rightarrow \tau \) assumption
3. \( (C, e_i) \rightarrow (C', e'_i) \) assumption
4. \( \vdash e_i : \tau \) \( \vdash e_n : \tau_n \) 2, Inversion Lemma
5. \( \tau = (\tau_1 \times \cdots \times \tau_n) \) 2, Inversion Lemma
6. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \Lambda' \vdash e'_i : \tau_i \wedge \Lambda \subseteq \Lambda') \) 1, 3, 4, IH
7. \( \Lambda' \vdash \ell_i \rightarrow \tau \) 4, 6, \( \Lambda \text{-weakening Lemma} \)
8. \( \Lambda' \vdash (\ell_i = e_1, \ldots, \ell_n = e_n : \tau) \) 5-7, Rule variant

Result is from 6, 8

---

Case \( (C, e_1) \rightarrow (C', e'_1) \) (projectionE)

1. \( \vdash C \text{ ok} \) assumption
2. \( \vdash e_1 : \tau \) assumption
3. \( (C, e_1) \rightarrow (C', e'_1) \) assumption
4. \( \vdash (\ell_1 : \ell_1 \times \cdots \times \ell_n : \tau_n) \) 2, Inversion Lemma
5. \( \tau = \tau_i \) 2, Inversion Lemma
6. \( i \in \{1, \ldots, n\} \) 2, Inversion Lemma
7. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \Lambda' \vdash e'_1 : (\ell_1 : \ell_1 \times \cdots \times \ell_n : \tau_n) \wedge \Lambda \subseteq \Lambda') \) 1, 3, 4, IH
8. \( \Lambda' \vdash e'_1 : \tau \) 5-7, Rule projection

Result is from 7, 8

---

Case \( (C, e_1) \rightarrow (C', e'_1) \) (projectionV)

1. \( \vdash C \text{ ok} \) assumption
2. \( \vdash (\ell_1 = v_1_1, \ldots, \ell_n = v_n), \ell_i : \tau \) assumption
3. \( (C, e_1) \rightarrow (C', e'_1) \) assumption
4. \( \tau = \ell_i \) 2, Inversion Lemma
5. \( \Delta \vdash \ell_i : \tau \) 2, Inversion Lemma
6. \( i \in \{1, \ldots, n\} \) 3, 4, Inversion Lemma

Result is from 1, 5

---

Case \( (C, e_1) \rightarrow (C', e'_1) \) (variantE)

1. \( \vdash C \text{ ok} \) assumption
2. \( \vdash (\in^n_{\ell_1} e_1 : \tau') : \tau \) assumption
3. \( (C, e_1) \rightarrow (C', e'_1) \) assumption
4. \( i \in \{1, \ldots, n\} \) 2, Inversion Lemma
5. \( \Delta \vdash e_i : \tau_i \) 2, Inversion Lemma
6. \( \tau = \ell_1 : \ell_1 \times \cdots \times \ell_n : \tau_n \) 2, Inversion Lemma
7. \( \tau' = \tau \) 2, Inversion Lemma
8. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \Lambda' \vdash e'_1 : \tau_1 \wedge \Lambda \subseteq \Lambda') \) 1, 3, 4, IH
9. \( \Lambda' \vdash (\in^n_{\ell_1} e'_1 : \tau') : \tau \) 5-8, Rule variant

Result is from 8, 9

---

Case \( (C, e_1) \rightarrow (C', e'_1) \) (caseE)

1. \( \vdash C \text{ ok} \) assumption
2. \( \vdash (\text{case } e_1 \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) \) assumption
3. \( (C, e_1) \rightarrow (C', e'_1) \) assumption
4. \( \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \) 2, Inversion Lemma
5. \( \vdash e_1 : \tau \) \( \vdash \Delta, x_1 : \tau_1 \vdash e_n : \tau \) 2, Inversion Lemma
6. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \Lambda' \vdash e'_1 : (\ell_1 : \ell_1 \times \cdots \times \ell_n : \tau_n) \wedge \Lambda \subseteq \Lambda') \) 1, 3, 4, IH
7. \( \Lambda' \vdash (\text{case } e'_1 \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau \) 5-6, \( \Lambda \text{-weakening Lemma} \)
8. \( \Lambda' \vdash (\text{case } e'_1 \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau \) 6, 7, Rule Case

Result is from 6, 8
\begin{align*}
\text{Case} \quad i \in \{1, \ldots, n\}
\end{align*}

\begin{align*}
(C, (\text{case} \ (\iota_{e_i}, v_i : \tau') \ if \ \ell_1 \ x_1 \Rightarrow e_1 \ | \ \cdots \ | \ \ell_n \ x_n \Rightarrow e_n) & \rightarrow (C, [v_i/x_i]e_i) \\
1. \Lambda \vdash C \ ok & \quad \text{assumption} \\
2. \Lambda \bullet \vdash (\text{case} \ (\iota_{e_i}, v_i : \tau') \ if \ \ell_1 \ x_1 \Rightarrow e_1 \ | \ \cdots \ | \ \ell_n \ x_n \Rightarrow e_n) : \tau & \quad \text{assumption} \\
3. \Lambda \bullet \vdash (\iota_{e_i}, v_i : \tau') : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) & \quad \text{2, Inversion Lemma} \\
4. \Lambda \bullet \vdash [v_i/x_i]e_i : \tau & \quad \text{2, Inversion Lemma} \\
5. \Lambda \bullet \vdash v_i : \tau_i & \quad \text{3, Inversion Lemma} \\
6. \Lambda \bullet \vdash [v_i/x_i]e_i : \tau & \quad \text{4, 5, Lemma substitution} \\
\text{Result is from 1, 6} \\
\end{align*}

\begin{align*}
\text{Case} \quad (C, e_1) & \rightarrow (C', e'_1) \\
(C, \text{makeTypedVal}(\tau_1, e_1)) & \rightarrow (C', \text{makeTypedVal}(\tau_1, e'_1)) \\
1. \Lambda \vdash C \ ok & \quad \text{assumption} \\
2. \Lambda \bullet \vdash \text{makeTypedVal}(\tau_1, e_1) : \tau & \quad \text{assumption} \\
3. (C, e_1) & \rightarrow (C', e'_1) & \quad \text{assumption} \\
4. \Lambda \bullet \vdash e_1 : \tau_1 & \quad \text{2, Inversion Lemma} \\
5. \tau = \text{TypedVal} & \quad \text{2, Inversion Lemma} \\
6. \exists \Lambda' : (\Lambda' \vdash C' \ ok) \land \Lambda' \bullet \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda' & \quad 1, 3, 4, \text{IH} \\
7. \Lambda' \bullet \vdash \text{makeTypedVal}(\tau_1, e'_1) : \tau & \quad 5, 6, \text{Rule makeTypedVal} \\
\text{Result is from 6, 7} \\
\end{align*}

\begin{align*}
\text{Case} \quad (C, e_1) & \rightarrow (C', e'_1) \\
(C, \text{tryCast}(\tau_1, e_1)) & \rightarrow (C', \text{tryCast}(\tau_1, e'_1)) \\
1. \Lambda \vdash C \ ok & \quad \text{assumption} \\
2. \Lambda \bullet \vdash \text{tryCast}(\tau_1, e_1) : \tau & \quad \text{assumption} \\
3. (C, e_1) & \rightarrow (C', e'_1) & \quad \text{assumption} \\
4. \Lambda \bullet \vdash e_1 : \text{TypedVal} & \quad \text{2, Inversion Lemma} \\
5. \tau = \tau_1 \ \text{Option} & \quad \text{2, Inversion Lemma} \\
6. \exists \Lambda' : (\Lambda' \vdash C' \ ok) \land \Lambda' \bullet \vdash e'_1 : \text{TypedVal} \land \Lambda \subseteq \Lambda' & \quad 1, 3, 4, \text{IH} \\
7. \Lambda' \bullet \vdash \text{tryCast}(\tau_1, e'_1) : \tau & \quad 5, 6, \text{Rule tryCast} \\
\text{Result is from 6, 7} \\
\end{align*}

\begin{align*}
\text{Case} \quad (C, \text{tryCast}(\tau', \text{makeTypedVal}(\tau', v))) & \rightarrow (C, \text{in} \ some(v) : \tau' \ \text{Option}) \\
1. \Lambda \vdash C \ ok & \quad \text{assumption} \\
2. \Lambda \bullet \vdash \text{tryCast}(\tau', \text{makeTypedVal}(\tau', v)) : \tau & \quad \text{assumption} \\
3. \Lambda \bullet \vdash \text{makeTypedVal}(\tau', v) : \text{TypedVal} & \quad \text{2, Inversion Lemma} \\
4. \tau = \tau' \ \text{Option} & \quad \text{2, Inversion Lemma} \\
5. \Lambda \bullet \vdash v : \tau' & \quad \text{3, Inversion Lemma} \\
6. \Lambda \bullet \vdash (\text{in} \ some(v) : \tau' \ \text{Option}) : \tau & \quad 4, 5, \text{Def of types, Rule variant} \\
\text{Result is from 1, 6} \\
\end{align*}

\begin{align*}
\text{Case} \quad (C, \text{tryCast}(\tau_1, \text{makeTypedVal}(\tau_2, v))) & \rightarrow (C, \text{in} \ none(unit) : \tau_1 \ \text{Option}) \\
1. \Lambda \vdash C \ ok & \quad \text{assumption} \\
2. \Lambda \bullet \vdash \text{tryCast}(\tau_1, \text{makeTypedVal}(\tau_2, v)) : \tau & \quad \text{assumption} \\
3. \tau = \tau_1 \ \text{Option} & \quad \text{2, Inversion Lemma} \\
4. \Lambda \bullet \vdash \text{unit} : \text{Unit} & \quad \text{Rule unitVal} \\
5. \Lambda \bullet \vdash (\text{in} \ none(unit) : \tau_1 \ \text{Option}) : \tau & \quad 3, 4, \text{Def of types, Rule variant} \\
\text{Result is from 1, 5} \\
\end{align*}
Case \((C, e_1) \rightarrow (C', e'_1)\) (endLabelE)

1. \(\Lambda \vdash C\ ok\) assumption
2. \(\Lambda, \bullet \vdash \{ e_1 \}_{s(v_2)} : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau\) 2, Inversion Lemma
5. \(\exists \Lambda' : (\Lambda' + C\ ok \land \Lambda', \bullet \vdash e'_1 : \tau \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
6. \(\Lambda', \bullet \vdash \{ e'_1 \}_{s(v_2)} : \tau\) 5, Rule endLabel

Result is from 5, 6

Case \((C, \{ v_1 \}_{s(v_2)}) \rightarrow (C, v_1)\) (endLabelE)

1. \(\Lambda \vdash C\ ok\) assumption
2. \(\Lambda, \bullet \vdash \{ v_1 \}_{s(v_2)} : \tau\) assumption
3. \(\Lambda, \bullet \vdash v_1 : \tau\) 2, Inversion Lemma

Result is from 1, 3

Case \((\{ M, R, inOb, rt, out, \tau_{out} \}, \{ v_1 \}_{s(v_2)}) \rightarrow (\{ M, R, false, rt, out, \tau_{out} \}, v_1)\) (endLabelE)

1. \(\Lambda \vdash (\{ M, R, inOb, rt, out, \tau_{out} \})\ ok\) assumption
2. \(\Lambda, \bullet \vdash \{ v_1 \}_{s(v_2)} : \tau\) assumption
3. \(\Lambda, \bullet \vdash v_1 : \tau\) 2, Inversion Lemma
4. \(M : \Lambda\) 1, C-Inversion Lemma
5. \(\Lambda \vdash R\ ok\) 1, C-Inversion Lemma
6. \(\Lambda, \bullet \vdash rt : ResList\) 1, C-Inversion Lemma
7. \(\Lambda, \bullet \vdash out : \tau_{out} Event Option\) 1, C-Inversion Lemma
8. \(\Lambda, \bullet \vdash false : \text{Bool}\) Rule boolVal
9. \(\Lambda \vdash (\{ M, R, false, rt, out, \tau_{out} \})\ ok\) 4-8, Rule C-ok

