Composition of Atomic-Obligation Security Policies

Danielle Ferguson*, Yan Albright*, Daniel Lomsak, Tyler Hanks, Kevin Orr, Jay Ligatti
Department of Computer Science and Engineering
University of South Florida

Technical Report CSE-SEC-021719

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 crosscutting concerns. As the complexity of policies increases, so does the likelihood of errors within the policies.

Following standard software-engineering practices, it is simpler to maintain modules of related functionality, where each security concern can be addressed in isolation, and then build more complex policies as compositions of the modules. When policies are simple enough, as with classic safety properties [2], composition is trivial because the only decision made is whether to permit or deny a given action; such decisions can be composed with boolean operators. However, these simple policies are insufficiently expressive in practice because they do not allow policies to propose alternative or additional actions to be executed. These actions, referred to as obligations [3], complicate the process of composing policies.

Obligation support enables policies that are impossible with safety properties. For example, a policy that grants or denies fund transfers may also include an obligation to log such requests for auditing, or a policy to prevent unintended file deletion may include an obligation to prompt the user for confirmation before rendering a decision on a file deletion request.

The challenge of handling conflicts in obligation-based policies is well known (e.g., [4–6]), but neglecting to do so could lead to unexpected behavior or security vulnerabilities. Consider policies P_a and P_b that respectively disallow file downloads and window pop-ups. P_a also defines an obligation to pop up a warning when a user attempts a file download, which violates P_b. A policy P that composes P_a and P_b using *conjunction* (i.e., enforcing both P_a and P_b) should disallow all downloads and pop-ups. However, without validating P_a's obligation with P_b's constraints, P would allow pop-ups.

Beyond these direct policy conflicts, some policies also require the ability to react to other policies' obligations. For example, a policy limiting the number of files open needs access to an accurate count of currently open files—including those opened and closed by other policies' obligations. If this open-file-limiting policy cannot observe and react to actions performed by other policies' obligations, it cannot be enforced.

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

*Joint first author

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- 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 execute 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,

	Supports obligations that are				Supports resolving conflicts for obligations that are		Supports reacting to obligations that are		Supports custom composition
	pre-	post-	effectful	Turing complete	non-atomic	atomic	non-atomic	atomic	
XACML [7]	✓		✓	✓					
XACML Extensions [8–12]	✓	✓	✓	✓	✓				✓
Polymer [4]	✓	✓	✓	✓	✓		✓		✓
Ponder [6, 13]	✓		✓	✓	✓				
SPL [5, 14]		✓				✓			
Heimdall [15]		✓							
Rei [16]		✓	✓	✓	✓				
Aspect Oriented [17]	✓	✓	✓	✓					

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

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

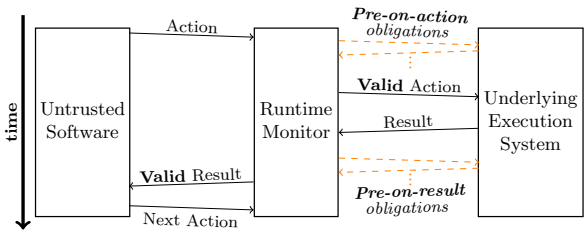

Figure 1: Obligations are either pre-on-action, which are fulfilled before a decision on an action is enforced, or pre-on-result, which are fulfilled before a decision is made to return a result.

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 $\tau_1 \rightarrow \tau_2$, homogeneous list τ_{List}, and reference type $\tau\ Ref$; 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(`e_1`,`e_2`)`

Types:

$$\tau ::= Bool \mid String \mid Int \mid TypedVal \mid Unit \mid \tau\ Ref \mid \tau_{List} \mid \tau_1 \rightarrow \tau_2 \mid (\ell_1 : \tau_1 \times \ldots \times \ell_n : \tau_n) \mid (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$$

$$\begin{aligned}
Act &\equiv (name : String \times arg : TypedVal)\\
\tau\ Res &\equiv (act : Act \times result : \tau)\\
Res &\equiv (act : Act \times result : TypedVal)\\
\tau\ Event &\equiv (act : Act + res : \tau\ Res)\\
Event &\equiv (act : Act + res : Res)\\
Obligation &\equiv (ot : (evt : Event \times onTrig : Event \rightarrow unit) +\\
&\quad\ oo : (rt : Res_{List} \times onOblig : Res_{List} \rightarrow unit))\\
CFG &\equiv (nodes : Act_{List} \times\\
&\quad\ edges : (start : Act \times end : Act)_{List} \times\\
&\quad\ obligation : Obligation)\\
Pol &\equiv (name : String \times\\
&\quad\ onTrigger : Event \rightarrow unit \times\\
&\quad\ onObligation : Res_{List} \rightarrow unit \times\\
&\quad\ vote : CFG \rightarrow Bool)\\
OS &\equiv (pol : Pol \times cfg : CFG)_{List} \rightarrow\\
&\quad\ (pol : Pol \times cfg : CFG)_{List}\\
VC &\equiv (name : String \times vote : Bool)_{List} \rightarrow Bool\\
\tau\ Option &\equiv (some : \tau + none : Unit)
\end{aligned}$$

Values:

$$\begin{aligned}
b &::= true \mid false\\
f &::= fun\ x_1(x_2 : \tau_1) : \tau_2 = e\\
v &::= b \mid s \mid n \mid f \mid unit \mid v_1 :: v_2 \mid [] : \tau_{List} \mid \ell\\
&\quad \mid in_\ell\ v : \tau \mid (\ell_1 = v_1, \ldots, \ell_2 = v_n)\\
&\quad \mid makeTypedVal(\tau, v)
\end{aligned}$$

Expressions:

$$\begin{aligned}
e ::= {} & v \mid x \mid e_1; e_2 \mid e_1 :: e_2 \mid e_1 @ e_2 \mid e_1 \vee e_2 \mid e_1 \wedge e_2 \mid e_1 == e_2\\
& \mid \neg e_1 \mid ref\ e \mid !e \mid e_1 := e_2 \mid e_1 + e_2 \mid while(e_1)\ \{e_2\}\\
& \mid let\ x = e_1\ in\ e_2\ end \mid in_\ell\ e : \tau \mid head(e) \mid monitor(\tau, e)\\
& \mid if\ e_1\ then\ e_2\ else\ e_3 \mid (\ell_1 = e_1, \ldots, \ell_n = e_n) \mid e.\ell_i \mid tail(e)\\
& \mid (\ case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n\ )\\
& \mid setOutput(e) \mid getOutput() \mid outputNotSet() \mid getRT()\\
& \mid call(e_1, e_2) \mid invoke(e_1, e_2) \mid \{e\}_{x(v)} \mid makeCFG(e)\\
& \mid makeTypedVal(\tau, e_1) \mid tryCast(\tau, e_1) \mid empty(e)
\end{aligned}$$

$$\begin{aligned}
act(e_1, e_2) &\equiv (name = e_1, arg = e2)\\
res(e_1, e_2) &\equiv (act = e_1, result = e2)
\end{aligned}$$

Monitored Functions $F ::= \bullet \mid (s, f), F$

Monitors $R ::= (F, pols, os, vc)$, where $pols$, os, vc are values

Memories $M ::= \bullet \mid (\ell, v), M$

Configurations:

$C ::= (M, R, inOb, rt, out, \tau_{out})$, where $inOb$, rt, out are values

Labels $label ::= begin_{s(v)} \mid end_{s(v_1)} : v_2$

Figure 2: Formal syntax for PoCo

and `invoke`(e_1, e_2) can be used to call functions, the two expressions are designed for different purposes. The expression `call`(e_1, e_2) is used to call functions that reside within the application or within the policies and is the elimination form for the function type. On the other hand, `invoke`(e_1, e_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, $begin_f$ and 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 execution.

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 Γ maps variables to their types while Λ maps memory locations to types. The rule `makeCFG` specifies the type of the undefined *makeCFG* function. For function calls within target applications and policies, the rule `call` specifies that the first parameter is a function type and the second is the type of this function's parameter. Different from `call`, the rule `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 *String* type which specifies a valid function name (it is assumed that functions will have unique names and signatures). Lastly, since `label` elements are merely used for recording start and end points, their type depends only on the type of their subexpression.

$$\boxed{\Lambda, \Gamma \vdash e : \tau}$$

$$\frac{\Lambda, \Gamma \vdash e : Obligation}{\Lambda, \Gamma \vdash makeCFG(e) : CFG}\ \text{(makeCFG)}$$

$$\frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Lambda, \Gamma \vdash e_2 : \tau_1}{\Lambda, \Gamma \vdash call(e_1, e_2) : \tau_2}\ \text{(call)}$$

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

$$\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash \{\ e\ \}_{s(v)} : \tau}\ \text{(endLabel)}$$

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 `begin` and an `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, *1*) for a security-irrelevant function call, `callNonMonitoredFunction` directly evaluates the call and attaches a `begin` and an `end` label to the function call's execution trace. *2*) for a security-relevant function call that originates from the target application, the rule `callFromApplication` will invoke the monitor to resolve the security-relevant event; *3*) the rule `monitorV`, which is directly responsible for invoking the PoCo monitor, sets the flag `inOb` to true before evaluation to indicate the current executing context; *4*) for `onTrigger` and `onObligation` function calls, the rules `callOnTrigger` and `callOnObligation` reset the current *result trace* before evaluation; *5*) for a security-relevant function call (i.e., included in the list of monitored functions F) that originates from an obligation, the rule `callFromObligation` will append the result of evaluating the call to the current *result trace*;

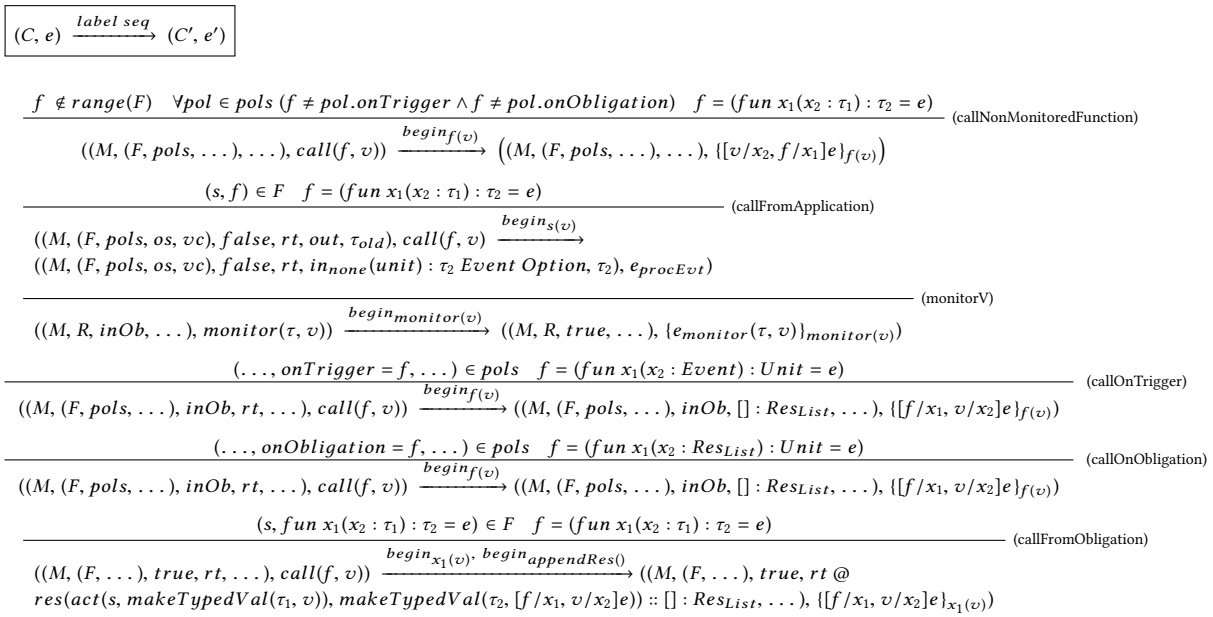

Figure 4: Dynamic semantics for function calls

4.4 Type Safety

The PoCo language is type safe via the standard Preservation and Progress Lemmas [25]. Type safety guarantees that well-typed PoCo programs will never get stuck (i.e., well-typed expressions are either values or can be further evaluated). The proof of type safety appears in Appendix E.

Theorem 2 (Type-safety).
$(C, e) : \tau \wedge (C, e) \longrightarrow^* (C', e') \Rightarrow$
$(C', e') : \tau \wedge$
$(\exists v : e' = v \ \vee \ \exists C'', e'' : \ (C', e') \longrightarrow (C'', e''))$.

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.

```
fun onTrigger(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)))); unit
      else unit
   | res r => 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.

```
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(
            in_res(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:

```
fun vote(cfg:CFG):Bool =
   ¬call(containsAct, (cfg=cfg, name="fopen",
   arg=(in_arg (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 undecidable. 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:

```
fun vote(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_none 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:

```
fun onObligation(rt: Res_List):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_1 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_{prelog} to log events in obligations prior to their execution, though it is possible to do so in `onTrigger`.

To summarize, there are two ways PoCo policies specify obligations: `onTrigger` specifies an obligation in response to a trigger event, and `onObligation` specifies an obligation in response to other obligations (which may be defined by `onTrigger` or `onObligation`). The `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:

```
fun disallow (x:Act) : Pol = (
name = dis_x,
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: Res_List):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)))
```

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_{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.

```
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):Unit = unit),
  vote = (fun vt(cfg:CFG):Unit  =
    ¬call(containsActAnyArg,
       (cfg=cfg, name="popup", count=2))
        ∧ ¬(call(containsActAnyArg,
           (cfg=cfg, name="popup", count=1))
            ∧ currTime < (!t+100)))) 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 `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 *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' `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 $(name : String \times vote : Bool)_{List} \rightarrow Bool$, 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:

```
fun VC_vote(votes:(name:String × vote:Bool)_List):Bool=
  let output = ref true in
  let rvotes = ref votes in
    while(¬empty(!rvotes)) {
      case head(!rvotes) of
        some v =>
          if v.name == "Pol1"
          then output := v.vote
          else unit
        none unit => unit;
      rvotes := tail(!rvotes)} 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:

```
fun VC_override(votes:(name:String × vote:Bool)_List)
   :Bool =
     call(VC_disjunction,
       call(VC_conjunction, tail(votes))::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 of 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 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 $(pol : Pol \times cfg : CFG)_{List} \rightarrow (pol : Pol \times cfg : CFG)_{List}$.

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:

```
fun OS_default(obs: CFG_List):CFG_List = 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.

```
fun OS_complexity(obs: CFG_List):CFG_List =
  call(sort, (list = obs,
    comparator = (fun c((o1,o2):(CFG×CFG)):Int =
      call(length,o1.nodes)-call(length,o2.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 o, the monitor collects a vote on o from each policy and sends these votes to the VC to make the final decision on whether the monitor should execute o.

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 $begin_{f(x)}$ and $end_{f(x):v}$ where $begin_{f(x)}$ indicates the beginning of a step with any applicable parameters and $end_{f(x):v}$ indicates the end of a step with applicable parameters and output. To simplify theorems' presentation, the syntax also defines the following values:

N	::=	# of policies in Pol_{List}
beginOb(e)	::=	$begin_{onTrigger(e)} \mid begin_{onObligation(e)}$
endOb(e)	::=	$end_{onTrigger(e)} \mid end_{onObligation(e)}$
ob(e)	::=	beginOb(e) | endOb(e)

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* $p \xrightarrow{t}{}^{*} p'$ *and t matches the* ∞*-expression* $((.^{\infty})\ beginOb(e_n)\ t'\ beginOb(e_m)\ .(.^{\infty}))$ *then t' matches the* ∞*-expression* $((.^{\infty})\ endOb(e_n)\ (.^{\infty}))$

Essentially, Theorem 3 states that another obligation will never start if the previous obligation has not completed execution. Note that an ∞-expression can generate possibly infinite length strings (i.e., belonging to the union of a regular and an ω language). A trace, t, is considered to match an ∞-expression, e, if t matches $[*/\infty]e$ or t matches a subexpression of e, $e_1, \ldots [\omega/\infty]e_i$. See Appendix F for a full definition of ∞-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* $p \xrightarrow{t}{}^{*} p'$, *t matches the* ∞*-expression* $(\neg beginOb(e))^{\infty}(v_{true}(e_n)beginOb(e_n)(\neg beginOb(e))^{\infty})^{\infty}$ *where:*

$v_{true}(e) ::= (begin_{vote(e)}\ (\neg beginOb(e))^{\infty}\ end_{vote(e)\ :\ v_n})^{N}$
$begin_{vc(v_1 :: \cdots ::v_N)}\ (\neg beginOb(e))^{\infty}\ end_{vc(v_1 :: \cdots ::v_N)\ :\ true}$

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 such that* $p \xrightarrow{t}{}^{*} p'$, *t matches the* ∞*-expression* $((\neg begin_{appendRes()})^{\infty}\ (begin_{appendRes()}\ .^{\infty}\ endOb(e)(.^{\infty}\ end_{makeCFG(onObligation,\ v)\ :\ g})^{N})?)^{\infty}$

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* $p \xrightarrow{t}{}^{*} p'$ *and p's monitor is the tuple* $(M,\ fun_{mon},\ p_1 :: \cdots :: p_n,\ e_{os},\ e_{vc})$ *where the functions* e_{vc}, $p_1.onTrigger, \ldots, p_n.onTrigger$, $p_1.onObligation, \ldots, p_n.onObligation$, *terminate, for each event* $end_{makeCFG(v1,\ v2)\ :\ g}$ *in t there must exist a* $v_{true(g)}$ *or* $v_{false(g)}$ *in t where:*

$v_{true}(e) ::= (begin_{vote(e)}\ (\neg beginOb(e))^{\infty}\ end_{vote(e)\ :\ v_n})^{N}$
$begin_{vc(v_1 :: \cdots :: v_N)}\ (\neg beginOb(e))^{\infty}\ end_{vc(v_1 :: \cdots ::v_N)\ :\ true}$ *and*

$v_{false}(e) ::= (begin_{vote(e)}\ (\neg beginOb(e))^{\infty}\ end_{vote(e)\ :\ v_n})^{N}$
$begin_{vc(v_1 :: \cdots :: v_N)}(\neg beginOb(e))^{\infty} end_{vc(v_1 :: \cdots ::v_N)\ :\ false}$

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 *where* $p_1 \xrightarrow{t_1}{}^{*} p_1'$, $p_2 \xrightarrow{t_2}{}^{*} p_2'$, *and* $p_3 \xrightarrow{t_3}{}^{*} p_3'$ *such that* t_1, t_2, *and* t_3 *match the* ∞*-expressions* e_{pre}, e_{post}, $e_{ongoing}$, *respectively where*

$e_{pre} ::= (.^{\infty})\ begin_{f(x)}\ (.^{\infty}) begin_{monitor(act(f,x))}\ (.^{\infty})\ end_{f(x):v}\ (.^{\infty})$

$e_{post} ::= (.^{\infty}) end_{f(x):v}\ (.^{\infty})\ begin_{monitor(res(act(f,x),rt))}\ (.\infty)$

$e_{ongoing} ::= (e_{pre} \mid e_{post})\ (.^{\infty})\ (e_{pre} \mid e_{post})$.

Theorem 7 shows that with the PoCo obligation design it is possible to implement pre-, post- and ongoing obligations by making use of both *pre-on-action* and *pre-on-result* obligations.

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 declared does not affect the outcome. This is a desirable feature because it allows for true modularity of policies and makes it simpler to test sets of policies in isolation. In order to prove this, we must first define what it means for the outcome to be unaffected. In general terms, this means that regardless of the order that policies are input, identical obligations should be executed in the same order, and the same output event should be decided. To formalize this, we define trace equivalence as:

$\boxed{\text{n1} \approx \text{n2}}$

$$\frac{n1 = n2}{n1 \approx n2}$$

$$\frac{\{p_1 :: \cdots :: p_n\} \approx \{p'_1 :: \cdots :: p'_n\} \quad n_1 = begin_{os(p_1:: \cdots ::p_n)} \quad n_2 = begin_{os(p'_1:: \cdots ::p'_n)}}{n1 \approx n2}$$

$$\frac{\{p_1 :: \cdots :: p_n\} \approx \{p'_1 :: \cdots :: p'_n\} \quad n_1 = end_{os(p_1:: \cdots ::p_n):v} \quad n_2 = end_{os(p'_1:: \cdots ::p'_n):v}}{n1 \approx n2}$$

$$\frac{\{p_1 :: \cdots :: p_n\} \approx \{p'_1 :: \cdots :: p'_n\} \quad n_1 = begin_{monitor(p_1:: \cdots ::p_n)} \quad n_2 = begin_{monitor(p'_1:: \cdots ::p'_n)}}{n1 \approx n2}$$

$$\frac{\{p_1 :: \cdots :: p_n\} \approx \{p'_1 :: \cdots :: p'_n\} \quad n_1 = end_{monitor(p_1:: \cdots ::p_n):v} \quad n_2 = end_{monitor(p'_1:: \cdots ::p'_n):v}}{n1 \approx n2}$$

$$\frac{n_1 = end_{makeCFG(o):v} \quad n_2 = end_{makeCFG(o'):v} \quad o \approx o'}{n1 \approx n2}$$

Trace equivalence guarantees that effectful code is executed in the same order in each trace.

Theorem 8 (Policy Permutability). *There exists well-typed programs p_1 with monitor $(M, fun_{mon}, p_1 :: \cdots :: p_n, e_{os}, e_{vc})$ and p_2 with monitor $(M, fun_{mon}, p'_1 :: \cdots :: p'_n, e_{os}, e_{vc})$ where $p'_1 :: \cdots :: p'_n$ is a permutation of $p_1 :: \cdots :: p_n$ such that $p_1 \xrightarrow{t_1}{}^* p'_1, p_2 \xrightarrow{t_2}{}^* p'_2$, and $t_1 \approx t_2$.*

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.

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 *advice*, 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 $P_{DisSysCalls}$ [4] which prevents a target application from exploiting *java.lang.Runtime.exec* methods. This policy's `onTrigger` calls the *matches* method to inspect the trigger action using the wildcard $*$ to avoid listing all six overloaded *exec* methods. If a trigger action matches the *java.lang.Runtime.exec* methods, the policy attempts to halt the target application by changing the output event to null as the following example illustrates.

```
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.

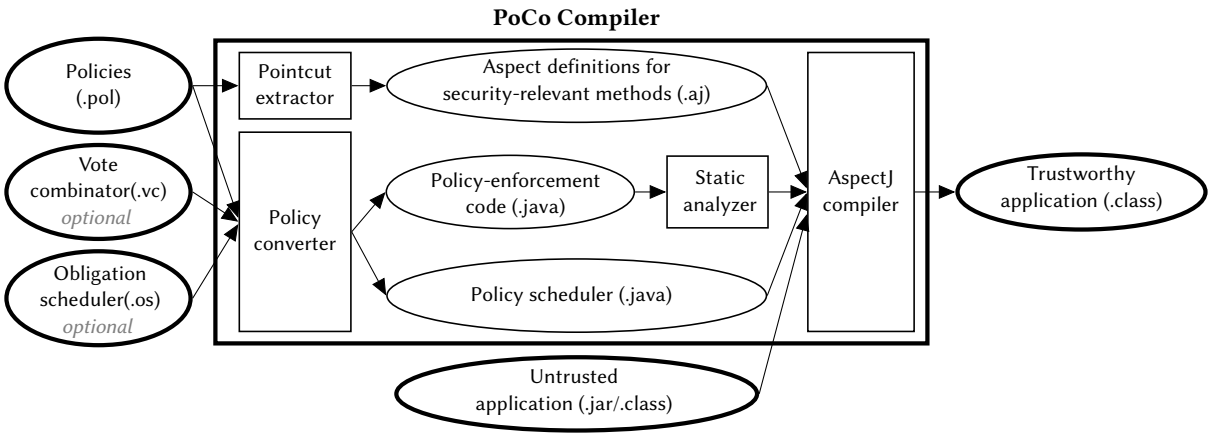

Figure 5: Overview of the PoCo compiler architecture. The compiler takes as input an untrusted application and outputs the same application with policy-enforcement code inlined before and after all security-relevant methods. Ovals are used to represent code files while rectangles represent processes executed during compilation.

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.

```
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 `e.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 $setOutput(trigger)$ 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.

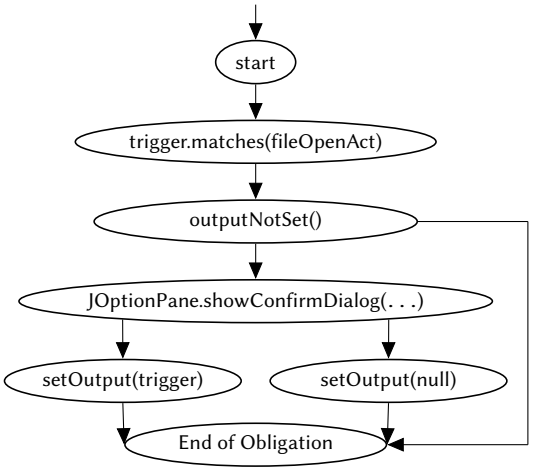

Figure 6: The control flow graph of $P_{confirm}$'s `onTrigger` function.