Result is from 3, 9

Case \((\ldots, rt, out, \tau_{out}), \text{getRT}()\) (getRT)

1. \(\Lambda \vdash (\ldots, rt, out, \tau_{out})\ ok\) assumption
2. \(\Lambda, \bullet \vdash \text{getRT}() : \tau\) assumption
3. \(\tau = ResList\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash rt : \tau\) 1, 3, C-Inversion Lemma

Result is from 1, 4

Case \((C, e_1) \rightarrow (C', e'_1)\) (makeCFG)

1. \(\Lambda \vdash C\ ok\) assumption
2. \(\Lambda, \bullet \vdash \text{makeCFG}(e_1) : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\tau = CFG\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash e_1 : Obligation\) 2, Inversion Lemma
6. \(\exists \Lambda' : (\Lambda' + C'\ ok \land \Lambda', \bullet \vdash e'_1 : Obligation \land \Lambda \subseteq \Lambda')\) 1, 3, 5, IH
7. \(\Lambda', \bullet \vdash \text{makeCFG}(e'_1) : \tau\) 4, 6, Rule makeCFG

Result is from 6, 7
\[ g = \text{makeCFG}_1(v) \]

\[
(C, \text{makeCFG}(v)) \xrightarrow{\text{begin makeCFG}(v)} (C, [g]_{\text{makeCFG}(v)})
\]

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda, \mathcal{\bullet} \vdash \text{makeCFG}(v) : \tau \)
3. \( g = \text{makeCFG}_1(v) \)
4. \( \Lambda, \mathcal{\bullet} \vdash g : \text{CFG} \)
5. \( \tau = \text{CFG} \)
6. \( \Lambda, \mathcal{\bullet} \vdash \{g\}_{\text{makeCFG}(v)} : \tau \)

Result is from 1, 6

\[
(C, e_1) \xrightarrow{\text{setOutput}(e_1)} (C', \text{setOutput}(e'_1))
\]

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda, \mathcal{\bullet} \vdash \text{setOutput}(e_1) : \tau \)
3. \( (C, e_1) \xrightarrow{\tau} (C', e'_1) \)
4. \( \Lambda, \mathcal{\bullet} \vdash e_1 : \text{Event} \)
5. \( \tau = \text{Bool} \)
6. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_1 : \text{Event} \land \Lambda \subseteq \Lambda') \)
7. \( \Lambda', \mathcal{\bullet} \vdash \text{setOutput}(e'_1) : \tau \)

Result is from 6, 7

\[
\text{out} = \text{in}_{\text{some}}(e) : \tau \text{ Event Option}
\]

1. \( \Lambda \vdash (\ldots, \text{out}, \text{true}) \text{ ok} \)
2. \( \Lambda, \mathcal{\bullet} \vdash \text{setOutput}(e) : \tau \)
3. \( \tau = \text{Bool} \)
4. \( \Lambda, \mathcal{\bullet} \vdash \text{true} : \tau \)

Result is from 1, 4

\[
([M, R, \text{inOb}, \text{rt}, \text{in}_{\text{none}}(\text{unit}) : \tau_{\text{out}} \text{ Event Option}, \tau_{\text{out}}), \text{setOutput}(\text{in}_{\text{act}}(v) : \text{Event})) \xrightarrow{\text{setOutput}} ([M, R, \text{inOb}, \text{rt}, \text{in}_{\text{some}}(\text{in}_{\text{act}}(v) : \tau_{\text{out}} \text{ Event Option}), \tau_{\text{out}}), \text{true})
\]

1. \( \Lambda \vdash (M, R, \text{inOb}, \text{rt}, \text{in}_{\text{none}}(\text{unit}) : \tau_{\text{out}} \text{ Event Option}, \tau_{\text{out}}) \text{ ok} \)
2. \( \Lambda, \mathcal{\bullet} \vdash \text{setOutput}(\text{in}_{\text{act}}(v) : \text{Event}) : \tau \)
3. \( \tau = \text{Bool} \)
4. \( \Lambda, \mathcal{\bullet} \vdash v : \text{Act} \)
5. \( M : \Lambda \)
6. \( A \vdash R \text{ ok} \)
7. \( \Lambda, \mathcal{\bullet} \vdash \text{inOb} : \text{Bool} \)
8. \( \Lambda, \mathcal{\bullet} \vdash \text{rt} : \text{ResList} \)
9. \( \Lambda, \mathcal{\bullet} \vdash (\text{in}_{\text{act}}(v) : \tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event} \)
10. \( \Lambda, \mathcal{\bullet} \vdash (\text{in}_{\text{some}}(\text{in}_{\text{act}}(v) : \tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event Option}) : \tau_{\text{out}} \text{ Event Option} \)
11. \( \Lambda, \mathcal{\bullet} \vdash \text{true} : \tau \)

Result is from 13, 14
Case:  
\[(M, R, \text{inOb}, r, \text{in}_\text{none}(\text{unit})): \tau_{\text{out}} \text{ Event Option}, \tau_{\text{out}}), \text{setOutput}(\text{in}_\text{res}(\text{res}(v_1, \text{makeTypedVal}(\tau_{\text{out}}, v_2))): \text{Event})\]  
\[
\rightarrow ((M, R, \text{inOb}, r, \text{in}_\text{some}(\text{in}_\text{res}(\text{res}(v_1, v_2))): \tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event Option}, \tau_{\text{out}}), \text{true})
\]

1. \(\Lambda \vdash (M, R, \text{inOb}, r, \text{in}_\text{none}(\text{unit})): \tau_{\text{out}} \text{ Event Option}, \tau_{\text{out}}) \text{ ok} \) assumption
2. \(\Lambda, \bullet \vdash \text{setOutput}(\text{in}_\text{res}(\text{res}(v_1, \text{makeTypedVal}(\tau_{\text{out}}, v_2))): \text{Event}) : \tau\) assumption
3. \(\Lambda, \bullet \vdash (\text{in}_\text{res}(\text{res}(v_1, \text{makeTypedVal}(\tau_{\text{out}}, v_2))): \text{Event}) : \tau\) 2, Inversion Lemma
4. \(\tau = \text{Bool}\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash \text{res}(v_1, \text{makeTypedVal}(\tau_{\text{out}}, v_2)) : \text{Res}\) 3, Inversion Lemma
6. \(\Lambda, \bullet \vdash v_1 : \text{Act}\) 5, Inversion Lemma
7. \(\Lambda, \bullet \vdash \text{makeTypedVal}(\tau_{\text{out}}, v_2) : \text{TypedVal}\) 7, Inversion Lemma
8. \(\Lambda, \bullet \vdash v_2 : \tau_{\text{out}}\) 1, C-Inversion Lemma
9. \(M : \Lambda\) 1, C-Inversion Lemma
10. \(\Lambda \vdash R \text{ ok}\) 6, 8, Rule variant, Def of types
11. \(\Lambda, \bullet \vdash \text{inOb} : \text{Bool}\) 1, C-Inversion Lemma
12. \(\Lambda, \bullet \vdash r : \text{ResList}\) 1, C-Inversion Lemma
13. \(\Lambda, \bullet \vdash \text{res}(v_1, v_2) : \tau_{\text{out}} \text{ Res}\) 6, 8, Rule variant, Def of types
14. \(\Lambda, \bullet \vdash (\text{in}_\text{res}(\text{res}(v_1, v_2))): \tau_{\text{out}} \text{ Event}\) 13, Rule variant, Def of types
15. \(\Lambda, \bullet \vdash (\text{in}_\text{res}(\text{res}(v_1, v_2))): \tau_{\text{out}} \text{ Event Option}\) 14, Rule variant, Def of types
16. \(\text{let out} = (\text{in}_\text{some}(\text{in}_\text{res}(\text{res}(v_1, v_2))): \tau_{\text{out}} \text{ Event Option})\) assumption
17. \(\Lambda \vdash (M, R, \text{inOb}, r, \tau_{\text{out}}) \text{ ok}\) 9-12, 15, 16, Rule C-ok
18. \(\Lambda, \bullet \vdash \tau : \text{true}\) 4, Rule boolVal

\[\tau' \neq \tau_{\text{out}} \rightarrow \text{out} = \text{in}_\text{none}(\text{unit}) : \tau_{\text{out}} \text{ Event Option}\]  
\[
((\ldots, \tau_{\text{out}}), \text{setOutput}(\text{in}_\text{res}(\text{res}(v_1, \text{makeTypedVal}(\tau', v_2))): \text{Event})\) \rightarrow  
\[
((\ldots, \tau_{\text{out}}), \text{false})
\]

1. \(\Lambda \vdash (\ldots, \tau_{\text{out}}) \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{setOutput}(\text{in}_\text{res}(\text{res}(v_1, \text{makeTypedVal}(\tau', v_2))): \text{Event}) : \tau\) assumption
3. \(\tau = \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{false} : \tau\) 3, Rule boolVal

\[\text{Result is from 1, 4}\]

Case:  
\[(\ldots, \tau_{\text{out}}), \text{outputNotSet}() \rightarrow ((\ldots, \tau_{\text{out}}), \text{true})\]  
\[
\text{outputNotSetTrue}
\]

1. \(\Lambda \vdash (\ldots, \tau_{\text{out}}) \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{outputNotSet}() : \tau\) assumption
3. \(\tau = \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{false} : \tau\) 3, Rule boolVal

\[\text{Result is from 1, 4}\]

Case:  
\[(\ldots, \tau_{\text{out}}), \text{outputNotSet}() \rightarrow ((\ldots, \tau_{\text{out}}), \text{false})\]  
\[
\text{outputNotSetFalse}
\]

1. \(\Lambda \vdash (\ldots, \tau_{\text{out}}) \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{outputNotSet}() : \tau\) assumption
3. \(\tau = \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{false} : \tau\) 3, Rule boolVal

\[\text{Result is from 1, 4}\]

Case:  
\[(\ldots, \tau_{\text{out}}), \text{getOutput}() \rightarrow ((\ldots, \tau_{\text{out}}), \text{in}_\text{some}(\text{in}_\text{act}(\tau)): \tau_{\text{out}} \text{ Event Option})\]  
\[
\text{getOutputSomeAct}
\]

1. \(\Lambda \vdash (\ldots, \tau_{\text{out}}) \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{getOutput}() : \tau\) assumption
3. \(\text{out} = \text{in}_\text{some}(\text{in}_\text{act}(\tau)): \tau_{\text{out}} \text{ Event Option}\) assumption
4. \(\tau = \text{Event Option}\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash (\text{in}_\text{act}(\tau)): \tau_{\text{out}} \text{ Event Option})\) 1, 3, C-Inversion Lemma
6. \(\Lambda, \bullet \vdash (\text{in}_\text{act}(\tau)): \tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event}\) 5, Inversion Lemma
7. \(\Lambda, \bullet \vdash v : \text{Act}\) 6, Inversion Lemma
8. \(\Lambda, \bullet \vdash (\text{in}_\text{act}(\tau)): \text{Event}) : \tau\) 7, Rule variant, Def of types
9. \(\Lambda, \bullet \vdash (\text{in}_\text{some}(\text{in}_\text{act}(\tau)): \text{Event}) : \tau_{\text{out}} \text{ Event Option}) : \tau\) 4, 8, Rule variant, Def of types

\[\text{Result is from 1, 9}\]
Case \[ \text{ out } = \text{ in}_{\text{some}}(\text{in}_{\text{res}}(\text{res}(v_1, v_2)) : \text{out} \text{ Event}) : \text{out} \text{ Event Option} \]

\[
\Rightarrow ((\ldots, \text{out}, \text{out} \text{ Option}), \text{getOutput}()) \quad \Rightarrow \quad ((\ldots, \text{out}, \text{out} \text{ Option}), \text{in}_{\text{some}}(\text{in}_{\text{res}}(\text{res}(v_1, \text{makeTypedVal}(\text{out}, v_2))): \text{Event}) : \text{Event Option})
\]