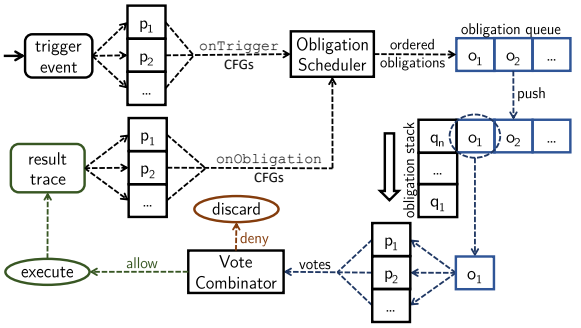

Figure 7: Policy scheduler flow — obligations are generated by policies based on the trigger event, prioritized by the obligation scheduler and then voted on. Executed obligations can result in additional obligations and this process continues until there are no remaining obligations to prioritize and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

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 `onObligation` of each policy to determine if the executed obligation triggers any additional obligations. As with the *onTrigger*s, 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.

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.

		No policy	One trivial policy	Ten trivial policies
Load Application	*Average(ms)*	6075.85	6128.40	6169.49
	Median(ms)	6187.14	6230.36	6255.93
	Overhead	-	0.87%	1.54%
Load Email Details	*Average(ms)*	286.40	272.52	300.89
	Median(ms)	282.24	281.38	302.29
	Overhead	-	0.39%	5.06%

Table 2: PoCo performance statistics on email client over 100 runs.

- `InterruptToCheckMem` - monitors the memory consumption of the target application.
- `Reflection` - prevents Java reflection methods from being used to call PoCo methods.

In addition to the variances noted in the policies above, PoCo's flat policy structure, rather than the tree-like structure of Polymer, means that PoCo does not need Polymer's concept of superpolicies (policies parameterized by other policies [4]); PoCo instead sorts and composes a list of policies using an obligation scheduler and a vote combinator. In order to achieve similar effects to those seen in the Polymer work, the default `OS` (`OrderAsListed`) and the default `VC` (`Conjunction`) were applied to the policies in the order in which they are listed above.

These policies are encoded in 1138 lines of PoCo code. We have successfully enforced this composed email policy on Pooka [32], an open-source email client, without modifying the application's source code.

PoCo performance was measured as the run-time overhead incurred by the system, since this impacts user experience. Specifically, the average overhead for loading the application and loading details for a specific email were measured. The application-loading time was measured from when the email client begins execution to when the user can view the inbox. The details-loading time, on the other hand, was calculated from the moment a user clicks on a specific email to the moment when the user can view that email's details. To take these measurements, an AspectJ program intercepted events from Pooka and recorded the time at relevant points during its execution.

These time vectors were measured under three scenarios. First, measuring these times without enforcing any policies establishes a baseline. Then, measuring these times with one *Trivial* policy as well as a policy composed of ten *Trivial* policies establishes how much of this overhead is due to the monitor and how much is due to the overhead of the individual policies. These measurements were performed using the same pointcuts that were generated by PoCo for the composed email policy, ensuring that the same events were considered security relevant across all four scenarios. Finally, the run-time overhead of the fully implemented email policy was measured. This evaluation was conducted on a MacBook Pro laptop running macOS Sierra version 10.12.4 with 8GB of memory and a 2.9GHz Intel quad-core i7 processor. For each scenario and time period, the test was repeated 100 times on a consistent university-network environment. The email account that was used to complete the testing contained 15 incoming emails.

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

eXtensible Access Control Markup Language (XACML) [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

[1] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, pp. 8 – 17, 2013.

[2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, pp. 125–143, March 1977.

[3] C. Xu and P. W. Fong, "The specification and compilation of obligation policies for program monitoring," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pp. 77–78, 2012.

[4] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 3, pp. 1–43, 2009.

[5] C. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes, "SPL: An access control language for security policies with complex constraints," in *In Network and Distributed System Security Symposium*, pp. 89–107, 2001.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "Ponder: A language for specifying security and management policies for distributed systems," tech. rep., Imperial College. UK, Research Report Department of Computing, 2000.

[7] H. L. Bill Parducci and R. Levinson, "Oasis extensible access control markup language (xacml)." http://www.oasis-open.org/committees/tc_home.phpl, 2012.

[8] J. Alqatawna, E. Rissanen, and B. Sadighi, "Overriding of access control in xacml," in *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, pp. 87–95, June 2007.

[9] D. W. Chadwick, L. Su, and R. Laborde, "Providing secure coordinated access to grid services," in *Proceedings of the 4th International Workshop on Middleware for Grid Computing*, MCG '06, (New York, NY, USA), pp. 1–, ACM, 2006.

[10] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin, "Access control policy combining: theory meets practice," in *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, (New York, NY, USA), pp. 135–144, ACM, 2009.

[11] M. Lischka, "Dynamic obligation specification and negotiation," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pp. 155–162, April 2010.

[12] N. Li, H. Chen, and E. Bertino, "On practical specification and enforcement of obligations," in *Proceedings of the ACM conference on Data and Application Security and Privacy*, pp. 71–82, 2012.

[13] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A policy system for autonomous pervasive environments," in *International Conference on Autonomic and Autonomous Systems*, pp. 330–335, 2009.

[14] C. Ribeiro, A. Zúquete, and P. Ferreira, "Enforcing obligation with security monitors," in *International Conference on Information and Communications Security*, pp. 172–176, 2001.

[15] P. Gama and P. Ferreira, "Obligation policies: An enforcement platform," in *IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 203–212, 2005.

[16] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 63–74, June 2003.

[17] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," *SIGSOFT Softw. Eng. Notes*, vol. 26, pp. 313–, Sept. 2001.

[18] J. Park and R. Sandhu, "Towards usage control models: beyond traditional access control," in *Proceedings of the ACM symposium on Access control models and technologies*, pp. 57–64, 2002.

[19] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Transactions on Information and System Security*, vol. 1, no. 1, pp. 128–174, 2004.

[20] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera, "Provisions and obligations in policy management and security applications," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 502–513, 2002.

[21] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[22] K. Tian, K. Cooper, K. Zhang, and H. Yu, "A classification of aspect composition problems," in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pp. 101–109, July 2009.

[23] K. Irwin, T. Yu, and W. H. Winsborough, "On the modeling and analysis of obligations," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 134–143, 2006.

[24] OASIS, "eXtensible Access Control Markup Language (XACML) Version 3.0." http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html. Accessed: 2017-04-22.

[25] A. Wright and M. Felleisen, "A syntactic approach to type soundness," pp. 38–94, 1994.

[26] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *Proceedings of the 15th European conference on Research in computer security*, ESORICS, pp. 87–100, 2010.

[27] P. W. L. Fong, "Access control by tracking shallow execution history," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 43–55, IEEE, May 2004.

[28] AspectJ, "The Aspectj & Project." https://www.eclipse.org/aspectj. Accessed: 2017-04-12.

[29] J. Ligatti, B. Rickey, and N. Saigal, "LoPSiL: A location-based policy-specification language," *Security and Privacy in Mobile Information and Communication Systems*, pp. 265–277, 2009.

[30] "Compiler tree API." http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/, 2017.

[31] "ASM Consortium." http://asm.ow2.org/index.html, 2017.

[32] A. Petersen, "Pooka: A java email client, 2003." http://www.suberic.net/pooka/. Accessed: 2018-01-12.

[33] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 30–50, Feb. 2000.

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 τ and $config$. An instance $e_{monitor}(\tau, (evt = e, pols = pols, os = os, vc = vc))$ monitors the security-relevant event e—whose return type is τ—using policies $pols$, obligation scheduler os, and vote combinator vc.

```
e_monitor(τ, c) :=
let pols = ref c.pols in
let obQueue = ref []:(pol:Pol × cfg:CFG)_List in
let obStack = ref []:(pol:Pol × cfg:CFG)_List_List in
  while (¬empty(pols)) {
    obQueue := !obQueue @ (pol=head(!pols), cfg=
    makeCFG(in_ot(evt=c.evt,onTrig=head(!pols).onTrigger):Obligation))::[]:(pol:Pol × cfg:CFG)_List;
    pols := tail(!pols); };
  let votingPolList = call(c.os, !obQueue) in
    obStack := votingPolList :: !obStack;
    while (¬empty(!obStack)) {
      obQueue := head(!obStack);
      let ob = head((!obQueue).cfg) in
      let votingPols = ref votingPolList in
      let votes = ref []:Bool_List in
        if ¬empty(tail(!obQueue)) then obStack := tail(!obQueue) :: tail(!obStack)
        else obStack := tail(!obStack);
        while(¬empty(!votingPols)) {
          let pol = head(!votingPols).pol in
            votingPols := tail(!votingPols);
            votes := !votes @ call(pol.vote, ob) :: []:Bool_List
          end };
        if call(c.vc, !votes) then
          case ob.obligation of ot o_1 ⇒ call(o_1.onTrig, o_1.evt)
                              | oo o_2 ⇒ call(o_2.onOblig, o_2.rt)
        else unit;
        if ¬empty(getRT()) then
          votingPols := votingPolList;
          obQueue := []:(pol:Pol ×cfg:CFG)_List;
          while(¬empty(!votingPols)) {
            obQueue := !obQueue @ (pol=head(!votingPols).pol, cfg=makeCFG(in_oo
            (rt=getRT(), onOblig=head(!votingPols).pol.onObligation):Obligation))
            :: []:(pol:Pol ×cfg:CFG)_List;
            votingPols := tail(!votingPols); };
          obStack := !obQueue :: !obStack
        else unit
      end end end }
  end end end end;
case getOutput() of some o ⇒ o | none unit ⇒ 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.

```
(
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: Res_List):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_none unit:(arg:TypedVal + none: unit, count=1)))
)
```

Figure 9: P_{file} disallows users and obligations from opening the secret.txt file

```
fun disallow (x:Act) : Pol = (
    name = dis_x,
    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: Res_List):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: fun disallow specifies a family of policies which disallow the action x

```
(
name = pol_postlog,
onTrigger = (fun ot(e:Event):Unit =
    case e of
        act a => unit
      | res r => if r.act.name == "fopen" then call(log,e) else unit),
onObligation = (fun oo(rt: Res_List):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: $P_{postlog}$ 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 a_1 =>
                                if a_1.name == e.name
                                then call(log,e) else unit
                          | res r_1 => unit
                | none n => unit
      | res r => unit ),
onObligation = (fun oo(rt: Res_List):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 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(in_res(res(a,makeTypedVal(Unit,unit))):Event)); unit
            else unit
     | res r => unit),
onObligation = (fun oo(rt: Res_List):Unit = unit),
vote = (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: Res_List):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))) end
```

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), Λ maps locations to values while Γ maps variables to values. It can be said that "under the contexts Λ and Γ, the expression e is of type τ" if and only if the judgment $\Lambda, \Gamma \vdash e : \tau$ is derivable by the following rules:

$\boxed{\Lambda, \Gamma \vdash e : \tau}$

$$\frac{}{\Lambda, \Gamma \vdash n : Int}\ (\text{intVal}) \qquad \frac{}{\Lambda, \Gamma \vdash b : Bool}\ (\text{boolVal}) \qquad \frac{}{\Lambda, \Gamma \vdash s : String}\ (\text{stringVal})$$

$$\frac{}{\Lambda, \Gamma \vdash unit : Unit}\ (\text{unitVal}) \qquad \frac{}{\Lambda, \Gamma' \cup \{x : \tau\} \vdash x : \tau}\ (\text{var}) \qquad \frac{}{(\Lambda' \cup \{\ell : \tau\}), \Gamma \vdash \ell : \tau\ Ref}\ (\text{location})$$

$$\frac{\Lambda, \Gamma \vdash e_1 : Int \quad \Lambda, \Gamma \vdash e_2 : Int}{\Lambda, \Gamma \vdash e_1 + e_2 : Int}\ (\text{add}) \qquad \frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash ref\ e : \tau\ Ref}\ (\text{createRef}) \qquad \frac{\Lambda, \Gamma \vdash e : \tau\ Ref}{\Lambda, \Gamma \vdash !e : \tau}\ (\text{accessRef})$$

$$\frac{\Lambda, \Gamma \vdash e : Bool}{\Lambda, \Gamma \vdash \neg e : Bool}\ (\text{negation}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : Bool}{\Lambda, \Gamma \vdash e_1 \wedge e_2 : Bool}\ (\text{con}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : Bool}{\Lambda, \Gamma \vdash e_1 \vee e_2 : Bool}\ (\text{dis})$$

$$\frac{\Lambda, \Gamma \vdash e : \tau_{List}}{\Lambda, \Gamma \vdash head(e) : \tau\ Option}\ (\text{head}) \qquad \frac{\Lambda, \Gamma \vdash e : \tau_{List}}{\Lambda, \Gamma \vdash tail(e) : \tau_{List}}\ (\text{tail}) \qquad \frac{\Lambda, \Gamma \vdash e : \tau_{List}}{\Lambda, \Gamma \vdash empty(e) : Bool}\ (\text{empty})$$