1. \( \Lambda \vdash (\ldots, \text{out}, \text{out} \text{ Option}) \text{ ok} \) \quad \text{assumption}
2. \( \Lambda \vdash \text{getOutput}() : \tau \) \quad \text{assumption}
3. \( \text{out} = \text{in}_{\text{some}}(\text{in}_{\text{res}}(\text{res}(v_1, v_2)) : \text{out} \text{ Event}) : \text{out} \text{ Event Option} \) \quad \text{assumption}
4. \( \tau = \text{Event Option} \) \quad \text{2, Inversion Lemma}
5. \( \Lambda \vdash (\text{in}_{\text{res}}(\text{res}(v_1, v_2)) : \text{out} \text{ Event}) : \text{out} \text{ Event Option} \) \quad \text{1, 3, Inversion Lemma}
6. \( \Lambda \vdash \text{res}(v_1, v_2) : \text{out} \text{ Res} \) \quad \text{6, Inversion Lemma}
7. \( \Lambda \vdash v_1 : \text{Act} \) \quad \text{7, Inversion Lemma}
8. \( \Lambda \vdash v_2 : \tau \) \quad \text{7, Inversion Lemma}
9. \( \Lambda \vdash \text{makeTypedVal}(\text{out}, v_2) : \text{TypedVal} \) \quad \text{9, Rule makeTypedVal}
10. \( \Lambda \vdash \text{res}(v_1, \text{makeTypedVal}(\text{out}, v_2)) : \text{Res} \) \quad \text{8, 10, Rule variant, Def of types}
11. \( \Lambda \vdash (\text{in}_{\text{res}}(\text{res}(v_1, \text{makeTypedVal}(\text{out}, v_2))): \text{Event}) : \) \quad \text{11, Rule variant, Def of types}
12. \( \Lambda \vdash (\text{in}_{\text{res}}(\text{res}(v_1, \text{makeTypedVal}(\text{out}, v_2))): \text{Event}) : \) \quad \text{12, Rule variant, Def of types}

Result is from 1, 13

Case \[ \text{out} = \text{in}_{\text{none}}(\text{unit}) : \text{out} \text{ Event Option} \]

\[
\Rightarrow ((\ldots, \text{out}, \text{out} \text{ Option}), \text{getOutput}()) \quad \Rightarrow \quad ((\ldots, \text{out}, \text{out} \text{ Option}), \text{in}_{\text{none}}(\text{unit}) : \text{Event Option})
\]

1. \( \Lambda \vdash (\ldots, \text{out}, \text{out} \text{ Option}) \text{ ok} \) \quad \text{assumption}
2. \( \Lambda \vdash \text{getOutput}() : \tau \) \quad \text{assumption}
3. \( \tau = \text{Event Option} \) \quad \text{2, Inversion Lemma}
4. \( \Lambda \vdash \text{unit} : \text{Unit} \) \quad \text{Rule unitVal}
5. \( \Lambda \vdash (\text{in}_{\text{none}}(\text{unit}) : \text{Event Option}) : \tau \quad 3, 4, \text{Rule variant, Def of types}

Result is from 1, 5

Case \[ \text{(C, e)} \quad \Rightarrow \quad \text{(C', e') } \]

\[
\Rightarrow (\text{C, monitor}(\tau_1, e)) \quad \Rightarrow \quad (\text{C', monitor}(\tau_1, e'))
\]

1. \( \Lambda \vdash \text{C ok} \) \quad \text{assumption}
2. \( \Lambda \vdash \text{monitor}(\tau_1, e) : \tau \) \quad \text{assumption}
3. \( \text{(C, e)} \quad \Rightarrow \quad \text{(C', e') } \quad \text{assumption}
4. \( \Lambda \vdash e_1 : \quad \text{(evt : } \tau_1 \text{ Event } \times \text{polls : PolList } \times \text{OS } \times \text{vc } : \text{VC} ) \quad 2, \text{Inversion Lemma}
5. \( \tau = \tau_1 \text{ Event} \quad 2, \text{Inversion Lemma}
6. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \land \quad \Lambda', e_1' : (\text{evt : } \tau_1 \text{ Event } \times \text{polls : PolList } \times \text{OS } \times \text{vc } : \text{VC} ) \land \quad \Lambda \subseteq \Lambda') \quad 1, 3, 4, \text{IH}
7. \( \Lambda', e_1' : \tau \) \quad 5, 6, \text{Rule monitor}

Result is from 6, 7

Case \[ ((R, \text{inOb}, rt, \text{out}, \text{out} \text{ Option}), \text{monitor}(\tau_1, v)) \quad \text{beginmonitor(v)} \quad \Rightarrow \quad ((R, \text{true}, rt, \text{out}, \text{out} \text{ Option}), \text{monitor}(\tau_1, v)) \]

1. \( \Lambda \vdash (R, \text{inOb}, rt, \text{out}, \text{out} \text{ Option}) \text{ ok} \) \quad \text{assumption}
2. \( \Lambda \vdash \text{monitor}(\tau_1, v) : \tau \) \quad \text{assumption}
3. \( \Lambda \vdash v : \quad \text{(evt : } \tau_1 \text{ Event } \times \text{polls : PolList } \times \text{OS } \times \text{vc } : \text{VC} ) \quad 2, \text{Inversion Lemma}
4. \( \tau = \tau_1 \text{ Event} \quad 2, \text{Inversion Lemma}
5. \( M : \Lambda \) \quad 1, \text{C-Inversion Lemma}
6. \( \Lambda \vdash R \text{ ok} \) \quad 1, \text{C-Inversion Lemma}
7. \( \Lambda \vdash \text{rt : ResList} \) \quad 1, \text{C-Inversion Lemma}
8. \( \Lambda \vdash \text{out : } \text{out} \text{ Event Option} \) \quad 1, \text{C-Inversion Lemma}
9. \( \Lambda \vdash \text{true : Bool} \) \quad \text{Rule boolVal}
10. \( \Lambda \vdash (R, \text{true}, rt, \text{out}, \text{out} \text{ Option}) \text{ ok} \) \quad 5, 9, Rule C-ok
11. \( \Lambda \vdash \text{monitor}(\tau_1, v) : \tau_1 \text{ Event} \) \quad 3, Monitor Type Lemma
12. \( \Lambda \vdash \text{monitor}(\tau_1, v) : \tau_1 \text{ Event} \) \quad 11, Rule endLabel

Result is from 10, 12
\[ (C, e_1) \rightarrow (C', e'_1) \] 

\[ (\text{invoke}(e_1, e_2)) \rightarrow (C', \text{invok}(e'_1, e_2)) \] 

(\text{invoke}\text{1})

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda, \bullet \vdash \text{invoke}(e_1, e_2) : \tau \)
3. \( (C, e_1) \rightarrow (C', e'_1) \)
4. \( \Lambda, \bullet \vdash e_1 : \text{String} \)
5. \( \Lambda, \bullet \vdash e_2 : \text{TypedVal} \)
6. \( \tau = \text{TypedVal Option} \)
7. \( \exists \Lambda' : (\Lambda' \vdash C \text{ ok} \land \Lambda' \vdash e'_1 : \text{String} \land \Lambda \subseteq \Lambda') \)
8. \( \Lambda', \bullet \vdash e_2 : \text{TypedVal} \)
9. \( \Lambda', \bullet \vdash \text{invoke}(e'_1, e_2) : \tau \)

Result is from 7, 9

\[ (C, e_2) \rightarrow (C', e'_2) \] 

\[ (\text{invoke}(s_1, e_2)) \rightarrow (C', \text{invoke}(s_1, e'_2)) \] 

(\text{invoke}\text{2})

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda, \bullet \vdash \text{invoke}(s_1, e_2) : \tau \)
3. \( (C, e_2) \rightarrow (C', e'_2) \)
4. \( \Lambda, \bullet \vdash e_2 : \text{TypedVal} \)
5. \( \tau = \text{TypedVal Option} \)
6. \( \exists \Lambda' : (\Lambda' \vdash C \text{ ok} \land \Lambda' \vdash e'_2 : \text{TypedVal} \land \Lambda \subseteq \Lambda') \)
7. \( \Lambda', \bullet \vdash s_1 : \text{String} \)
8. \( \Lambda', \bullet \vdash \text{invoke}(s_1, e'_2) : \tau \)

Result is from 6, 8

\[ \forall (s', f) \in F, s_1 \neq s' \] 

\[ (\text{invoke}(s_1, v_2)) \rightarrow ((M, (F, \ldots, \ldots), \text{in}_{\text{none}}(\text{unit}) : \text{TypedVal Option})) \] 

(\text{invokeValNotExists})

1. \( \Lambda \vdash (M, (F, \ldots, \ldots), \text{ok}) \)
2. \( \Lambda, \bullet \vdash \text{invoke}(s_1, v_2) : \tau \)
3. \( \tau = \text{TypedVal Option} \)
4. \( \Lambda, \bullet \vdash \text{in}_{\text{none}}(\text{unit}) : \text{Unit} \)
5. \( \Lambda, \bullet \vdash \text{in}_{\text{none}}(\text{unit}) : \text{TypedVal Option} : \tau \)

Result is from 3, 4

\[ (s_1, \text{fun} x_1(x_2 : \tau_1 : \tau_2 = e_1) \in F \rightarrow \text{v}_2 = \text{makeTypedVal}(\tau_2, \text{v}_2') \] 

\[ ((M, (F, \ldots, \ldots), \text{in}_{\text{none}}(\text{makeTypedVal}(\text{call} \text{fun} x_1(x_2 : \tau_1 : \tau_2 = e_1, v_2')))) : \text{TypedVal Option}) \] 

(\text{invokeValueExistsOK})

1. \( \Lambda \vdash (M, (F, \ldots, \ldots), \text{ok}) \)
2. \( \Lambda, \bullet \vdash \text{invoke}(s_1, v_2) : \tau \)
3. \( \text{v}_2 = \text{makeTypedVal}(\tau_1, v_2') \)
4. \( \Lambda, \bullet \vdash v_2 : \text{TypedVal} \)
5. \( \tau = \text{TypedVal Option} \)
6. \( \Lambda, \bullet \vdash v_2' : \tau_1 \)
7. \( \Lambda \vdash (F, \ldots, \ldots) \)
8. \( \Lambda \vdash F \text{ ok} \)
9. \( (s_1, \text{fun} x_1(x_2 : \tau_1 : \tau_2 = e_1) \in F \) 
10. \( \) 
11. \( \) 
12. \( \) 
13. \( \) 
14. \( \) 
15. \( \) 
16. \( \) 
17. \( \) 