$$\frac{}{\Lambda, \Gamma \vdash ([] : \tau_{List}) : \tau_{List}}\ (\text{listEmptyVal}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : \tau \quad \Lambda, \Gamma \vdash e_2 : \tau_{List}}{\Lambda, \Gamma \vdash e_1 :: e_2 : \tau_{List}}\ (\text{listCons}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : \tau_{List} \quad \Lambda, \Gamma \vdash e_2 : \tau_{List}}{\Lambda, \Gamma \vdash e_1 \ @\ e_2 : \tau_{List}}\ (\text{listAppend})$$

$$\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash \{\ e\ \}_{s(v)} : \tau}\ (\text{endLabel}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda, \Gamma \vdash e_2 : \tau_1}{\Lambda, \Gamma \vdash call(e_1, e_2) : \tau_2}\ (\text{call}) \qquad \frac{}{\Lambda, \Gamma \vdash getRT() : Res_{List}}\ (\text{getRT})$$

$$\frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Lambda, \Gamma \vdash let\ x = e_1\ in\ e_2\ end : \tau_2}\ (\text{let}) \qquad \frac{\Lambda, \Gamma \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma \vdash (in_{\ell_i}\ e_i : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}\ (\text{variant})$$

$$\frac{\Lambda, \Gamma \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma \vdash e.\ell_i : \tau_i}\ (\text{projection}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma \vdash e_n : \tau_n}{\Lambda, \Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}\ (\text{record})$$

$$\frac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : \tau \quad \Lambda, \Gamma \vdash e_3 : \tau}{\Lambda, \Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}\ (\text{if}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : \tau}{\Lambda, \Gamma \vdash while(e_1)\ \{e_2\} : Bool}\ (\text{while})$$

$$\frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_2}{\Lambda, \Gamma \vdash e_1;\ e_2 : \tau_2}\ (\text{sequence}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : \tau\ Ref \qquad \Lambda, \Gamma \vdash e_2 : \tau}{\Lambda, \Gamma \vdash e_1\ :=\ e_2 : Unit}\ (\text{assignment})$$

$$\frac{\Lambda, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e : \tau_2}{\Lambda, \Gamma \vdash fun\ x_1(x_2 : \tau_1) : \tau_2 = e : \tau_1 \to \tau_2}\ (\text{fun}) \qquad \frac{\Lambda, \Gamma \vdash e : Obligation}{\Lambda, \Gamma \vdash makeCFG(e) : CFG}\ (\text{makeCFG})$$

$$\frac{\Lambda, \Gamma \vdash e : Event}{\Lambda, \Gamma \vdash setOutput(e) : Bool}\ (\text{setOutput}) \qquad \frac{}{\Lambda, \Gamma \vdash getOutput() : Event\ Option}\ (\text{getOutput})$$

$$\frac{}{\Lambda, \Gamma \vdash outputNotSet() : Bool}\ (\text{outputNotSet}) \qquad \frac{\Lambda, \Gamma \vdash e_1 : \tau \quad \Lambda, \Gamma \vdash e_2 : \tau \quad \tau \in \{Int, Bool, String\}}{\Lambda, \Gamma \vdash e_1 == e_2 : Bool}\ (\text{equality})$$

$$\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash makeTypedVal(\tau, e) : TypedVal}\ (\text{makeTypedVal}) \qquad \frac{\Lambda, \Gamma \vdash e : TypedVal}{\Lambda, \Gamma \vdash tryCast(\tau, e) : \tau\ Option}\ (\text{tryCast})$$

$$\frac{\Lambda, \Gamma \vdash e : (evt : \tau\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda, \Gamma \vdash monitor(\tau, e) : \tau\ Event}\ (\text{monitor})$$

$$\frac{\Lambda, \Gamma \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau}{\Lambda, \Gamma \vdash (case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau}\ (\text{case})$$

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

$\boxed{\Lambda, \Gamma \vdash e \ \textbf{ok}}$

The rules for $\Lambda, \Gamma \vdash e$ **ok** are the same as for $\Lambda, \Gamma \vdash e : \tau$ (that is, $\Lambda, \Gamma \vdash e$ **ok** $\Leftrightarrow \exists \tau . \Lambda, \Gamma \vdash e : \tau$), except there are no equivalents for Rules label and monitor. $\Lambda, \Gamma \vdash e$ **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 ($monitor(\tau, e)$). 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 $(C, invoke(\text{"}exit\text{"}, v)) \longrightarrow (C, \{\cdots\}_{exit(v)})$.

$\boxed{\Lambda \vdash F \ \textbf{ok}}$

A list of monitored functions F is **ok** iff for each pair (s_i, f_i), s_i is of type $String$ and f_i is of type $\tau_1 \to \tau_2$, for some τ_1 and τ_2. We also assume that each monitored function name (i.e., each s_i) is unique in F.

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

$$\frac{F = \{(s_1, f_1), \ldots, (s_n, f_n)\} \qquad \forall i \in \{1, \ldots, n\} \ \ \Lambda, \bullet \vdash s_i : String \qquad \forall i \in \{1, \ldots, n\} \ \ \exists \tau_1 \exists \tau_2 \ \ \Lambda, \bullet \vdash f_i : \tau_1 \to \tau_2}{\Lambda \vdash F \ \textbf{ok}} \text{ (F-ok)}$$

$\boxed{\Lambda \vdash R \ \textbf{ok}}$

$$\frac{\begin{array}{c}\Lambda \vdash F \ \textbf{ok} \\ \Lambda, \bullet \vdash pols : Pol_{List} \\ \Lambda, \bullet \vdash os : OS \\ \Lambda, \bullet \vdash vc : VC\end{array}}{\Lambda \vdash (F, pols, os, vc) \ \textbf{ok}} \text{ (R-ok)}$$

$\boxed{M : \Lambda}$

$$\frac{M = \{(\ell_1, v_1), \ldots, (\ell_n, v_n)\} \qquad \Lambda = \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \qquad \forall i \in \{1, \ldots, n\} \ \ \Lambda, \bullet \vdash v_1 : \tau_1}{M : \Lambda} \text{ (TMem)}$$

$\boxed{\Lambda \vdash C \ \textbf{ok}}$

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 $Event$, but differs in that the $result$ field of the res variant is of type τ_{out}, not $TypedVal$. 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.

$$\frac{\begin{array}{c}M : \Lambda \\ \Lambda \vdash R \ \textbf{ok} \\ \Lambda, \bullet \vdash inOb : Bool \\ \Lambda, \bullet \vdash rt : Res_{List} \\ \Lambda, \bullet \vdash out : \tau \ Event \ Option\end{array}}{\Lambda \vdash (M, R, inOb, rt, out, \tau_{out}) \ \textbf{ok}} \text{ (C-ok)}$$

$\boxed{\Lambda \vdash (C, e) : \tau}$

$$\frac{\Lambda \vdash C \ \textbf{ok} \qquad \Lambda, \bullet \vdash e : \tau}{\Lambda \vdash (C, e) : \tau} \text{ (TConfig)}$$

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 $makeCFG_\alpha$ 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. $\forall v \forall \Lambda \ \Lambda, \bullet \vdash v : Obligation \Rightarrow \Lambda, \bullet \vdash makeCFG_\alpha(v) : CFG$

$\boxed{(C, e) \xrightarrow{label_1, \ldots, label_n} (C', e')}$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 \wedge e_2) \longrightarrow (C', e_1' \wedge e_2)}\ (\text{andE}) \qquad \frac{}{(C, true \wedge e_2) \longrightarrow (C, e_2)}\ (\text{andTrue}) \qquad \frac{}{(C, false \wedge e_2) \longrightarrow (C, false)}\ (\text{andFalse})$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 \vee e_2) \longrightarrow (C', e_1' \vee e_2)}\ (\text{orE}) \qquad \frac{}{(C, true \vee e_2) \longrightarrow (C, true)}\ (\text{orTrue}) \qquad \frac{}{(C, false \vee e_2) \longrightarrow (C, e_2)}\ (\text{orFalse})$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, \neg e) \longrightarrow (C', \neg e')}\ (\text{notE}) \qquad \frac{}{(C, \neg false) \longrightarrow (C, true)}\ (\text{notFalse}) \qquad \frac{}{(C, \neg true) \longrightarrow (C, false)}\ (\text{notTrue})$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 + e_2) \longrightarrow (C', e_1' + e_2)}\ (\text{addE1}) \qquad \frac{(C, e_2) \longrightarrow (C', e_2')}{(C, n_1 + e_2) \longrightarrow (C', n_1 + e_2')}\ (\text{addE2}) \qquad \frac{}{(C, n_1 + n_2) \longrightarrow (C, n_1 +_\alpha n_2)}\ (\text{addValue})$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 == e_2) \longrightarrow (C', e_1' == e_2)}\ (\text{eqE1}) \qquad \frac{(C, e_2) \longrightarrow (C', e_2')}{(C, v_1 == e_2) \longrightarrow (C', v_1 == e_2')}\ (\text{eqE2})$$

$$\frac{n_1 = n_2}{(C, n_1 == n_2) \longrightarrow (C, true)}\ (\text{eqIntTrue}) \qquad \frac{n_1 \neq n_2}{(C, n_1 == n_2) \longrightarrow (C, false)}\ (\text{eqIntFalse})$$

$$\frac{}{(C, true == b_2) \longrightarrow (C, b_2)}\ (\text{eqBoolTrue}) \qquad \frac{}{(C, false == b_2) \longrightarrow (C, \neg b_2)}\ (\text{eqBoolFalse})$$

$$\frac{s_1 = s_2}{(C, s_1 == s_2) \longrightarrow (C, true)}\ (\text{eqStrTrue}) \qquad \frac{s_1 \neq s_2}{(C, s_1 == s_2) \longrightarrow (C, false)}\ (\text{eqStrFalse})$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1; e_2) \longrightarrow (C', e_1'; e_2)}\ (\text{sequenceE1}) \qquad \frac{}{(C, v_1; e_2) \longrightarrow (C, e_2)}\ (\text{sequenceE2})$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, !e) \longrightarrow (C', !e')}\ (\text{derefE}) \qquad \frac{}{((M \cup \{(\ell, v)\}, \ldots), !\ell) \longrightarrow ((M \cup \{(\ell, v)\}, \ldots), v)}\ (\text{derefValue})$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, ref\ e) \longrightarrow (C', ref\ e')}\ (\text{refE}) \qquad \frac{\ell \notin dom(M)}{((M, \ldots), ref\ v) \longrightarrow ((M \cup \{(\ell, v)\}, \ldots), \ell)}\ (\text{refValue})$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, head(e)) \longrightarrow (C', head(e'))}\ (\text{listHeadE}) \qquad \frac{(C, e) \longrightarrow (C', e')}{(C, tail(e)) \longrightarrow (C', tail(e'))}\ (\text{listTailE})$$

$$\frac{}{(C, head(v_1 :: \cdots :: [] : \tau_{List})) \longrightarrow (C, in_{some}(v_1) : \tau\ Option)}\ (\text{listHeadCons}) \qquad \frac{}{(C, tail(v_1 :: v_2)) \longrightarrow (C, v_2)}\ (\text{listTailCons})$$

$$\frac{}{(C, head([] : \tau_{List})) \longrightarrow (C, in_{none}(unit) : \tau\ Option)}\ (\text{listHeadNil}) \qquad \frac{}{(C, tail([] : \tau_{List})) \longrightarrow (C, [] : \tau_{List})}\ (\text{listTailNil})$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, empty(e)) \longrightarrow (C', empty(e'))}\ (\text{listEmptyE}) \qquad \frac{}{(C, empty([] : \tau_{List})) \longrightarrow (C, true)}\ (\text{listEmptyNil})$$

$$\frac{}{(C, empty(v_1 :: v_2)) \longrightarrow (C, false)}\ (\text{listEmptyCons})$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 @ e_2) \longrightarrow (C', e_1' @ e_2)}\ (\text{listAppendE1}) \qquad \frac{(C, e_2) \longrightarrow (C', e_2')}{(C, v_1 @ e_2) \longrightarrow (C', v_1 @ e_2')}\ (\text{listAppendE2})$$

$$\frac{}{(C, ([] : \tau_{List}) \mathbin{@} v_3) \longrightarrow (C, v_3)}\ \text{(appendNil)}$$

$$\frac{}{(C, (v_1 :: v_2) \mathbin{@} v_3) \longrightarrow (C, v_1 :: (v_2 \mathbin{@} v_3))}\ \text{(appendCons)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 :: e_2) \longrightarrow (C', e_1' :: e_2)}\ \text{(listPrependE1)}$$

$$\frac{(C, e_2) \longrightarrow (C', e_2')}{(C, v_1 :: e_2) \longrightarrow (C', v_1 :: e_2')}\ \text{(listPrependE2)}$$

$$\frac{}{(C, if\ true\ then\ e_2\ else\ e_3)) \longrightarrow (C, e_2)}\ \text{(ifTrue)}$$