Result is from 1, 17
\( \text{Case} \quad ((s_1, \text{fun} \; x_1(x_2 : t_1) : t_2 = e_1) \in F \quad v_2 = \text{makeTypedVal}(r_3, v'_2) \quad r_1 \neq t_3 \) \) \\
\( \quad (M, (F, \ldots, \ldots), \text{invoke}(s_1, v_2)) \quad \rightarrow \quad (M, (F, \ldots, \ldots), \text{in} \_\text{none}(\text{unit}) : \text{TypedVal Option}) \) \\
\( \begin{align*}
1. \quad & \lambda \vdash (M, (F, \ldots, \ldots), \text{ok}) \quad \text{assumption} \\
2. \quad & \lambda \cdot \vdash \text{call}(e_1, e_2) : \tau \quad \text{assumption} \\
3. \quad & (C, e_1) \rightarrow (C', \text{call}(e'_1, e_2)) \quad \text{(call1)} \\
4. \quad & \lambda \cdot \vdash e_1 : t_1 \rightarrow t_2 \quad 2, \text{Inversion Lemma} \\
5. \quad & \lambda \cdot \vdash e_2 : t_1 \quad 2, \text{Inversion Lemma} \\
6. \quad & \tau = t_2 \quad 2, \text{Inversion Lemma} \\
7. \quad & \exists \Lambda' : (\lambda \vdash C' \text{ok} \land \lambda' \cdot \vdash e'_1 : t_1 \rightarrow t_2 \land \lambda \subseteq \lambda') \quad 1, 3, 4, \text{IH} \\
8. \quad & \lambda' \cdot \vdash e_2 : t_1 \quad 5, 7, \text{A-weakening Lemma} \\
9. \quad & \lambda' \cdot \vdash \text{call}(e'_1, e_2) : \tau \quad 6-8 \text{ Rule call} \\
\end{align*} \)

\( \text{Case} \quad ((C, e_2) \rightarrow (C', e'_2)) \quad \rightarrow \quad (C', \text{call}(f, e'_2)) \) \\
\( \text{(call2)} \)

\( \begin{align*}
1. \quad & \lambda \vdash C \text{ ok} \quad \text{assumption} \\
2. \quad & \lambda \cdot \vdash \text{call}(f, e_2) : \tau \quad \text{assumption} \\
3. \quad & (C, e_2) \rightarrow (C', e'_2) \quad \text{assumption} \\
4. \quad & \lambda \cdot \vdash f : t_1 \rightarrow t_2 \quad 2, \text{Inversion Lemma} \\
5. \quad & \lambda \cdot \vdash e_2 : t_1 \quad 2, \text{Inversion Lemma} \\
6. \quad & \tau = t_2 \quad 2, \text{Inversion Lemma} \\
7. \quad & \exists \Lambda' : (\lambda \vdash C' \text{ok} \land \lambda' \cdot \vdash e'_2 : t_1 \land \lambda \subseteq \lambda') \quad 1, 3, 5, \text{IH} \\
8. \quad & \lambda' \cdot \vdash e_2 : t_1 \quad 4, 7, \text{A-weakening Lemma} \\
9. \quad & \lambda' \cdot \vdash \text{call}(f, e'_2) : \tau \quad 6-8, \text{Rule Call} \\
\end{align*} \)

\( f \notin \text{range}(F) \quad \forall \text{pol} \in \text{pols} (f \neq \text{pol.onTrigger} \land f \neq \text{pol.onObligation}) \quad f = \text{fun} \; x_1(x_2 : t_1) : t_2 = e_1 \) \\
\( \begin{align*}
\text{Case} \quad ((M, (F, \text{pols}, \ldots, \ldots), \text{call}(f, v)) \quad \rightarrow \quad (M, (F, \text{pols}, \ldots, \ldots), \{(v/x_2, f/x_1)\}_{f(v)})) \) \\
\( \text{(callNonMonitoredFunction)} \)

\( \begin{align*}
1. \quad & \lambda \vdash (M, (F, \ldots, \ldots), \text{ok}) \quad \text{assumption} \\
2. \quad & \lambda \cdot \vdash \text{call}(f, v) : \tau \quad \text{assumption} \\
3. \quad & f = (\text{fun} \; x_1(x_2 : t_1) : t_2 = e_1) \quad \text{assumption} \\
4. \quad & \lambda \cdot \vdash f : t'_1 \rightarrow t'_2 \quad 2, \text{Inversion Lemma} \\
5. \quad & \lambda \cdot \vdash v : t'_1 \quad 2, \text{Inversion Lemma} \\
6. \quad & \tau = t'_2 \quad 2, \text{Inversion Lemma} \\
7. \quad & \lambda \cdot x_1 : t'_1 \rightarrow t'_2, x_2 : t_1' \vdash e_1 : t_2' \quad 3, 4, \text{Inversion Lemma} \\
8. \quad & \lambda \cdot \vdash [v/x_2, f/x_1]e_1 : \tau \quad 3-7, \text{Substitution Lemma} \\
9. \quad & \lambda \cdot \vdash [(v/x_2, f/x_1)e_1]_{f(v)} : \tau \quad 8, \text{Rule endLabel} \\
\end{align*} \)

Result is from 1, 9
(name = s, onTrigger = f₁, onObligation = f₂, vote = f₃) ∈ pols \ f₁ = (\text{fun } x₁(x₂ : \text{Event}) : \text{Unit} = e₁)

Case
\begin{align*}
((M, (F, \text{pol}, os, vc), \text{in}\text{Ob}, rt, out, \tau_{out}), \text{call}(f₁, v))
\end{align*}

\begin{align*}
\begin{tabular}{ll}
1. & A \vdash (M, (F, \text{pol}, os, vc), \text{in}\text{Ob}, \tau_{out}, \text{call}(f₁, v)) \text{ assumption} \\
2. & A \cdot \vdash \text{call}(f₁, v) : \tau \text{ assumption} \\
3. & f₁ = (\text{fun } x₁(x₂ : \text{Event}) : \text{Unit} = e₁) \\
4. & A \cdot \vdash f₁ : \tau₁ → τ₂ \\
5. & A \cdot \vdash v : τ₁ \\
6. & \tau = τ₂ \\
7. & A \cdot \vdash \{f₁/x₁, v/x₂\}[e₁ : \tau] \\
8. & \begin{tabular}{l}
M : \text{Unit}
\end{tabular} \\
9. & A \vdash (F, \text{pol}, os, vc) \text{ ok} \\
10. & A \cdot \vdash \text{in Ob} : \text{Bool} \\
11. & A \cdot \vdash \text{out} : \tau_{out} \text{ Event Option} \\
12. & A \cdot \vdash \{\} : \text{ResList} \\
13. & A \cdot \vdash \{\} : \text{ResList} \\
14. & A \vdash (M, (F, \text{pol}, os, vc), \text{in}\text{Ob}, \tau_{out}, \text{call}(f₂, v)) \text{ Rule listEmptyVal} \\
15. & A \cdot \vdash \{\} : \text{ResList} \text{ Rule listEmptyVal} \\
\end{tabular}
\end{align*}

Result is from 14, 15

\begin{align*}
\begin{tabular}{ll}
(name = s, onTrigger = f₁, onObligation = f₂, vote = f₃) ∈ pols \ f₂ = (\text{fun } x₁(x₂ : \text{ResList}) : \text{Unit} = e₁)
\end{tabular}
\end{align*}

Case
\begin{align*}
\begin{tabular}{ll}
((M, (F, \text{pol}, os, vc), \text{in}\text{Ob}, rt, out, \tau_{out}), \text{call}(f₂, v))
\end{tabular}
\end{align*}

\begin{align*}
\begin{tabular}{ll}
1. & A \vdash (M, (F, \text{pol}, os, vc), \text{in}\text{Ob}, \tau_{out}, \text{call}(f₂, v)) \text{ assumption} \\
2. & A \cdot \vdash \text{call}(f₂, v) : \tau \text{ assumption} \\
3. & f₂ = (\text{fun } x₁(x₂ : \text{Event}) : \text{Unit} = e₁) \\
4. & A \cdot \vdash f₂ : \tau₁ → τ₂ \\
5. & A \cdot \vdash v : τ₁ \\
6. & \tau = τ₂ \\
7. & A \cdot \vdash \{f₂/x₁, v/x₂\}[e₁ : \tau] \\
8. & \begin{tabular}{l}
M : \text{Unit}
\end{tabular} \\
9. & A \vdash (F, \text{pol}, os, vc) \text{ ok} \\
10. & A \cdot \vdash \text{in Ob} : \text{Bool} \\
11. & A \cdot \vdash \text{out} : \tau_{out} \text{ Event Option} \\
12. & A \cdot \vdash \{\} : \text{ResList} \\
13. & A \cdot \vdash \{\} : \text{ResList} \\
14. & A \vdash (M, (F, \text{pol}, os, vc), \text{in}\text{Ob}, \tau_{out}, \text{call}(f₂, v)) \text{ Rule listEmptyVal} \\
15. & A \cdot \vdash \{\} : \text{ResList} \text{ Rule listEmptyVal} \\
\end{tabular}
\end{align*}

Result is from 14, 15

\begin{align*}
\begin{tabular}{ll}
s, f = (\text{fun } x₁(x₂ : \tau₁) : \tau₂ = e₁)
\end{tabular}
\end{align*}

Case
\begin{align*}
\begin{tabular}{ll}
((M, (F, \text{pol}, os, vc), \text{true}, rt, out, \tau_{out}), \text{call}(f, v))
\end{tabular}
\end{align*}

\begin{align*}
\begin{tabular}{ll}
res(\text{act}(s, \text{makeTypedVal}(\tau₁, v)), \text{makeTypedVal}(\tau₂, \{f/x₁, v/x₂\}[e₁ : \tau₁] : \text{ResList}, \text{out}, \tau_{out}), \{\} : \text{ResList})
\end{tabular}
\end{align*}
1. \( \Lambda \vdash (M, (F, \text{pol}, \text{os}, \text{ve}), \text{true}, \text{rt}, \text{out}, \tau_{\text{out}}) \text{ ok} \)
2. \( \Lambda. \bullet \vdash \text{call}(f, v) : \tau \)
3. \( f = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1) \)
4. \( \Lambda. \bullet \vdash f : \tau_1^1 \rightarrow \tau_2^1 \)
5. \( \Lambda. \bullet \vdash v : \tau_1^1 \)
6. \( \tau = \tau_{\text{out}}^2 \)
7. 4 is only derivable by Rule fun
8. \( \tau_1 = \tau_1^1 \land \tau_2 = \tau_2^1 \)
9. \( \Lambda. \vdash [x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1] \vdash e_1 : \tau_2 \)
10. \( \Lambda. \bullet \vdash [f/x_1, v/x_2]e_1 : \tau_2 \)
11. \( \Lambda. \bullet \vdash \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1) : \text{TypedVal} \)
12. \( \Lambda. \bullet \vdash \text{makeTypedVal}(\tau_1, v) : \text{TypedVal} \)
13. \( \Lambda. \bullet \vdash s : \text{String} \)
14. \( \Lambda. \bullet \vdash \text{act}(s, \text{makeTypedVal}(\tau_1, v)) : \text{Act} \)
15. \( \Lambda. \bullet \vdash \text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v)), \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1)) : \text{Res} \)
16. \( \text{Let } r = \text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v)), \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1)) \)
17. \( \Lambda. \bullet \vdash ([] : \text{ResList}) : \text{ResList} \)
18. \( \Lambda. \bullet \vdash (r : []) : \text{ResList} : \text{ResList} \)
19. \( \Lambda. \bullet \vdash \text{rt} : \text{ResList} \)
20. \( \Lambda. \bullet \vdash \text{rt} @ (r : []) : \text{ResList} : \text{ResList} \)
21. \( M : \Lambda \)
22. \( \Lambda \vdash (F, \text{pol}, \text{os}, \text{ve}) \text{ ok} \)
23. \( \Lambda. \bullet \vdash \text{true} : \text{Bool} \)
24. \( \Lambda. \bullet \vdash \text{out} : \tau_{\text{out}} \text{ Event Option} \)
25. \( \Lambda \vdash (M, (F, \text{pol}, \text{os}, \text{ve}), \text{true}, \text{rt} @ (r : []) : \text{ResList}, \text{out}, \tau_{\text{out}}) \text{ ok} \)
26. \( \Lambda. \bullet \vdash ([f/x_1, v/x_2]e_1)x_1(v) : \tau \)