$$\frac{}{(C, if\ false\ then\ e_2\ else\ e_3)) \longrightarrow (C, e_3)}\ \text{(ifFalse)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, e.\ell_i) \longrightarrow (C', e'.\ell_i)}\ \text{(projectionE)}$$

$$\frac{i \in \{1, \ldots, n\}}{(C, (\ell_1 = v_1, \ldots, \ell_n = v_n).\ell_i) \longrightarrow (C, v_i)}\ \text{(projectionV)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 := e_2) \longrightarrow (C', e_1' := e_2)}\ \text{(assignE1)}$$

$$\frac{(C, e_2) \longrightarrow (C', e_2')}{(C, \ell_1 := e_2) \longrightarrow (C', \ell_1 := e_2')}\ \text{(assignE2)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, let\ x = e_1\ in\ e_2\ end) \longrightarrow (C', let\ x = e_1'\ in\ e_2\ end)}\ \text{(letE)}$$

$$\frac{}{(C, let\ x = v\ in\ e_2\ end) \longrightarrow (C, [v/x]e_2)}\ \text{(letValue)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, tryCast(\tau, e_1)) \longrightarrow (C', tryCast(\tau, e_1'))}\ \text{(tryCastE)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, setOutput(e)) \longrightarrow (C', setOutput(e'))}\ \text{(setOutputE)}$$

$$\frac{}{((\ldots, rt, out, \tau_{out}), getRT()) \longrightarrow ((\ldots, rt, out, \tau_{out}), rt)}\ \text{(getRTVal)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, makeCFG(e)) \longrightarrow (C', makeCFG(e'))}\ \text{(makeCFGE)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, call(e_1, e_2)) \longrightarrow (C', call(e_1', e_2))}\ \text{(callE1)}$$

$$\frac{(C, e_2) \longrightarrow (C', e_2')}{(C, call(f_1, e_2)) \longrightarrow (C', call(f_1, e_2'))}\ \text{(callE2)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, invoke(e_1, e_2)) \longrightarrow (C', invoke(e_1', e_2))}\ \text{(invokeE1)}$$

$$\frac{(C, e_2) \longrightarrow (C', e_2')}{(C, invoke(s_1, e_2)) \longrightarrow (C', invoke(s_1, e_2'))}\ \text{(invokeE2)}$$

$$\frac{(C, e_1) \longrightarrow (C', e_1')}{(C, if\ e_1\ then\ e_2\ else\ e_3) \longrightarrow (C', if\ e_1'\ then\ e_2\ else\ e_3)}\ \text{(ifE)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, \{\ e\ \}_{s(v)}) \longrightarrow (C', \{\ e'\ \}_{s(v)})}\ \text{(endLabelE)}$$

$$\frac{s \neq \text{“}monitor\text{”}}{((M, R, false, \ldots), \{\ v_1\ \}_{s(v_2)}) \xrightarrow{end_{s(v_2)}:v_1} ((M, R, false, \ldots), v_1)}\ \text{(endLabelValue)}$$

$$\frac{out = in_{none}(unit) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), outputNotSet()) \longrightarrow ((\ldots, out, \tau_{out}), true)}\ \text{(outputNotSetTrue)}$$

$$\frac{out = in_{some}(e) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}, outputNotSet()) \longrightarrow ((\ldots, out, \tau_{out}), false)}\ \text{(outputNotSetFalse)}$$

$$\frac{\forall (s', f) \in F.s_1 \neq s'}{((M, (F, \ldots), \ldots), invoke(s_1, v_2)) \longrightarrow ((M, (F, \ldots), \ldots), in_{none}(unit) : TypedVal\ Option)}\ \text{(invokeValNotExists)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, makeTypedVal(\tau, e)) \longrightarrow (C', makeTypedVal(\tau, e'))}\ \text{(makeTypedValE)}$$

$$\frac{}{(C, tryCast(\tau, makeTypedVal(\tau, v))) \longrightarrow (C, in_{some}(v) : \tau\ Option)}\ \text{(tryCastVOk)}$$

$$\frac{\tau_1 \neq \tau_2}{(C, tryCast(\tau_1, makeTypedVal(\tau_2, v))) \longrightarrow (C, in_{none}(unit) : \tau_1\ Option)}\ \text{(tryCastVBad)}$$

$$\frac{(s_1, fun\ x_1(x_2 : \tau_1) : \tau_2 = e) \in F \quad v_2 = makeTypedVal(\tau_1, v_2')}{\begin{array}{l}((M, (F, \ldots), \ldots), invoke(s_1, v_2)) \longrightarrow \\ ((M, (F, \ldots), \ldots), in_{some}(makeTypedVal(\tau_2, call(fun\ x_1(x_2 : \tau_1) : \tau_2\{e\}, v_2'))) : TypedVal\ Option)\end{array}}\ \text{(invokeValueExistsOk)}$$

$$\frac{(s_1, fun\ x_1(x_2 : \tau_1) : \tau_2 = e) \in F \quad v_2 = makeTypedVal(\tau_3, v_2') \quad \tau_1 \neq \tau_3}{((M, (F, \ldots), \ldots), invoke(s_1, v_2)) \longrightarrow ((M, (F, \ldots), \ldots), in_{none}(unit) : TypedVal\ Option)}\ \text{(invokeValueExistsBad)}$$

$$\frac{}{(C, while(e_1)\ \{e_2\}) \longrightarrow (C, if\ e_1\ then\ (e_2;\ while(e_1)\ \{e_2\})\ else\ false)}\ \text{(whileE)}$$

$$\frac{}{((M \cup \{(\ell, v)\}, \ldots), \ell := v') \longrightarrow ((M \cup \{(\ell, v')\}, \ldots), unit)}\ \text{(assignValue)}$$

$$\frac{out = in_{some}(e) : \tau\ Event\ Option}{((\ldots, out, \tau_{out}), setOutput(v)) \longrightarrow ((\ldots, out, \tau_{out}), false)}\ \text{(setOutputSet)}$$

$$\frac{\forall j (1 \leq j < i).e_j = v_j \quad (C, e_i) \longrightarrow (C', e_i') \quad i \in \{1, \ldots, n\}}{(C, (l_1 = e_1, \ldots, l_n = e_n)) \longrightarrow (C', (l_1 = e_1, \ldots, l_i = e_i', \ldots, l_n = e_n))}\ \text{(recordE)}$$

$$\frac{(C, e_i) \longrightarrow (C', e_i')}{(C, in_{\ell_i}\ e_i : \tau) \longrightarrow (C', in_{\ell_i}\ e_i' : \tau)}\ \text{(variantE)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, (case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n)) \longrightarrow (C', (case\ e'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n))}\ \text{(caseE)}$$

$$\frac{i \in \{1, \ldots, n\}}{(C, (case\ (in_{\ell_i} v_i : \tau)\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n)) \longrightarrow (C, [v_i/x_i]e_i)}\ \text{(caseV)}$$

$$\frac{(C, e) \longrightarrow (C', e')}{(C, monitor(\tau, e)) \longrightarrow (C', monitor(\tau, e'))}\ \text{(monitorE)}$$

$$\frac{}{((M, R, inOb, \ldots), monitor(\tau, v)) \xrightarrow{begin_{monitor(v)}} ((M, R, true, \ldots), \{e_{monitor}(\tau, v)\}_{monitor(v)})}\ \text{(monitorV)}$$

$$\frac{g = makeCFG_{\alpha}(v)}{(C, makeCFG(v)) \xrightarrow{begin_{makeCFG(v)}} (C, \{g\}_{makeCFG(v)})}\ \text{(makeCFGValue)}$$

$$\frac{s = \text{“}monitor\text{”}}{((M, R, inOb, \ldots), \{\ v_1\ \}_{s(v_2)}) \xrightarrow{end_{s(v_2)}:v_1} ((M, R, false, \ldots), v_1)}\ \text{(endLabelValueMonitor)}$$

$$\frac{}{\begin{array}{l}((\ldots, in_{none}(unit) : \tau_{out}\ Event\ Option, \tau_{out}), setOutput(in_{act}(v) : Event)) \longrightarrow \\ ((\ldots, in_{some}(in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option, \tau_{out}), true)\end{array}}\ \text{(setOutputNotSetAct)}$$

$$\frac{}{\begin{array}{l}((\ldots, in_{none}(unit) : \tau_{out}\ Event\ Option, \tau_{out}), setOutput(in_{res}(res(v_1, makeTypedVal(\tau_{out}, v_2))) : Event)) \longrightarrow \\ ((\ldots, in_{some}(in_{res}(res(v_1, v_2)) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option, \tau_{out}), true)\end{array}}\ \text{(setOutputNotSetResGood)}$$

$$\frac{\tau' \neq \tau_{out} \quad out = in_{none}(unit) : \tau_{out}\ Event\ Option}{\begin{array}{l}((\ldots, out, \tau_{out}), setOutput(in_{res}(res(v_1, makeTypedVal(\tau', v_2))) : Event)) \longrightarrow \\ ((\ldots, out, \tau_{out}), false)\end{array}}\ \text{(setOutputNotSetResBad)}$$

$$\frac{out = in_{some}(in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), getOutput()) \longrightarrow ((\ldots, out, \tau_{out}), in_{some}(in_{act}(v) : Event) : Event\ Option)}\ \text{(getOutputSomeAct)}$$

$$\frac{out = in_{some}(in_{res}(res(v_1, v_2)) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), getOutput()) \longrightarrow ((\ldots, out, \tau_{out}), in_{some}(in_{res}(res(v_1, makeTypedVal(\tau_{out}, v_2))) : Event) : Event\ Option)}\ \text{(getOutputSomeRes)}$$

$$\frac{out = in_{none}(unit) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), getOutput()) \longrightarrow ((\ldots, out, \tau_{out}), in_{none}(unit) : Event\ Option)}\ \text{(getOutputNone)}$$

$$\frac{(s, fun\ x_1(x_2 : \tau_1) : \tau_2 = e) \in F \quad f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e)}{\begin{array}{l}((M, (F, \ldots), true, rt, \ldots), call(f, v)) \xrightarrow{begin_{x_1(v)},\ begin_{appendRes()}} \\ \Big((M, (F, \ldots), true, rt\ @ \\ res(act(s, makeTypedVal(\tau_1, v)), makeTypedVal(\tau_2, [f/x_1, v/x_2]e)) :: [] : Res_{List}, \ldots), \{[f/x_1, v/x_2]e\}_{x_1(v)}\Big)\end{array}}\ \text{(callFromObligation)}$$

$$\frac{f \notin range(F) \quad \forall pol \in pols\ (f \neq pol.onTrigger \wedge f \neq pol.onObligation) \quad f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e)}{((M, (F, pols, \ldots), \ldots), call(f, v)) \xrightarrow{begin_{f(v)}} \Big((M, (F, pols, \ldots), \ldots), \{[v/x_2, f/x_1]e\}_{f(v)}\Big)}\ \text{(callNonMonitoredFunction)}$$

$$\frac{(name = s, onTrigger = f_1, onObligation = f_2, vote = f_3) \in pols \quad f_1 = (fun\ x_1(x_2 : Event) : Unit = e)}{((M, (F, pols, os, vc), inOb, rt, out, \tau_{out}), call(f_1, v)) \xrightarrow{begin_{f_1(v)}} \left((M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out}), \{[f_1/x_1, v/x_2]e\}_{f_1(v)}\right)} \text{(callOnTrigger)}$$

$$\frac{(name = s, onTrigger = f_1, onObligation = f_2, vote = f_3) \in pols \quad f_2 = (fun\ x_1(x_2 : Res_{List}) : Unit = e)}{((M, (F, pols, os, vc), inOb, rt, out, \tau_{out}), call(f_2, v)) \xrightarrow{begin_{f_2(v)}} \left((M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out}), \{[f_2/x_1, v/x_2]e\}_{f_2(v)}\right)} \text{(callOnObligation)}$$

$$\frac{(s, f) \in F \quad f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e)}{((M, (F, pols, os, vc), false, rt, out, \tau_{old}), call(f, v) \xrightarrow{begin_{s(v)}} ((M, (F, pols, os, vc), false, rt, in_{none}(unit) : \tau_2\ Event\ Option, \tau_2), e_{procEvt})} \text{(callFromApplication)}$$

Where $e_{procEvt} =$

$\{let\ aux = (fun\ aux(event : \tau_2\ Event) : \tau_2\ Res =$
$case\ event\ of$
$act\ a \Rightarrow$
$case\ invoke(a.name, a.arg)\ of$
$some\ r \Rightarrow$
$case\ tryCast(\tau_2, r_1)\ of$
$some\ v_1 \Rightarrow$
$let\ action\_output = in_{res}\ res(a, v_1) : \tau_2\ Event\ in$
$let\ mon\_output = monitor(\tau_2, (evt = action\_output, pols = pols, os = os, vc = vc))\ in$
$call(aux, mon\_output)$
end
end
$| none\ u_1 \Rightarrow call(aux, event)$
$| none\ u_2 \Rightarrow call(aux, event)$
$| res\ r_2 \Rightarrow r_2)$
in
$call(aux, in_{act}\ act(s, v) : \tau_2\ Event).result$
$end\}_{s(v)}$

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).