Result is from 25, 26

Assumption
2, Inversion Lemma
2, Inversion Lemma
2, Inversion Lemma
Inspection of typing rules
3, 4, 7, Inversion of Rule fun
3, 4, 6, Inversion Lemma
4, 5, 9, Substitution Lemma
10, Rule makeTypedVal
5, Rule makeTypedVal
Rule stringVal
12, 13, Rule product, Def of types
11, 14, Rule product, Def of types
Assumption
Rule listEmptyVal
15-17, Rule listCons
1, C-Inversion Lemma
18, 19, Rule listAppend
1, C-Inversion Lemma
1, C-Inversion Lemma
Rule boolVal
1, C-Inversion Lemma
20-24, Rule C-ok
6, 8, 10, Rule endLabel
\[
(s, f) \in F \quad f = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1) \\
\to \quad \begin{aligned}
(M, (F, \text{pols, os, } \text{vc}, \text{false, } r_{t, \text{out}}, \tau_{old}), \text{call}(f, v)) \\
(M, (F, \text{pols, os, } \text{vc}, \text{false, } r_{t, \text{in}}(\text{none(unit)}) : \tau_2 \text{ Event Option}, \tau_2), \epsilon_{\text{procEvt}})
\end{aligned}
\]

Where \(\epsilon_{\text{procEvt}} = \)

\[
\begin{aligned}
&\text{let aux = (fun aux : \tau_2 \text{ Event) : } \tau_2 \text{ Res = case event of}} \\
&\quad \text{act a } \Rightarrow \text{case invoke(a.name, a.arg) of}} \\
&\quad \text{some } r_1 \Rightarrow \text{case tryCast}(r_2, r_1) \text{ of}} \\
&\quad \text{some } v_1 \Rightarrow \text{let action_output = } \text{in} \text{res } \text{res}(a, v_1) : \tau_2 \text{ Event in}} \\
&\quad \text{let mon_output = monitor}(r_2, (\text{event } = \text{action_output}, \text{pols} = \text{pols}, \text{os} = \text{os}, \text{vc} = \text{vc})) \text{ in}} \\
&\quad \text{call}(\text{aux, mon_output}) \\
&\quad \text{end} \\
&\quad \text{end} \\
&\quad | \text{none } u_1 \Rightarrow \text{call}(\text{aux, event}) \\
&\quad | \text{none } u_2 \Rightarrow \text{call}(\text{aux, event}) \\
&\quad | \text{res } t_2 \Rightarrow r_2 \\
&\quad \text{in} \\
&\quad \text{call}(\text{aux, inact } \text{act}(s, v) : \tau_2 \text{ Event).result end}\_\_\_1(v)
\end{aligned}
\]
APPENDIX F  DEFINITION OF $\infty$-LANGUAGES

To express program traces that are used in theorems in Appendix G, this appendix defines the notion of an $\infty$-language.

**Definition:** Let $\Sigma$ be an alphabet, then $\Sigma^\infty$ denotes the set of all $\infty$-languages over $\Sigma$. A set $L \subseteq \Sigma^\infty$ is an $\infty$-language over $\Sigma$, and $L$ satisfies the following rules:

1. If a language $L$ is regular, then $L$ is an $\infty$-language.
2. If an $\omega$-language $L$ is $\omega$-regular, then $L$ is an $\infty$-language.
3. If $L_1$, $L_2$ are $\infty$-languages, then $L_1 L_2$ is an $\infty$-language, and $L_1 L_2 = \{xy \mid x \text{ is finite, } y \in L_1 \times L_2\}$

As Rules 1 and 2 show, the set of $\infty$-languages is the union of regular languages and $\omega$-regular languages, while Rule 3 defines concatenation such that the set of $\infty$-languages is closed under this operation. Defining concatenation this way allows for an infinitely-long string to be concatenated with another string (note that concatenating a finite string with an infinite string is well-defined in $\omega$-regular languages).

With this additional rule, a left-hand concatenation operand $L^\infty$ could be either a finite repetition of $L$ followed by the rest of the expression, or an infinite repetition of $L$. Such an operand captures the notion of a divergent program: it expresses a loop that either iterates a finite number of times and cedes control to the continuation, or iterates an infinite number of times and never
while producing traces matching expression $t$. Theorem, and pages 87 to 88 present the proof for the Obligation-Permutability Theorem. This appendix proves four important properties of PoCo obligations. Specifically, based on the twelve lemmas (pages 79 to 82), pages 87 and 88 present the proof of the Obligation-Reaction Theorem, pages 86 through 87 present the proof of the Pre-Obligation Completeness Theorem, and pages 84 through 86 present the proof for the Obligation-Permutability Theorem.

**APPENDIX G PROOF OF OBLIGATION PROPERTIES**

This appendix proves four important properties of PoCo obligations. Specifically, based on the twelve lemmas (pages 79 to 82), page 82 and 83 present the proof of the Atomic-Obligation and Conflict-Resolution Theorems respectively, pages 84 through 86 presents the proof of the Obligation-Reaction Theorem, pages 86 through 87 present the proof of the Pre-Obligation Completeness Theorem, and pages 87 to 88 present the proof for the Obligation-Permutability Theorem.

**Lemma 12 (Sequence Traces).** Given two well-typed expressions $e_1$ and $e_2$: if $e_1 \rightarrow^* v_1$ and $e_2 \rightarrow^* v_2$ while producing traces $t_1$ and $t_2$ respectively, then $e_1; e_2 \rightarrow^* v_1; v_2$ while producing trace $t = [t_1, t_2]$.

**Proof.**

1. $e_2$ cannot be reduced until $e_1 \rightarrow^* v_1$ (Rule sequence $E_2$)
2. after $e_1 \rightarrow^* v_1$ producing trace $t_1, t = t_1$ (1, sequence $E_1$)
3. values are always added to end of trace (Definition of $\Rightarrow$)
4. after $e_2 \rightarrow^* v_2$ producing trace $t_2, t = [t_1, t_2]$ (2, 3, Rule sequence $E_2$)

**Lemma 13 (Case-statement Traces).** Given a well-typed expression $e$ where $e = (case e_x of \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n)$: if $e_x \rightarrow^* in_{i_x} v_i$ while producing traces $t_x$ and $[v_i/x_i]e_i \rightarrow^* v'_i$ while producing traces $t_i$, then $e \rightarrow^* v'_i$ while producing trace $t$ and $t = [t_x, t_i] (i \in \{1, \ldots, n\})$.

**Proof.**

1. $e = (case e_x of \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n)$ (assumption)
2. $e_x$ must be reduced before $e_1, \ldots, e_n$ are reduced (Rule CaseE, Rule caseV)
3. $e_x \rightarrow^* in_{i_x} v_i$ while producing traces $t_x$ (assumption)
4. after $e_x \rightarrow^* in_{i_x} v_i$ producing trace $t_x, t = t_x$ (2, 3, Rule CaseE)
5. $e_x \rightarrow^* in_{i_x} v_i, e \rightarrow^* [v_i/x_i]e_i$ (1, 2, Rule caseV)
6. $[v_i/x_i]e_i \rightarrow^* v'_i$ while producing traces $t_i$ (assumption)
7. values are always added to end of trace (Definition of $\Rightarrow$)
8. after $[v_i/x_i]e_i \rightarrow^* v'_i$ producing traces $t_i, t = [t_x, t_i] (i \in \{1, \ldots, n\})$ (4, 6, 7)

**Lemma 14 (While-statement Traces).** Given a well-typed expression $e$ where $e = \mathit{while} (e_1)(e_2)$, if $e_1 \rightarrow^* v_1$ and $e_2 \rightarrow^* v_2$ while producing traces matching expression $t_1$ and $t_2$ respectively, then $e \rightarrow^* e'$ while producing trace $t$ that matches $\infty$-language expression $(t_1, t_2)^* t_1$.  

\[
\begin{align*}
ax^\infty b \cdot y^\infty c & = a(x^\infty|x^\infty)b(y^\infty|y^\infty)c \quad \text{[definition of } L^\infty] \\
& = (ax^\infty|ax^\infty)b(y^\infty|y^\infty)c \quad \text{[distribution]} \\
& = (ax^\infty b|ax^\infty b)(y^\infty|y^\infty)c \quad \text{[distribution]} \\
& = (ax^\infty(ax^\infty b)(y^\infty|y^\infty)c \\
& = (ax^\infty y^\infty|ax^\infty y^\infty|ax^\infty y^\infty|ax^\infty y^\infty|ax^\infty y^\infty) \\
& = (ax^\infty|ax^\infty|ax^\infty|ax^\infty|ax^\infty) \quad \text{[distribution]} \\
& = (ax^\infty|ax^\infty|ax^\infty|ax^\infty) \quad \text{[distribution]} \\
& = (ax^\infty|ax^\infty|ax^\infty) \quad \text{[idempotent]} \\
& = ax^\infty|ax^\infty|ax^\infty \quad \text{[distribution]} \\
& = ax^\infty|ax^\infty|ax^\infty \quad \text{[concatenation]}
\end{align*}
\]
Proof.
1. $e = \text{while } (e_1) | e_2$
2. $e \to \text{if } e_1 \text{ then } (e_2; \text{while } (e_1) | e_2) \text{ else unit}$
3. $e_1 \to^* v_1 \text{ producing trace } t_1$
4. assume $v_1 = false$ while producing trace $t_1$ in the first iteration
5. $e \to^* \text{unit} \text{ while producing trace } t_1$
6. assume $v_1 = true$ while producing trace $t_1$ in the first iteration
7. $e \to^* (e_2; \text{while } (e_1) | e_2) \text{ while producing trace } t_1$
8. $e_2 \to^* v_2 \text{ producing trace } t_2$
9. after $e_2 \to^* v_2$, the produced trace $t = [t_1, t_2]$
10. the produced trace will be either $t_1$ or $[t_1, t_2]$ after the first iteration
11. both $t_1$ and $[t_1, t_2]$ matches $\infty$-regular expression $(t_1, t_2)^\omega t_1$
12. the trace produced after first iteration matches $\infty$-regular expression $(t_1, t_2)^\omega t_1$
13. assume that $e$ has run $n(n > 1)$ iterations and the trace $t_n$ produced after the $n^{th}$ iteration matches $\infty$-regular expression $(t_1, t_2)^\omega t_1$

Definition of $label: \beta$

Lemma 15 (Let-statement Traces). Given a well-typed expression $e$ where $e = (let x = e_1 in e_2 end)$: if $e_1 \to^* v_1$ while producing a trace $t_1$ and $[v_2/x]e_2 \to^* v$ while producing a trace $t_2$, then $e \to^* v$ while producing a trace $t$ where $t = [t_1, t_2]$.

Proof.
1. $e_1$ must be reduced before $e_2$ is reduced Rule letE, Rule letValue
2. after $e_1 \to^* v_1 \text{ producing trace } t_1$, $t = t_1$
3. values are always added to end of trace Definition of $label: \beta$
4. after $[v_1/x]e_2 \to^* v$, $t = [t_1, t_2]$ 2, 3

Result from 12, 17

Lemma 16 (If-statement Traces). Given a well-typed expression $e$ where $e = if e_1 \text{ then } e_1 \text{ else } e_2$: if $e_1 \to^* v_1, e_2 \to^* v_2$, and $e_1 \to^* v_3$ while producing traces $t_1, t_2$ and $t_3$ respectively, then either $e \to^* v_1$ while producing trace $[t_3, t_1]$ when $v_3 = true$ or $e \to^* v_2$ while producing trace $[t_3, t_2]$ when $v_3 = false$.