$\Lambda \vdash (M, R, inOb, rt, out, \tau_{out})$ **ok** $\Rightarrow M : \Lambda \wedge \Lambda \vdash R$ **ok** $\wedge\, \Lambda, \bullet \vdash inOb : Bool \wedge \Lambda, \bullet \vdash rt : Res_{List} \wedge \Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$

Proof.

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

□

Lemma 2 (R-Inversion). $\Lambda \vdash (F, pols, os, vc)$ **ok** $\Rightarrow \Lambda \vdash F$ **ok** $\wedge\, \Lambda, \bullet \vdash pols : Pol_{List} \wedge \Lambda, \bullet \vdash os : OS \wedge \Lambda, \bullet \vdash vc : VC$

Proof.

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

□

Lemma 3 (C-Weakening).

$\Lambda \vdash (M, (F, pols, os, vc), inOb, rt, out, \tau_{out})$ **ok** $\wedge\, M' : \Lambda' \wedge\ \Lambda \subseteq \Lambda' \Rightarrow \Lambda' \vdash (M', (F, pols, os, vc), inOb, rt, out, \tau_{out})$ **ok**

Proof.

1.	$\Lambda \vdash (M, (F, pols, os, vc), inOb, rt)$ **ok**	assumption
2.	$M' : \Lambda'$	assumption
3.	$\Lambda \subseteq \Lambda'$	assumption
4.	$\Lambda \vdash (F, pols, os, vc)$ **ok**	1, C-Inversion Lemma
5.	$\Lambda \vdash F$ **ok**	4, R-Inversion Lemma
6.	$\Lambda, \bullet \vdash pols : Pol_{List}$	4, R-Inversion Lemma
7.	$\Lambda, \bullet \vdash os : OS$	4, R-Inversion Lemma
8.	$\Lambda, \bullet \vdash vc : VC$	4, R-Inversion Lemma
9.	$\Lambda, \bullet \vdash inOb : Bool$	1, C-Inversion Lemma
10.	$\Lambda, \bullet \vdash rt : Res_{List}$	1, C-Inversion Lemma
11.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, C-Inversion Lemma
12.	$\Lambda', \bullet \vdash pols : Pol_{List}$	3, 6, Lemma Λ-Weakening
13.	$\Lambda', \bullet \vdash os : OS$	3, 7, Lemma Λ-Weakening
14.	$\Lambda', \bullet \vdash vc : VC$	3, 8, Lemma Λ-Weakening
15.	$\Lambda', \bullet \vdash inOb : Bool$	3, 9, Lemma Λ-Weakening
16.	$\Lambda', \bullet \vdash rt : Res_{List}$	3, 10, Lemma Λ-Weakening
17.	$\Lambda', \bullet \vdash out : \tau_{out}\ Event\ Option$	3, 11, Lemma Λ-Weakening
18.	(5) is only derivable with Rule F-ok	Inspection of $\Lambda \vdash F$ **ok** rules
19.	$F = \{(s_1, f_1), \ldots, (s_n, f_n)\}$	5, 18, Inversion of Rule F-ok
20.	$\forall i \in \{1, \ldots, n\}.\exists \tau_1, \tau_2.\Lambda, \bullet \vdash f_i : \tau_1 \to \tau_2$	5, 18, Inversion of Rule F-ok
21.	$\forall i \in \{1, \ldots, n\}.\Lambda, \bullet \vdash s_i : String$	5, 18, Inversion of Rule F-ok
22.	$\forall i \in \{1, \ldots, n\}.\exists \tau_1, \tau_2.\Lambda', \bullet \vdash f_i : \tau_1 \to \tau_2$	3, 20, Lemma Λ-Weakening
23.	$\forall i \in \{1, \ldots, n\}.\Lambda', \bullet \vdash s_i : String$	3, 21, Lemma Λ-Weakening
22.	$\Lambda' \vdash F$ **ok**	19, 22, 23, Rule F-ok
23.	$\Lambda' \vdash (F, pols, os, vc))$ **ok**	12-14, 22, Rule R-ok
24.	$\Lambda' \vdash (M, (F, pols, os, vc), inOb, rt, out, \tau_{out})$ **ok**	2, 15-17, 23, Rule C-ok

□

Lemma 4 (Λ-Weakening). $(\Lambda_1, \Gamma \vdash e : \tau \wedge \Lambda_1 \subseteq \Lambda_2) \Rightarrow \Lambda_2, \Gamma \vdash e : \tau$

Proof. By induction on the derivation of $\Lambda_1, \Gamma \vdash e : \tau$

Case $\dfrac{}{\Lambda_1, \Gamma \vdash n : Int}$ (intVal)

1. $\Lambda_2, \Gamma \vdash n : Int$ Rule intVal

Case $\dfrac{}{\Lambda_1, \Gamma \vdash b : Bool}$ (boolVal)

1. $\Lambda_2, \Gamma \vdash b : Bool$ Rule boolVal

Case $\dfrac{}{\Lambda_1, \Gamma \vdash s : String}$ (stringVal)

1. $\Lambda_2, \Gamma \vdash s : String$ Rule stringVal

Case $\dfrac{}{\Lambda_1, \Gamma \vdash unit : Unit}$ (unitVal)

1. $\Lambda_2, \Gamma \vdash unit : Unit$ Rule unitVal

Case $\dfrac{\Lambda_1, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e : \tau_2}{\Lambda_1, \Gamma \vdash (fun\ x_1(x_2 : \tau_1) : \tau_2 = e) : \tau_1 \to \tau_2}$ (fun)

1.	$\Lambda_1, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e : \tau_2$	assumption
2.	$\Lambda_1 \subseteq \Lambda_2$	assumption
3.	$\Lambda_2, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e : \tau_2$	1, 2, IH
4.	$\Lambda_2, \Gamma \vdash (fun\ x_1(x_2 : \tau_1) : \tau_2 = e) : \tau_1 \to \tau_2$	3, Rule fun

Case $\dfrac{}{(\Lambda_1' \cup \{\ell : \tau\}), \Gamma \vdash \ell : \tau\ Ref}$ (location)

1.	$\Lambda_1 = \Lambda_1' \cup \{\ell : \tau\}$	assumption
2.	$\Lambda_1 \subseteq \Lambda_2$	assumption
3.	$\ell : \tau \in \Lambda_2$	1, 2, Definition of $\subseteq$
4.	$\Lambda_2 = \Lambda_2' \cup \{\ell : \tau\}$	3
5.	$\Lambda_2, \Gamma \vdash \ell : \tau\ Ref$	4, Rule location

Case $\dfrac{}{\Lambda_1, \Gamma' \cup \{x : \tau\} \vdash x : \tau}$ (var)

1. $\Lambda_2, \Gamma' \cup \{x : \tau\} \vdash x : \tau$ Rule var

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : Bool \quad \Lambda_1, \Gamma \vdash e_2 : Bool}{\Lambda_1, \Gamma \vdash e_1 \wedge e_2 : Bool}$ (con)

1.	$\Lambda_1, \Gamma \vdash e_1 : Bool$	assumption
2.	$\Lambda_1, \Gamma \vdash e_2 : Bool$	assumption
3.	$\Lambda_1 \subseteq \Lambda_2$	assumption
4.	$\Lambda_2, \Gamma \vdash e_1 : Bool$	1, 3, IH
5.	$\Lambda_2, \Gamma \vdash e_2 : Bool$	2, 3, IH
6.	$\Lambda_2, \Gamma \vdash e_1 \wedge e_2 : Bool$	4, 5, Rule con

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : Bool \quad \Lambda_1, \Gamma \vdash e_2 : Bool}{\Lambda_1, \Gamma \vdash e_1 \vee e_2 : Bool}$ (dis)

1. $\Lambda_1, \Gamma \vdash e_1 : Bool$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : Bool$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : Bool$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : Bool$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1 \vee e_2 : Bool$ 4, 5, Rule dis

Case $\dfrac{\Lambda_1, \Gamma \vdash e : Bool}{\Lambda_1, \Gamma \vdash \neg e : Bool}$ (negation)

1. $\Lambda_1, \Gamma \vdash e : Bool$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : Bool$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash \neg e : Bool$ 3, Rule negation

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau \quad \Lambda_1, \Gamma \vdash e_2 : \tau \quad \tau \in \{Int, Bool, String\}}{\Lambda_1, \Gamma \vdash e_1 == e_2 : Bool}$ (equality)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau$ assumption
3. $\tau \in \{Int, Bool, String\}$ assumption
4. $\Lambda_1 \subseteq \Lambda_2$ assumption
5. $\Lambda_2, \Gamma \vdash e_1 : \tau$ 1, 4, IH
6. $\Lambda_2, \Gamma \vdash e_2 : \tau$ 2, 4, IH
7. $\Lambda_2, \Gamma \vdash e_1 == e_2 : Bool$ 3, 5, 6, Rule equality

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : Int \quad \Lambda_1, \Gamma \vdash e_2 : Int}{\Lambda_1, \Gamma \vdash e_1 + e_2 : Int}$ (add)

1. $\Lambda_1 \subseteq \Lambda_2$ assumption
2. $\Lambda_1, \Gamma \vdash e_1 : Int$ assumption
3. $\Lambda_1, \Gamma \vdash e_2 : Int$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : Int$ 1, 2, IH
5. $\Lambda_2, \Gamma \vdash e_2 : Int$ 1, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1 + e_2 : Int$ 4, 5, Rule add

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau_1 \quad \Lambda_1, \Gamma \vdash e_2 : \tau_2}{\Lambda_1, \Gamma \vdash e_1;\ e_2 : \tau_2}$ (sequence)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau_1$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau_2$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau_1$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau_2$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1;\ e_2 : \tau_2$ 4, 5, Rule sequence

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : Bool \quad \Lambda_1, \Gamma \vdash e_2 : \tau \quad \Lambda_1, \Gamma \vdash e_3 : \tau}{\Lambda_1, \Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}$ (if)

1. $\Lambda_1, \Gamma \vdash e_1 : Bool$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau$ assumption
3. $\Lambda_1, \Gamma \vdash e_3 : \tau$ assumption
4. $\Lambda_1 \subseteq \Lambda_2$ assumption
5. $\Lambda_2, \Gamma \vdash e_1 : Bool$ 1, 4, IH
6. $\Lambda_2, \Gamma \vdash e_2 : \tau$ 2, 4, IH
7. $\Lambda_2, \Gamma \vdash e_3 : \tau$ 3, 4, IH
8. $\Lambda_2, \Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau$ 5-7, Rule if

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : Bool \quad \Lambda_1, \Gamma \vdash e_2 : \tau}{\Lambda_1, \Gamma \vdash while(e_1)\ \{e_2\} : Bool}$ (while)

1. $\Lambda_1, \Gamma \vdash e_1 : Bool$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : Bool$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash while(e_1)\ \{e_2\} : Bool$ 4, 5, Rule while

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau_1 \quad \Lambda_1, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Lambda_1, \Gamma \vdash let\ x = e_1\ in\ e_2\ end : \tau_2}$ (let)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau_1$ assumption
2. $\Lambda_1, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau_1$ 1, 3, IH
5. $\Lambda_2, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash let\ x = e_1\ in\ e_2\ end : \tau_2$ 4, 5, Rule let

Case $\dfrac{\Lambda_1, \Gamma \vdash e : \tau}{\Lambda_1, \Gamma \vdash ref\ e : \tau\ Ref}$ (createRef)

1. $\Lambda_1, \Gamma \vdash e : \tau$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : \tau$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash ref\ e : \tau\ Ref$ 3, Rule createRef

Case $\dfrac{\Lambda_1, \Gamma \vdash e : \tau\ Ref}{\Lambda_1, \Gamma \vdash !e : \tau}$ (accessRef)

1. $\Lambda_1, \Gamma \vdash e : \tau\ Ref$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : \tau\ Ref$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash !e : \tau$ 3, Rule accessRef

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau\ Ref \qquad \Lambda_1, \Gamma \vdash e_2 : \tau}{\Lambda_1, \Gamma \vdash e_1\ :=\ e_2 : Unit}$ (assignment)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau\ Ref$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau\ Ref$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1\ :=\ e_2 : Unit$ 4, 5, Rule assignment

Case $\dfrac{}{\Lambda_1, \Gamma \vdash ([] : \tau_{List}) : \tau_{List}}$ (listEmptyVal)

1. $\Lambda_2, \Gamma \vdash ([] : \tau_{List}) : \tau_{List}$ Rule listEmptyVal

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau \quad \Lambda_1, \Gamma \vdash e_2 : \tau_{List}}{\Lambda_1, \Gamma \vdash e_1 :: e_2 : \tau_{List}}$ (listCons)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau_{List}$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau_{List}$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1 :: e_2 : \tau_{List}$ 4, 5, Rule listCons

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau_{List} \quad \Lambda_1, \Gamma \vdash e_2 : \tau_{List}}{\Lambda_1, \Gamma \vdash e_1 \mathbin{@} e_2 : \tau_{List}}$ (listAppend)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau_{List}$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau_{List}$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau_{List}$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau_{List}$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1 \mathbin{@} e_2 : \tau_{List}$ 4, 5, Rule listAppend

Case $\dfrac{\Lambda_1, \Gamma \vdash e : \tau_{List}}{\Lambda_1, \Gamma \vdash head(e) : \tau\ Option}$ (head)

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

Case $\dfrac{\Lambda_1, \Gamma \vdash e : \tau_{List}}{\Lambda_1, \Gamma \vdash tail(e) : \tau_{List}}$ (tail)

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

Case $\dfrac{\Lambda_1, \Gamma \vdash e : \tau_{List}}{\Lambda_1, \Gamma \vdash empty(e) : Bool}$ (empty)

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

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau_1 \ \cdots \ \Lambda_1, \Gamma \vdash e_n : \tau_n}{\Lambda_1, \Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}$ (record)

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

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

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

Case $\dfrac{\Lambda_1, \Gamma \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda_1, \Gamma \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)}$ (variant)

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

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : TypedVal}{\Lambda_1, \Gamma \vdash tryCast(\tau, e_1) : \tau\ Option}$ (tryCast)

1. $\Lambda_1, \Gamma \vdash e_1 : TypedVal$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e_1 : TypedVal$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash tryCast(\tau, e_1) : \tau\ Option$ 3, Rule tryCast

Case $\dfrac{\Lambda_1, \Gamma \vdash e : \tau}{\Lambda_1, \Gamma \vdash \{\ e\ \}_{s(v)} : \tau}$ (endLabel)

1. $\Lambda_1, \Gamma \vdash e : \tau$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : \tau$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash \{\ e\ \}_{s(v)} : \tau$ 3, Rule endLabel

Case $\dfrac{}{\Lambda_1, \Gamma \vdash getRT() : Res_{List}}$ (getRT)

1. $\Lambda_2, \Gamma \vdash getRT() : Res_{List}$ Rule getRT

Case $\dfrac{\Lambda_1, \Gamma \vdash e : Obligation}{\Lambda_1, \Gamma \vdash makeCFG(e) : CFG}$ (makeCFG)

1. $\Lambda_1, \Gamma \vdash e : Obligation$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : Obligation$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash makeCFG(e) : CFG$ 3, Rule makeCFG

Case $\dfrac{\Lambda_1, \Gamma \vdash e : Event}{\Lambda_1, \Gamma \vdash setOutput(e) : Bool}$ (setOutput)

1. $\Lambda_1, \Gamma \vdash e : Event$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : Event$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash setOutput(e) : Bool$ 3, Rule setOutput

Case $\dfrac{}{\Lambda_1, \Gamma \vdash outputNotSet() : Bool}$ (outputNotSet)

1. $\Lambda_2, \Gamma \vdash outputNotSet() : Bool$ Rule outputNotSet

Case $\dfrac{}{\Lambda_1, \Gamma \vdash getOutput() : Event\ Option}$ (getOutput)

1. $\Lambda_2, \Gamma \vdash getOutput() : Event\ Option$ Rule getOutput

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : String \quad \Lambda_1, \Gamma \vdash e_2 : TypedVal}{\Lambda_1, \Gamma \vdash invoke(e_1, e_2) : TypedVal\ Option}$ (invoke)

1. $\Lambda_1, \Gamma \vdash e_1 : String$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : TypedVal$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : String$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : TypedVal$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash invoke(e_1, e_2) : TypedVal\ Option$ 4, 5, Rule invoke

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda_1, \Gamma \vdash e_2 : \tau_1}{\Lambda_1, \Gamma \vdash call(e_1, e_2) : \tau_2}$ (call)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau_1 \to \tau_2$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau_1$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau_1 \to \tau_2$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau_1$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash call(e_1, e_2) : \tau_2$ 4, 5, Rule call

Case $\dfrac{\Lambda_1, \Gamma \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda_1, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda_1, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau}{\Lambda_1, \Gamma \vdash (case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau}$ (case)

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

Case $\dfrac{\Lambda_1, \Gamma \vdash e_1 : \tau}{\Lambda_1, \Gamma \vdash makeTypedVal(\tau, e_1) : TypedVal}$ (makeTypedVal)

1. $\Lambda_1, \Gamma \vdash e_1 : \tau$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e_1 : \tau$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash makeTypedVal(\tau, e_1) : TypedVal$ 3, Rule makeTypedVal

Case $\dfrac{\Lambda_1, \Gamma \vdash e' : (evt : \tau'\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda_1, \Gamma \vdash monitor(\tau', e') : \tau'\ Event}$ (monitor)

1. $\Lambda_1, \Gamma \vdash e' : (evt : \tau'\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e' : (evt : \tau'\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash monitor(\tau', e') : \tau'\ Event$ 3, Rule monitor

□

Lemma 5 (Weakening). $\Lambda, \Gamma_1 \vdash e : \tau \wedge \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Lambda, \Gamma_2 \vdash e : \tau$

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

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash n : Int}$ (intVal)

1. $\Lambda, \Gamma_2 \vdash n : Int$ Rule intVal

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash b : Bool}$ (boolVal)

1. $\Lambda, \Gamma_2 \vdash b : Bool$ Rule boolValue

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash s : String}$ (stringVal)

1. $\Lambda, \Gamma_2 \vdash s : String$ Rule stringVal

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash unit : Unit}$ (unitVal)

1. $\Lambda, \Gamma_2 \vdash unit : Unit$ Rule unitVal

Case $\dfrac{\Lambda, \Gamma_1 \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e' : \tau_2}{\Lambda, \Gamma_1 \vdash fun\ x_1(x_2 : \tau_1) : \tau_2 = e' : \tau_1 \to \tau_2}$ (fun)

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

Case $\dfrac{}{\Lambda' \cup \{\ell : \tau\}, \Gamma_1 \vdash \ell : \tau\ Ref}$ (location)

1. $\Lambda' \cup \{\ell : \tau\}, \Gamma_2 \vdash \ell : \tau\ Ref$ Rule location

Case $\dfrac{}{\Lambda, \Gamma_1' \cup \{x : \tau\} \vdash x : \tau}$ (var)

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

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : Bool \quad \Lambda, \Gamma_1 \vdash e_2 : Bool}{\Lambda, \Gamma_1 \vdash e_1 \wedge e_2 : Bool}$ (con)

1. $\Lambda, \Gamma_1 \vdash e_1 : Bool$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : Bool$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Bool$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : Bool$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 \wedge e_2 : Bool$ 4, 5, rule con

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : Bool \quad \Lambda, \Gamma_1 \vdash e_2 : Bool}{\Lambda, \Gamma_1 \vdash e_1 \vee e_2 : Bool}$ (dis)

1. $\Lambda, \Gamma_1 \vdash e_1 : Bool$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : Bool$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Bool$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : Bool$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 \vee e_2 : Bool$ 4, 5, Rule dis

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : Bool}{\Lambda, \Gamma_1 \vdash \neg e : Bool}$ (negation)

1. $\Lambda, \Gamma_1 \vdash e : Bool$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : Bool$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash \neg e : Bool$ 3, Rule negation

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau \quad \Lambda, \Gamma_1 \vdash e_2 : \tau \quad \tau \in \{Int, Bool, String\}}{\Lambda, \Gamma_1 \vdash e_1 == e_2 : Bool}$ (equality)

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

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : Int \quad \Lambda, \Gamma_1 \vdash e_2 : Int}{\Lambda, \Gamma_1 \vdash e_1 + e_2 : Int}$ (add)

1. $\Lambda, \Gamma_1 \vdash e_1 : Int$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : Int$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Int$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : Int$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 + e_2 : Int$ 4, 5, Rule add

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \Lambda, \Gamma_1 \vdash e_2 : \tau_2}{\Lambda, \Gamma_1 \vdash e_1;\ e_2 : \tau_2}$ (sequence)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau_1$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau_2$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : \tau_1$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \tau_2$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1;\ e_2 : \tau_2$ 4, 5, Rule sequence

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : Bool \quad \Lambda, \Gamma_1 \vdash e_2 : \tau \quad \Lambda, \Gamma_1 \vdash e_3 : \tau}{\Lambda, \Gamma_1 \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}$ (if)

1. $\Lambda, \Gamma_1 \vdash e_1 : Bool$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau$ assumption
3. $\Lambda, \Gamma_1 \vdash e_3 : \tau$ assumption
4. $\Gamma_1 \subseteq \Gamma_2$ assumption
5. $\Lambda, \Gamma_2 \vdash e_1 : Bool$ 1, 4, IH
6. $\Lambda, \Gamma_2 \vdash e_2 : \tau$ 2, 4, IH
7. $\Lambda, \Gamma_2 \vdash e_3 : \tau$ 3, 4, IH
8. $\Lambda, \Gamma_2 \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau$ 5, 6, 7, Rule If

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : Bool \quad \Lambda, \Gamma_1 \vdash e_2 : \tau}{\Lambda, \Gamma_1 \vdash while(e_1)\ \{e_2\} : Bool}$ (while)

1. $\Lambda, \Gamma_1 \vdash e_1 : Bool$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : Bool$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \tau$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash while(e_1)\ \{e_2\} : Bool$ 4, 5, Rule while

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \Lambda, \Gamma_1 \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Lambda, \Gamma_1 \vdash let\ x = e_1\ in\ e_2\ end : \tau_2}$ (let)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau_1$ assumption
2. $\Lambda, \Gamma_1 \cup \{x : \tau_1\} \vdash e_2 : \tau_2$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Gamma_1 \cup \{x : \tau_1\} \subseteq \Gamma_2 \cup \{x : \tau_1\}$ 3, definition of $\subseteq$
5. $\Lambda, \Gamma_2 \vdash e_1 : \tau_1$ 1, 3, IH
6. $\Lambda, \Gamma_2 \cup \{x : \tau_1\} \vdash e_2 : \tau_2$ 2, 4, IH
7. $\Lambda, \Gamma_2 \vdash let\ x = e_1\ in\ e_2\ end : \tau_2$ 5, 6, Rule let

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : \tau}{\Lambda, \Gamma_1 \vdash ref\ e : \tau\ Ref}$ (createRef)

1. $\Lambda, \Gamma_1 \vdash e : \tau$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : \tau$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash ref\ e : \tau\ Ref$ 3, Rule createRef

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : \tau\ Ref}{\Lambda, \Gamma_1 \vdash !e : \tau}$ (accessRef)

1. $\Lambda, \Gamma_1 \vdash e : \tau\ Ref$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : \tau\ Ref$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash !e : \tau$ 3, Rule accessRef

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau\ Ref \qquad \Lambda, \Gamma_1 \vdash e_2 : \tau}{\Lambda, \Gamma_1 \vdash e_1\ :=\ e_2 : unit}$ (assignment)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau\ Ref$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : \tau\ Ref$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \tau$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1\ :=\ e_2 : unit$ 4, 5, Rule assignment

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash ([] : \tau_{List}) : \tau_{List}}$ (listEmptyVal)

1. $\Lambda, \Gamma_2 \vdash ([] : \tau_{List}) : \tau_{List}$ Rule listEmptyVal

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau \quad \Lambda, \Gamma_1 \vdash e_2 : \tau_{List}}{\Lambda, \Gamma_1 \vdash e_1 :: e_2 : \tau_{List}}$ (listCons)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau_{List}$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : \tau$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \tau_{List}$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 :: e_2 : \tau_{List}$ 4, 5, Rule listCons

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau_{List} \quad \Lambda, \Gamma_1 \vdash e_2 : \tau_{List}}{\Lambda, \Gamma_1 \vdash e_1 \mathbin{@} e_2 : \tau_{List}}$ (listAppend)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau_{List}$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau_{List}$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : \tau_{List}$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \tau_{List}$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 \mathbin{@} e_2 : \tau_{List}$ 4, 5, Rule listAppend

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : \tau_{List}}{\Lambda, \Gamma_1 \vdash head(e) : \tau\ Option}$ (head)

1. $\Lambda, \Gamma_1 \vdash e : \tau_{List}$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : \tau_{List}$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash head(e) : \tau\ Option$ 3, Rule head

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : \tau_{List}}{\Lambda, \Gamma_1 \vdash tail(e) : \tau_{List}}$ (tail)

1. $\Lambda, \Gamma_1 \vdash e : \tau_{List}$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : \tau_{List}$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash tail(e) : \tau_{List}$ 3, Rule tail