Proof.
1. $e_1$ must be reduced before $e_1$ or $e_2$ are reduced Rule ifE, Rule ifTrue, Rule ifFalse
2. after $e_1 \to^* v_3 \text{ producing trace } t_3, t = t_3$
3. if $v_3 = false$, if $v_3$ then $e_1 \text{ else } e_2 \to^* e_2$ Rule ifFalse
4. values are always added to end of trace Definition of $label: \beta$
5. if $v_3 = false$, after if $v_3$ then $e_1$ else $e_2 \to^* v_2$, $t = [t_3, t_2]$ 2, 3, 4
6. if $v_3 = true$, if $v_3$ then $e_1$ else $e_2 \to^* e_1$ Rule ifTrue
7. if $v_3 = true$, after if $v_3$ then $e_1$ else $e_2 \to^* v_1$, $t = [t_3, t_1]$ 2, 6, 4

Result from 5, 7

Lemma 17 (Traverse-List-statement Traces). Given a well-typed expression $e = \text{while } (\neg\text{empty}(!\ell)) | \text{item ::= head(\ell)}; \ell ::= \text{tail(\ell); e}_2 |$ where $\ell : r_1 \text{ list } \text{ ref } f$: if $e_2 \to^* v_2$ while producing a trace $t_2$ and $e_2$ does not add or remove values from the list value stored at $\ell$, then the trace $t$ produced when $e \to^* v$ matches regular expression $t_2^N$ where $N$ is the length of $\ell$.

Proof.
1. $e_2 \to^* v_2$ while producing a trace $t_2$
2. $e_2$ does not add or remove values from the list value stored at $\ell$
3. the produced trace as the expression $\neg\text{empty}(!\ell)$ reduces to $v$ if $\ell$ is $\emptyset$
4. $e \to^* v$ while producing trace $t$ that matches $(\ell_1) t_2^\omega[t]$
5. $t$ matches $t_2^\omega$
6. $|\text{tail(!}\ell)| = |\ell| - 1$ Rule listTailValue, Rule listTailEmptyValue
7. $\ell ::= \text{tail(\ell)}$ reduce $\ell$ by 1 6, Rule AssignE2, Rule assignValue
Lemma 18 (CallOnTrigger-statement Traces). Given an expression \( e_1 = \text{call}(\text{onTrigger}, e) \): if \( \Lambda \cdot \triangleright e_1 \text{ ok} \), and \( e_1 \rightarrow^* v_1 \) while producing a trace \( t \), then \( t \) matches the \( \omega \)-expression \( \text{beginOb}(e) (\neg \text{ob}(e))^{\omega} \text{endOb}(e) \).

Proof.
1. \( \text{beginOb}(e) \) can not occur directly from \( e_{\text{onTrigger}} \)
2. \( \text{endOb}(e) \) can not occur directly from \( e_{\text{onTrigger}} \)
3. \( \text{call}(\text{onTrigger}, e) \rightarrow^* [v/x]_{\text{onTrigger}} \text{ onTrigger}(e) \)
   while producing trace \( \text{begin}_{\text{onTrigger}}(e) \)
4. the trace \( t_1 \) produced while \( [v/x]_{\text{onTrigger}} \rightarrow^* v' \) matched \( (\neg \text{ob}(e))^{\omega} \)
5. values are always added to end of trace
6. after \( [v/x]_{\text{onTrigger}} \text{ onTrigger}(e) \rightarrow^* [v']_{\text{onTrigger}}(e) \).
   \( t = [\text{begin}_{\text{onTrigger}}(e), t_1] \)
7. \( [v']_{\text{onTrigger}} \rightarrow^* v_1 \) while producing trace \( \text{end}_{\text{onTrigger}}(e) \)
8. after \( [v']_{\text{onTrigger}} \rightarrow^* v_1 \), the produced trace
   \( t = [\text{begin}_{\text{onTrigger}}(e), t_1, \text{end}_{\text{onTrigger}}(e)] \)
9. the trace \( t \) produced while \( e_1 \rightarrow^* v_1 \) matches the \( \omega \)-expression
   \( \text{beginOb}(e) (\neg \text{ob}(e))^{\omega} \text{endOb}(e) \)

Lemma 19 (CallOnObligation-statement Traces). Given an expression \( e_1 = \text{call}(\text{onObligation}, e) \): if \( \Lambda \cdot \triangleright e_1 \text{ ok} \), and \( e_1 \rightarrow^* v_1 \) while producing a trace \( t \), then \( t \) matches the \( \omega \)-expression \( \text{beginOb}(e) (\neg \text{ob}(e))^{\omega} \text{endOb}(e) \).

Proof.
1. \( \text{beginOb}(e) \) can not occur directly from \( e_{\text{onObligation}} \)
2. \( \text{endOb}(e) \) can not occur directly from \( e_{\text{onObligation}} \)
3. \( \text{call}(\text{onObligation}, e) \rightarrow^* [v/x]_{\text{onObligation}} \text{ onObligation}(e) \)
   while producing trace \( \text{begin}_{\text{onObligation}}(e) \)
4. the trace \( t_1 \) produced while \( [v/x]_{\text{onObligation}} \rightarrow^* v' \) matched \( (\neg \text{ob}(e))^{\omega} \)
5. values are always added to end of trace
6. after \( [v/x]_{\text{onObligation}} \text{ onObligation}(e) \rightarrow^* [v']_{\text{onObligation}}(e) \).
   \( t = [\text{begin}_{\text{onObligation}}(e), t_1] \)
7. \( [v']_{\text{onObligation}} \rightarrow^* v_1 \) while producing trace \( \text{end}_{\text{onObligation}}(e) \)
8. after \( [v']_{\text{onObligation}} \rightarrow^* v_1 \), the produced trace
   \( t = [\text{begin}_{\text{onObligation}}(e), t_1, \text{end}_{\text{onObligation}}(e)] \)
9. the trace \( t \) produced while \( e_1 \rightarrow^* v_1 \) matches the \( \omega \)-expression
   \( \text{beginOb}(e) (\neg \text{ob}(e))^{\omega} \text{endOb}(e) \)

Lemma 20 (No-Monitored-function Traces). Given a program \( p \) which is an untrusted application \( e_{\text{app}} \) with enforced policies, if \( \Lambda \cdot \triangleright p \text{ ok} \), and \( e_{\text{app}} \) does not contain any monitored functions and \( p \rightarrow^* p' \) while producing a trace \( t \), then \( t \) matches the \( \omega \)-expression \( (\neg \text{beginOb}(e))^{\omega} \).

Proof.
1. \( \text{beginOb}(e) \) cannot occur directly from \( e_{\text{app}} \)
2. \( \text{call}(\text{monitor}, v) \) cannot occur directly from \( e_{\text{app}} \)
3. \( \text{call}(\text{monitor}, v) \) happens only when a monitored function is called
4. \( \text{beginOb}(e) \) cannot exist in \( t \)
5. \( t \) matched \( (\neg \text{beginOb}(e))^{\omega} \)

Lemma 21 (App-no-append-result Traces). Given a program \( p \) which is an untrusted application \( e_{\text{app}} \) with enforced policies, if \( \Lambda \cdot \triangleright p \text{ ok} \), and \( e_{\text{app}} \) does not contain any monitored functions, and \( p \rightarrow^* p' \) while producing a trace \( t \), then \( t \) matches the \( \omega \)-expression \( (\neg \text{beginAppendRes}(e))^{\omega} \).
Proof.

1. Rule callFromObligation is the only rule that adds $begin_{appendRes}$
2. no function calls from $e_{app}$ can trigger rule callFromObligation
3. $begin_{appendRes}$ cannot exist in $t$
4. $t$ matched $\langle\neg begin_{appendRes}\rangle\infty$ inspection of the dynamic semantics

Lemma 22 (Policy-no-append-result Traces).
Given a program $p$ which is an untrusted application $e_{app}$ with enforced policies, if $\Lambda \vDash p \mathit{ok}$, and $p$'s obligations do not contain any monitored functions, and $p \rightarrow^* p'$ while producing a trace $t$, then $t$ matches the $\infty$-expression $\langle\neg begin_{appendRes}\rangle\infty$.

Proof.

1. Rule callFromObligation is the only rule that adds $begin_{appendRes}$
2. no function calls from $e_{app}$ can trigger rule callFromObligation
3. no non-function calls outside of $e_{app}$ can trigger rule callFromObligation
4. $begin_{appendRes}$ cannot exist in $t$
5. $t$ matched $\langle\neg begin_{appendRes}\rangle\infty$

Lemma 23 (No-makeCFG Traces).
Given a program $p$ which is an untrusted application $e_{app}$ with enforced policies, if $\Lambda \vDash p \mathit{ok}$, and $e_{app}$ does not contain any monitored functions, and $p \rightarrow^* p'$ while producing a trace $t$, then $t$ matches the $\infty$-expression $\langle\neg makeCFG\rangle\infty$.

Proof.

1. $\infty$-expression $\langle\neg makeCFG \langle v_1, v_2 \rangle \rangle\infty$
2. $\infty$-expression $\langle\neg makeCFG \langle v_1, v_2 \rangle \rangle\infty$
3. $\infty$-expression $\langle\neg makeCFG \langle v_1, v_2 \rangle \rangle\infty$
4. $\infty$-expression $\langle\neg makeCFG \langle v_1, v_2 \rangle \rangle\infty$

Theorem 9 (Atomic Obligation).
For all $p$, $t$, and $t'$, if $\Lambda \vDash p \mathit{ok}$, and $p \rightarrow^* p'$, and $t$ matches the $\infty$-expression $\langle\infty\rangle\begin{align*}beginOb(e_n) \quad & t' \begin{align*}beginOb(e_m) \quad & \infty\end{align*}\end{align*}$ then $t'$ matches the $\infty$-expression $\langle\infty\rangle\begin{align*}endOb(e_n) \quad & \infty\end{align*}$.

Proof.

Case 1: $e_{app}$ does not contain any monitored functions:

1. $t$ matches $\langle\neg beginOb\rangle\infty$
2. The theorem holds vacuously in this case because $t$ does not match $\langle\infty\rangle\begin{align*}beginOb(e_n) \quad & t' \begin{align*}beginOb(e_m) \quad & \infty\end{align*}\end{align*}$ Lemma 9, Rules of $\infty$-expression

Case 2: $e_{app}$ contains at least one monitored functions:

1. $beginOb(e)$ cannot occur directly from $e_{app}$ Definition of $\cdots \vdash p \mathit{ok}$
2. $call\langle\langle \mathit{monitor}, v \rangle\rangle$ cannot occur directly from $e_{app}$ Definition of $\cdots \vdash p \mathit{ok}$
3. $onTrigger \notin F$ and $onObligation \notin F$ Definition of function scope for $F$
4. only lines 24 and 25 of monitor can result in a trace including $beginOb(e)$ 3, Definition of monitor, Rule callNonMonitoredFunction
5. the trace produced while $call\langle\langle \mathit{call}, \mathit{onTrigg}, \mathit{arg}, \mathit{call} \rangle\rangle$ on line 24 reducing to $v_{call}$ is $[beginOb(e), \langle\neg \mathit{ob}\rangle\infty, endOb(e)]$ Line 7, Definition of monitor
6. the trace produced while $call\langle\langle \mathit{call}, \mathit{onOblig}, 2, \mathit{rt} \rangle\rangle$ on line 25 reducing to $v_{call}$ is $[beginOb(e), \langle\neg \mathit{ob}\rangle\infty, endOb(e)]$ Line 7, Definition of monitor
7. containing of the two call expressions, the case expression spans from line 24 to 25 does not introduce additional $beginOb(e)$ in the overall trace Line 2, Definition of monitor
8. the trace $t_1$ produced while case expression that spans from line 24 to 25 reducing to $v_{call}$ is $[beginOb(e), \langle\neg \mathit{ob}\rangle\infty, endOb(e)]$ 5, 6, 7, Line 2, Rules of $\infty$-expression
9. taken the case expression that spans from line 24 to 25 as its else branch, the if-then-else expression spans from line 23 to 26 does not introduce Line 5, Definition of monitor
10. the trace $t_2$ produced while the if-then-else expression spans from line 23 to 26 reducing to $v_{call}$ is either $\langle\neg beginOb(e)\rangle\infty$ or $\langle\neg beginOb(e)\rangle\infty, t_1$ Line 8, 9, Line 5, Rule of $\infty$-expression
Theorem 10 (Conflict Resolution). For all programs $P$ such that $\Lambda, \bullet \vdash P \text{ ok}$ and $P \xrightarrow{\ast} P'$, $t$ matches the $\infty$-expression $\langle \text{beginOb(e)} \rangle^{\infty}$ over $\langle \text{true}(e) \rangle^{\infty}$ where $\text{true}(e) := (\text{beginVote}(e) \langle \text{beginOb(e)} \rangle^{\infty})$ and $\text{endVote}(e) := \langle \text{true}(e) \rangle^{\infty}$.

\[ \text{begin}_v(e) := \text{end}_v(e) := \text{true}. \]

Case 1: $e_{app}$ does not contain any monitored functions calls:
1. $t$ matches $\langle \text{beginOb(e)} \rangle^{\infty}$
2. $t$ matches $\langle \text{beginOb(e)} \rangle^{\infty} \langle \text{true}(e) \rangle^{\infty} \langle \text{beginOb(e)} \rangle^{\infty}$

Case 2: $e_{app}$ contains at least one monitored functions call:
1. $\text{beginOb(e)} \text{ cannot occur directly from } e_{app}$
2. $\text{call(monitor, v)} \text{ cannot occur directly from } e_{app}$
3. $\text{onTrigger} \notin F \text{ and } \text{onObliation} \notin F$
4. only lines 24 and 25 of $t$ can result in a trace including $\text{beginOb(e)}$
5. the trace $t_{24}$ produced while $\text{call(v), onTrigger, onEvent}$ on line 24 reducing to $\text{true}$ is $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$
6. the trace $t_{25}$ produced while $\text{call(v), onObliation, onReturn}$ on line 25 reducing to $\text{true}$ matches $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$
7. containing of the two call expressions, the case expression spans from line 24 to 25 does not introduce additional $\text{beginOb(e)}$
8. the trace $t_1$ produced while the case expression reducing to $v_1$ is $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$
9. the case expression that spans from line 24 to 25 as its then branch, the if-then-else expression spans from line 23 to 26 does not introduce additional $\text{beginOb(e)}$
10. the trace $t_2$ produced while the condition expression $\text{call(c, v), true, v}$ on line 23 reducing to $v_2$ is $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$
11. the trace $t_3$ produced while the if-then-else expression spanned from line 23 to 26 reducing to $v_3$ is either $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$ or $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$
12. the trace $t_4$ produced while the expression $\text{call(p, policy, vote, ob)}$ on line 21 reducing to $v_4$ is $\langle \text{beginOb(e)} \rangle^{\infty}, \langle \text{endOb(e)} \rangle^{\infty}$

Proof:

\[ \square \]
13. the trace \( \tau_5 \) produced while the while expression spans from line 30 to 34 reducing to \( \nu_9 \) is \([\text{begin}\_\text{true}(\epsilon), \text{end}\_\text{true}(\epsilon); v_N]^N\).

14. the trace \( \tau_6 \) produced while the expressions spans from line 30 to 26 reducing to \( \nu_6 \) is \([\text{begin}\_\text{true}(v_1; \cdots; v_n), \text{end}\_\text{true}(v_1; \cdots; v_n)\text{false}_1, \text{true}_1]\).

15. the trace \( \tau_7 \) produced while the while expression spans from line 5 to 37 reducing to \( \nu_7 \) is \([\text{begin}\_\text{true}(\epsilon), \text{end}\_\text{true}(\epsilon); v_N]^N\).

16. the trace \( t \) produced when the overall monitor function reducing to \( v \) \([\text{begin}\_\text{true}(\epsilon), \text{end}\_\text{true}(\epsilon); v_N]^N\).

17. \( t \) can be simplified to \([\text{begin}\_\text{true}(\epsilon), \text{end}\_\text{true}(\epsilon); v_N]^N\).

18. all possible traces produced by \( p \) match the infinite-expression \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))]\).

\[\square\]

**Theorem 11 (Obligation Reaction Part 1).** For all programs \( p \) such that \( \Lambda, \bullet \vdash p \text{ ok} \) and \( p \xrightarrow{t} p' \), \( t \) matches the infinite-expression \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))\text{end}\_\text{true}(\epsilon); v_N]^N\).

**Proof.**

**Case 1:** \( e_{\text{app}} \) does not contain any monitored functions calls:

1. \( t \) matches \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))]\).

2. \( t \) matches \( e \).

**Case 2:** no obligations contain any monitored functions calls:

1. \( t \) matches \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))]\).

2. \( t \) matches \( e \).

**Case 3:** \( e_{\text{app}} \) contains at least one monitored functions call:

1. \( \text{begin}\_\text{true}(\epsilon) \) only can occur with the context of executions of obligations.

2. only lines 24 and 25 of monitor can result in a trace including \( \text{begin}\_\text{true}(\epsilon) \).

3. the trace produced while calling \( \text{call}(\text{on}\_\text{Trig}, \text{on}\_\text{Evt}) \) on line 24 reducing to \( \nu_8 \) is \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))]\).

4. the trace produced while calling \( \text{call}(\text{on}\_\text{Trig}, \text{on}\_\text{Evt}) \) on line 25 reducing to \( \nu_8 \) is \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))]\).

5. containing of the two call expressions, the trace produced while the case expression spans from line 24 to 25 reducing to \( \nu_2 \) is \([\text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))]\).

6. the trace produced while the then branch spans from line 28 to 35 reducing to \( \nu_3 \) is \([\text{true}_N(\text{begin}\_\text{true}(\epsilon)), \text{true}_N(\text{true}_N(\text{begin}\_\text{true}(\epsilon)))\text{end}\_\text{true}(\epsilon); v_N]^N\).

7. the trace produced while the if-then-else expression spans from line 27 to 36 evaluating to \( \nu_4 \) is \([\text{true}_N(\text{begin}\_\text{true}(\epsilon)), \text{true}_N(\text{true}_N(\text{begin}\_\text{true}(\epsilon)))\text{end}\_\text{true}(\epsilon); v_N]^N\).

8. the trace \( \tau_8 \) produced while the expressions spans from line 23 to 26 reducing to \( \nu_6 \) is \([\text{true}_N(\text{begin}\_\text{true}(\epsilon)), \text{true}_N(\text{true}_N(\text{begin}\_\text{true}(\epsilon)))\text{end}\_\text{true}(\epsilon); v_N]^N\).

9. the trace \( \tau_9 \) produced while the while expression spans from line 5 to 37 reducing to \( \nu_7 \) is \([\text{true}_N(\text{begin}\_\text{true}(\epsilon)), \text{true}_N(\text{true}_N(\text{begin}\_\text{true}(\epsilon)))\text{end}\_\text{true}(\epsilon); v_N]^N\).

10. the trace \( \tau_8 \) produced when the overall monitor function is reduced to \( \nu_8 \) is \([\text{true}_N(\text{begin}\_\text{true}(\epsilon)), \text{true}_N(\text{true}_N(\text{begin}\_\text{true}(\epsilon)))\text{end}\_\text{true}(\epsilon); v_N]^N\).

11. \( p \) produces the trace \( t = \text{begin}\_\text{true}(\epsilon), \text{end}_N(v_n), \text{true}_N(\text{begin}\_\text{true}(\epsilon))\).

12. \( t \) can be simplified to \([\text{begin}\_\text{true}(\epsilon), \text{true}_N(\text{begin}\_\text{true}(\epsilon))\text{end}\_\text{true}(\epsilon); v_N]^N\).

13. \( t \) matches infinite-expression \([\text{begin}\_\text{true}(\epsilon), \text{true}_N(\text{begin}\_\text{true}(\epsilon))\text{end}\_\text{true}(\epsilon); v_N]^N\).

\[\square\]
THEOREM 12 (Obligation Reaction Part 2). For all programs \( p \) s.t. \( \Lambda \vdash p \ ok \), and \( p \)’s monitor is \((M, \ fun_{\ mon}, p_1 : \cdots \ \ p_n, \ c_{\ mon}, e_{\ mon})\) where the functions \( e_{\ mon}, p_1, \ onTrigger, \ldots, p_n, \ onTrigger, p_1, \ onObligation, \ldots, p_n, \ onObligation \) terminate, for each event \( \text{end} \ makeCFG(v, v_2) : q \) in \( \tau \) there must exist a \( \nu_{\ true}(q) \) or \( \nu_{\ false}(q) \) in \( \tau \) where:

\[
\nu_{\ true}(e) := \begin{cases} \begin{align*}
& (\text{begin}_{\mon}(e) \ (\text{begin}_{\mon}(e))' \text{end}_{\mon}(e))
& \text{inOb}_{\mon}(e) : \nu_N
& \text{begin}_{\mon}(v_1, \ldots, v_N) \ (\text{begin}_{\mon}(e))' \text{end}_{\mon}(e) \ (\text{begin}_{\mon}(e))' \text{end}_{\mon}(e) : \nu_N
& \text{ob}_{\mon} \end{cases}
\]
\[
\nu_{\ false}(e) := \begin{cases} \begin{align*}
& (\text{begin}_{\mon}(e) \ (\text{begin}_{\mon}(e))' \text{end}_{\mon}(e))
& \text{inOb}_{\mon}(e) : \nu_N
& \text{begin}_{\mon}(v_1, \ldots, v_N) \ (\text{begin}_{\mon}(e))' \text{end}_{\mon}(e) \ (\text{begin}_{\mon}(e))' \text{end}_{\mon}(e) : \nu_N
& \text{ob}_{\mon} \end{cases}
\]

\]

PROOF.

Case 1: \( e_{\ app} \) does not contain any monitored functions calls:
1. \( \tau \) matches \( (\text{end} \ makeCFG(v, v_2) : q) \)
2. \( \nu_{\ true}(e) \) or \( \nu_{\ false}(e) \)

Lemma 10
1. Rules of \( \infty \)-expression

Case 2: \( e_{\ app} \) contains at least one monitored functions call:
1. \( \text{end} \ makeCFG(v, v_2) : p \) cannot occur directly from \( e_{\ app} \)
2. \( \text{call} \ (\text{monitor}, e) \) cannot occur directly from \( e_{\ app} \)
3. the two expressions \( \text{makeCFG} \) on lines 7 and 33 of monitor code can result in a trace including \( \text{end} \ makeCFG(v, v_2) : q \)
4. the result of the expression \( \text{makeCFG} \) on line 7 is immediately appended to a list stored in \( \ell_{\mon} \)
5. the list stored in \( \ell_{\mon} \) is prepended to a list stored in \( \ell_{\ob} \) on the line 10 of the monitor code resulting in all results of \( \text{makeCFG} \) in line 7 being included in \( \ell_{\ob} \)
6. the result of the expression \( \text{makeCFG} \) on line 33 immediately appended to a list stored in \( \ell_{\ob} \)
7. the list stored in \( \ell_{\ob} \) is prepended to a list stored in \( \ell_{\ob} \) on line 35 of the monitor code resulting in all results of \( \text{makeCFG} \) in line 33 being included in \( \ell_{\ob} \)
8. results from all possible calls to \( \text{makeCFG} \) are stored in \( \ell_{\ob} \)
9. the trace produced while code from lines 30 to 26 is reduced to a value is either \( \text{begin}_{\mon}(v_1, \ldots, v_N) \text{end}_{\mon}(v_1, \ldots, v_N) : \text{false} \) or \( \text{begin}_{\mon}(v_1, \ldots, v_N) \text{end}_{\mon}(v_1, \ldots, v_N) : \text{true} \)
10. the expression \( \text{ob}_{\mon} = \text{head}(\ell_{\ob}) \) on line 12 of the monitor code takes \( \ell_{\ob} \)’s first list of obligations and stores it in \( \ell_{\ob} \)
11. the expression \( \text{let} \ ob = \text{head}(\ell_{\ob}) \) on line 13 of the monitor code takes the first obligation stored in \( \ell_{\ob} \)
12. if \( \ell_{\ob} = [ob] \), then the expression \( \text{empty}(\text{tail}(\ell_{\ob})) \) on line 16 of the monitor code will evaluate to \( \text{false} \)
13. If \( \text{empty}(\text{tail}(\ell_{\ob})) \) evaluates to \( \text{false} \), then the first element of \( \ell_{\ob} \) will be removed after executing the expression \( \text{ob}_{\mon} = \text{tail}(\ell_{\ob}) \) on line 17 of the monitor code
14. if \( \ell_{\ob} = [ob, \ldots] \), then \( \text{empty}(\text{tail}(\ell_{\ob})) \) on line 16 of the monitor code will evaluate to \( \text{true} \)
15. If \( \text{empty}(\text{tail}(\ell_{\ob})) \) evaluates to \( \text{true} \), the first element of \( \ell_{\ob} \) will be replaced with \( \text{tail}(\ell_{\ob}) \) after executing the expression on line 16 of the monitor code
16. for each iteration, one obligation will be removed from \( \ell_{\ob} \) and stored in \( ob \)
17. no locations besides lines 10, 16, 17, and 35 in the monitor code assign to \( \ell_{\ob} \)
18. \( ob \) is only assigned to on line 13 of the monitor code where it is assigned with \( \ell_{\ob} \)’s the first list of obligations

Rules \( \text{deref}_{\mon}, \text{head}_{\mon} \), \( \text{assign}_{\mon} \), \( \text{Definition of monitor} \)
Rules \( \text{deref}_{\mon}, \text{head}_{\mon} \), \( \text{assign}_{\mon} \), \( \text{Definition of monitor} \)
Rules \( \text{deref}_{\mon}, \text{head}_{\mon} \), \( \text{assign}_{\mon} \), \( \text{Definition of monitor} \)
Rules \( \text{deref}_{\mon}, \text{head}_{\mon} \), \( \text{assign}_{\mon} \), \( \text{Definition of monitor} \)
Rules \( \text{deref}_{\mon}, \text{head}_{\mon} \), \( \text{assign}_{\mon} \), \( \text{Definition of monitor} \)
Rules \( \text{deref}_{\mon}, \text{head}_{\mon} \), \( \text{assign}_{\mon} \), \( \text{Definition of monitor} \)

Theorem 3(14)
There exists well-typed programs

**Theorem 13 (Pre-obligation Completeness).** There exists well-typed programs $p_1, p_2$, and $p_3$ where $p_1 \rightarrow^* p'_1, p_2 \rightarrow^* p'_2$, and $p_3 \rightarrow^* p'_3$ while producing traces $t_1, t_2$, and $t_3$ respectively such that $t_1, t_2$, and $t_3$ match the $\infty$-expressions $ile_{pre}$ for pre-obligations, $ile_{post}$ for post-obligations and $ile_{ongoing}$ for ongoing obligations where $ile_{pre} = (\infty) begin_f(x) (\infty) begin_monitor(act(f,x)) (\infty) end_f(x):v. (\infty))$, $ile_{post} = (\infty) end_f(x):v. (\infty) begin_monitor(res(act(f,x),x)) (\infty))$, and $ile_{ongoing} = (ile_{pre} \mid ile_{post}) (\infty) (ile_{pre} \mid ile_{post})$.

**Proof.**

**Case 1:** pre-obligation:

1. Assume $p_1$ has a policy $Pol_1$ with the onTrigger $onTrigger(e) = if matches(e, act("print", v)) then call(log, e) else unit
   
   **Assumption**

2. print $\in F$
   
   1, Definition of F

3. when $e_{app}$ attempts to execute $call(print, v)$ expression, $begin_{print(v)}$ gets added to the trace resulting in a trace of $[(\infty), begin_{print(v)}]$.
   
   2, Rule callFromApplication

4. when $e_{app}$ attempts to execute the expression $call(print, v)$, $call(monitor, act("print", v))$ will be triggered
   
   2, Rule callFromApplication

5. after $call(monitor, act("print", v))$ is executed, $begin_{monitor(act("print", v))}$ gets added to the trace resulting in a trace of $[(\infty), begin_{print(v)}, begin_{monitor(act("print", v))}]$
   
   3, 4, Rule callNonMonitoredFunctions

6. after expression added by Rule callFromApplication are fully executed, expression $print(v)$ will be executed next
   
   Rule callFromApplication

7. after the execution of $print(v)$, the trace is $[(\infty), begin_{print(v)}, begin_{monitor(act("print", v))}, (\infty), end_{print(x):v}. (\infty)]$
   
   5, 6, Rule endLabelValue

8. execution of $p_1$ results in a trace of $[(\infty), begin_{print(v)}, begin_{monitor(act("print", v))}, (\infty), end_{print(x):v}. (\infty)]$
   
   2, 7

9. execution of $p_1$ results in a trace of that matches $r_{preOb}$
   
   8, Rules of $\infty$-expression

**Case 2:** post-obligation:

1. Assume $p_2$ has a policy $Pol_2$ with the onTrigger $onTrigger(e) = if matches(e, res("print", v)) then call(log, e) else unit
   
   **Assumption**

2. print $\in F$
   
   1, Definition of F

3. when $e_{app}$ attempts to execute the expression $call(print, v)$, the $call(monitor, res("print", v))$ will be triggered
   
   2, Rule callFromApplication

4. since $Pol_2$ does not consider $call(print, v)$ as security relevant, setOutput() is never called in $Pol_2$
   
   1

5. $call(print, v)$ will be executed
   
   Rule callFromApplication, Rule while

6. after $call(print, v)$ is executed $end_{log(v)}:r$ gets added to the trace resulting in a trace of $[(\infty), end_{log(v)}:r]$.

7. a successful execution of $call(print, v)$ triggers $call(monitor, res(act("print", v), r))$
   
   Rule callFromApplication

8. after a successful execution of $call(monitor, res(act("print", v), r))$, the trace is $[(\infty), end_{log(v)}:r, (\infty), begin_{monitor(res(act("print", v), r))}]$
   
   7, Rule callNonMoniteredFunctions

9. execution of $p_2$ results in a trace of $[(\infty), end_{log(v)}:r, (\infty), begin_{monitor(res(act(print,v), r))}]$
   
   8

10. execution of $p_2$ results in a trace that matches $r_{postOb}$

9, Rules of $\infty$-expression
Theorem 14 (Policy Permutability).

If $p$ is a well-typed program with monitor ($fun_{mon}, p_1 :: \cdots :: p_n$, $e_{os}$, $e_{oc}$) and $p \rightarrow^* p'$ while producing trace $t$, then there exists $e_{os}$ and $e_{oc}$ such that if $q$ is a well-typed program with monitor ($fun_{mon}, p_1' :: \cdots :: p_n'$, $e_{os}$, $e_{oc}$) where $p_1' :: \cdots :: p_n'$ is a permutation of $p_1 :: \cdots :: p_n$ and $q \rightarrow^* q'$ while producing trace $t'$ then $t \equiv t'$ (all $e_{onTrigger}$, $e_{onObligation}$, $e_{os}$, and $e_{oc}$ must be pure functions).

Proof.

1. only $call(e_1, e_2)$ and $makeCFG(e)$ expressions can result in values being added to traces

2. the parts of $t$ and $t'$ generated outside of calls to the monitor are identical

3. calls to monitor from $p$ and $q$ will differ only in the order of policies in parameter

4. calls to monitor from $p$ and $q$ result in traces that are equivalent if the current traces are equivalent

5. only lines 6, 10, 23, 24, 25 and 41 of monitor result in values being added to the trace

6. calls to the expression $makeCFG(e)$ on line 6 of the monitor code in $p$ and $q$ will differ only in parameters

7. the traces result from calls to $makeCFG(e)$ on line 6 in $p$ and $q$ will be equivalent if the current traces are equivalent

8. the traces result from the execution of the expression line 25 in $p$ and $q$ will be identical if $pol$ in $p$ and $q$ are identical

9. $pol$ on line 25 in $p$ and $q$ will be identical for each iteration of the loop if the values of the $votingPols$ in $p$ and $q$ are identical

10. $votingPols$ in $p$ and $q$ are identical if $votingPolsList$ in $p$ and $q$ are identical

11. $votingPolsList$ in $p$ and $q$ will be identical at each iteration if the $votingPolsList$ in $p$ is initially assigned with a value that is identical to the value that is assigned to $votingPolsList$ in $q$

12. $votingPolsList$ in $p$ and $q$ will be assigned with an identical value if the results of $call(os, p_1 :: \cdots :: p_N)$ in $p$ and the result of $call(os, p_1' :: \cdots :: p_N')$ in $q$ are identical

13. assume the results of $call(os, p_1 :: \cdots :: p_N)$ in $p$ and the result of $call(os, p_1' :: \cdots :: p_N')$ in $q$ will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$

14. the traces result from the execution of the expression line 25 in $p$ and $q$ will be identical if $e_{os}(obs) = call(orderByPolicyName, obs)$

15. the traces result from the execution of the expression line 25 in $p$ and $q$ will be identical when the values of $votes$ in $p$ and $q$ are identical

16. $votes$ in $p$ and $q$ will be identical if traces produced on line 25 of $p$ and $q$ are identical

17. the traces result from each iteration of line 25 will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$

18. the traces result from the execution of the expression line 23 in $p$ and $q$ will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$

Case 3: ongoing-obligation:

1. assume $p_3$ has policies as specified in Case 1 and Case 2

2. execution of $p_3$ matches the $\infty$-expression

3. execution of $p_3$ matches the $\infty$-expression
19. the traces result from the execution of the case expression that spans from lines 24 to 25 in p and q will be identical for each iteration if the results of call(c.vc,!votes) in p and q on line 23 are identical as well as ob in p and q are identical

20. the traces result from the execution of the call(vc,!votes) expression in p and q will be identical when e_{os}(obs) = call(orderByPolicyName,obs)

21. ob in p and q will be identical for each iteration if pols on line 2 of the monitor code in p and q are identical

22. the traces result from the execution of the expressions span from lines 24 to 25 in p and q will be identical for each iteration with e_{os}(obs) = call(orderByPolicyName,obs)

23. the traces result from the execution of the expressions on line 9 for each iteration in p and q will be identical if pols in p and q are identical

24. the traces result from the execution of the expressions on line 9 for each iteration in p and q will be identical if votingPols in p and q are identical

25. the traces result from the execution of the expressions on line 42 for each iteration in p and q will be identical if votingPols in p and q are identical

26. votingPols in p and q will be identical when e_{os}(obs) is defined as call(orderByPolicyName,obs)

27. the traces result from the execution of the expressions on line 42 in p and q will be identical when e_{os}(obs) is defined as call(orderByPolicyName,obs)

28. when e_{os}(obs) = call(orderByPolicyName,obs), t = t'