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_{List}}{\Lambda, \Gamma \vdash empty(e) : Bool}$ (empty)

1. $\Lambda, \Gamma_1 \vdash e : \tau_{List}$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : \tau_{List}$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash empty(e) : Bool$ 3, Rule empty

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_1 \vdash e_n : \tau_n}{\Lambda, \Gamma_1 \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}$ (record)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma_1 \vdash e_n : \tau_n$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $\Lambda, \Gamma_2 \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma_2 \vdash e_n : \tau_n$ — 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \ldots \times \ell_n : \tau_n)$ — 3, Rule record

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\}}{\Gamma_1 \vdash e.\ell_i : \tau_i}$ (projection)

1. $\Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $\Lambda, \Gamma_2 \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$ — 1, 2, IH
4. $i \in \{1, \ldots, n\}$ — assumption
5. $\Lambda, \Gamma_2 \vdash e.\ell_i : \tau_i$ — 3, 4, Rule projection

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma_1 \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}$ (variant)

1. $\Lambda, \Gamma_1 \vdash e_i : \tau_i$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $i \in \{1, \ldots, n\}$ — assumption
4. $\Lambda, \Gamma_2 \vdash e_i : \tau_i$ — 1, 2, IH
5. $\Lambda, \Gamma_2 \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — 3, 4, Rule variant

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda, \Gamma_1 \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \Gamma_1 \cup \{x_n : \tau_n\} \vdash e_n : \tau}{\Lambda, \Gamma_1 \vdash (case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau}$ (case)

1. $\Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — assumption
2. $\Lambda, \Gamma_1 \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau$ — assumption
3. $\Gamma_1 \subseteq \Gamma_2$ — assumption
4. $\forall i : \Gamma_1 \cup \{x_i : \tau_i\} \subseteq \Gamma_2 \cup \{x_i : \tau_i\}$ — 3, definition of $\subseteq$
5. $\Lambda, \Gamma_2 \vdash e : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — 1, 3, IH
6. $\Lambda, \Gamma_2 \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \Gamma_2 \cup \{x_n : \tau_n\} \vdash e_n : \tau$ — 2, 4, IH
7. $\Lambda, \Gamma_2 \vdash (case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau$ — 5, 6, Rule case

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau}{\Lambda, \Gamma_1 \vdash makeTypedVal(\tau, e_1) : TypedVal}$ (makeTypedVal)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $\Lambda, \Gamma_2 \vdash e_1 : \tau$ — 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash makeTypedVal(\tau, e_1) : TypedVal$ — 3, Rule makeTypedVal

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : TypedVal}{\Lambda, \Gamma_1 \vdash tryCast(\tau, e_1) : \tau\ Option}$ (tryCast)

1. $\Lambda, \Gamma_1 \vdash e_1 : TypedVal$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $\Lambda, \Gamma_2 \vdash e_1 : TypedVal$ — 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash tryCast(\tau, e_1) : \tau\ Option$ — 3, Rule tryCast

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : \tau}{\Lambda, \Gamma_1 \vdash \{\ e\ \}_{label} : \tau}$ (endLabel)

1. $\Lambda, \Gamma_1 \vdash e : \tau$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $\Lambda, \Gamma_2 \vdash e : \tau$ — 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash \{\ e\ \}_{label} : \tau$ — 3, Rule endLabel

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash getRT() : Res_{List}}$ (getRT)

1. $\Lambda, \Gamma_2 \vdash getRT() : Res_{List}$ — Rule getRT

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : Obligation}{\Lambda, \Gamma_1 \vdash makeCFG(e) : CFG}$ (makeCFG)

1. $\Lambda, \Gamma_1 \vdash e : Obligation$ — assumption
2. $\Gamma_1 \subseteq \Gamma_2$ — assumption
3. $\Lambda, \Gamma_2 \vdash e : Obligation$ — 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash makeCFG(e) : CFG$ — 3, Rule makeCFG

Case $\dfrac{\Lambda, \Gamma_1 \vdash e : Event}{\Lambda, \Gamma_1 \vdash setOutput(e) : Bool}$ (setOutput)

1. $\Lambda, \Gamma_1 \vdash e : Event$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e : Event$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash setOutput(e) : Bool$ 3, Rule setOutput

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash outputNotSet() : Bool}$ (outputNotSet)

1. $\Lambda, \Gamma_2 \vdash outputNotSet() : Bool$ Rule outputNotSet

Case $\dfrac{}{\Lambda, \Gamma_1 \vdash getOutput() : Event\ Option}$ (getOutput)

1. $\Lambda, \Gamma_2 \vdash getOutput() : Event\ Option$ Rule getOutput

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda, \Gamma_1 \vdash monitor(\tau_1, e_1) : \tau_1\ Event}$ (monitor)

1. $\Lambda, \Gamma_1 \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$ assumption
2. $\Gamma_1 \subseteq \Gamma_2$ assumption
3. $\Lambda, \Gamma_2 \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$ 1, 2, IH
4. $\Lambda, \Gamma_2 \vdash monitor(\tau_1, e_1) : \tau_1\ Event$ 3, Rule monitor

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : String \quad \Lambda, \Gamma_1 \vdash e_2 : TypedVal}{\Lambda, \Gamma_1 \vdash invoke(e_1, e_2) : TypedVal}$ (invoke)

1. $\Lambda, \Gamma_1 \vdash e_1 : String$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : TypedVal$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : String$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : TypedVal$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash invoke(e_1, e_2) : TypedVal$ 4, 5, Rule invoke

Case $\dfrac{\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda, \Gamma_1 \vdash e_2 : \tau_1}{\Lambda, \Gamma_1 \vdash call(e_1, e_2) : \tau_2}$ (call)

1. $\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau_1$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : \tau_1 \to \tau_2$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \tau_1$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash call(e_1, e_2) : \tau_2$ 4, 5, Rule call

□

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

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_s : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda, \Gamma \cup \{x : \tau, x_1 : \tau_1\} \vdash e_1 : \tau' \ \cdots \ \Lambda, \Gamma \cup \{x : \tau, x_n : \tau_n\} \vdash e_n : \tau'}{\Lambda, \Gamma \cup \{x : \tau\} \vdash (case\ e_s\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau'}$ (case)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_s : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ assumption
2. $\Lambda, \Gamma \cup \{x : \tau, x_1 : \tau_1\} \vdash e_1 : \tau' \ \cdots \ \Gamma \cup \{x : \tau, x_n : \tau_n\} \vdash e_n : \tau'$ assumption
3. $\Lambda, \Gamma \vdash e : \tau$ assumption
4. $\Lambda, \Gamma \vdash [e/x]e_s : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ 1, 3, IH
5. $\Lambda, \Gamma \cup \{x_1 : \tau_1\} \vdash [e/x]e_1 : \tau' \ \cdots \ \Lambda, \Gamma \cup \{x_n : \tau_n\} \vdash [e/x]e_n : \tau'$ 2, 3, IH
6. $\Lambda, \Gamma \vdash (case\ [e/x]e_s\ of\ \ell_1\ x_1 \Rightarrow [e/x]e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow [e/x]e_n) : \tau'$ 4, 5, Rule case
7. $\Lambda, \Gamma \vdash [e/x](case\ e_s\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau'$ 6, definition of $[e/x](case\ e_s\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : String \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : TypedVal}{\Lambda, \Gamma \cup \{x : \tau\} \vdash invoke(e_1, e_2) : TypedVal\ Option}$ (invoke)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : String$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : TypedVal$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : String$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : TypedVal$ — 2, 3, IH
6. $\Lambda, \Gamma \vdash invoke([e/x]e_1, [e/x]e_2) : TypedVal\ Option$ — 4, 5, Rule invoke
7. $\Lambda, \Gamma \vdash [e/x](invoke(e_1, e_2)) : TypedVal\ Option$ — 6, definition of $[e/x]invoke(e_1, e_2)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1}{\Lambda, \Gamma \cup \{x : \tau\} \vdash makeTypedVal(\tau_1, e_1) : TypedVal}$ (makeTypedVal)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1$ — 1, 2, IH
4. $\Lambda, \Gamma \vdash makeTypedVal(\tau_1, [e/x]e_1) : TypedVal$ — 3, Rule makeTypedVal
5. $\Lambda, \Gamma \vdash [e/x]makeTypedVal(\tau_1, e_1) : TypedVal$ — 4, definition of $[e/x]makeTypedVal(\tau_1, e_1)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : TypedVal}{\Lambda, \Gamma \cup \{x : \tau\} \vdash tryCast(\tau_1, e_1) : \tau_1\ Option}$ (tryCast)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : TypedVal$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $\Lambda, \Gamma \vdash [e/x]e_1 : TypedVal$ — 1, 2, IH
4. $\Lambda, \Gamma \vdash tryCast(\tau_1, [e/x]e_1) : \tau_1\ Option$ — 3, Rule tryCast
5. $\Lambda, \Gamma \vdash [e/x]tryCast(\tau_1, e_1) : \tau_1\ Option$ — 4, definition of $[e/x]tryCast(\tau_1, e_1)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \ \cdots \ \Lambda, \Gamma \cup \{x : \tau\} \vdash e_n : \tau_n}{\Lambda, \Gamma \cup \{x : \tau\} \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : \ (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}$ (record)

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda, \Gamma \cup \{x : \tau\} \vdash monitor(\tau_1, e_1) : \tau_1\ Event}$ (monitor)

1. $\tau' = (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau'$ — 1, assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau'$ — 2, 3, IH
5. $\Lambda, \Gamma \vdash monitor(\tau_1, [e/x]e_1) : \tau_1\ Event$ — 4, Rule monitor
6. $\Lambda, \Gamma \vdash [e/x]monitor(\tau_1, e_1) : \tau_1\ Event$ — 5, definition of $[e/x]monitor(\tau_1, e_1)$

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash b : Bool}$ (boolVal)

1. $\tau' = Bool$ — assumption
2. $[e/x]b = b$ — definition of $[e/x]b$
3. $\Lambda, \Gamma \vdash [e/x]b : \tau'$ — 1, 2, Rule boolVal

Case $\dfrac{}{\Lambda' \cup \{\ell : \tau''\}, \Gamma \cup \{x : \tau\} \vdash \ell : \tau''\ Ref}$ (location)

1. $\tau' = \tau''\ Ref$ — assumption
2. $[e/x]\ell = \ell$ — definition of $[e/x]\ell$
3. $\Lambda' \cup \{\ell : \tau\}, \Gamma \vdash [e/x]\ell : \tau'$ — 1, 2, Rule location

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash getRT() : Res_{List}}$ (getRT)

1. $\tau' = Res_{List}$ assumption
2. $[e/x]getRT() = getRT()$ definition of $[e/x]getRT()$
3. $\Lambda, \Gamma \vdash [e/x]getRT() : \tau'$ 1, 2, Rule getRT

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Bool}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \neg e_1 : Bool}$ (negation)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Bool$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e_1 : Bool$ 1, 2, IH
4. $\Lambda, \Gamma \vdash \neg([e/x]e_1) : Bool$ 3, Rule negation
5. $\Lambda, \Gamma \vdash [e/x](\neg e_1) : Bool$ 4, definition of $[e/x]e'$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Bool \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : Bool}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 \vee e_2 : Bool}$ (dis)

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Bool \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : Bool}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 \wedge e_2 : Bool}$ (con)

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau'' \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau'' \quad \tau'' \in \{Int, Bool, String\}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 == e_2 : Bool}$ (equality)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau''$ assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau''$ assumption
3. $\Lambda, \Gamma \vdash e : \tau$ assumption
4. $\tau'' \in \{Int, Bool, String\}$ assumption
5. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau''$ 1, 3, IH
6. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau''$ 2, 3, IH
7. $\Lambda, \Gamma \vdash [e/x]e_1 == [e/x]e_2 : Bool$ 4, 5, 6, Rule equality
8. $\Lambda, \Gamma \vdash [e/x](e_1 == e_2) : Bool$ 7, definition of $[e/x](e_1 == e_2)$

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash n : Int}$ (intVal)

1. $\tau' = Int$ assumption
2. $[e/x]n = n$ definition of $[e/x]n$
3. $\Lambda, \Gamma \vdash [e/x]n : \tau'$ 1, 2, Rule intVal

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Bool \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau' \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_3 : \tau'}{\Lambda, \Gamma \cup \{x : \tau\} \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau'}$ (if)

1. $e' = if\ e_1\ then\ e_2\ else\ e_3$ assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Bool$ assumption
3. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau'$ assumption
4. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_3 : \tau'$ assumption
5. $\Lambda, \Gamma \vdash e : \tau$ assumption
6. $\Lambda, \Gamma \vdash [e/x]e_1 : Bool$ 2, 5, IH
7. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau'$ 3, 5, IH
8. $\Lambda, \Gamma \vdash [e/x]e_3 : \tau'$ 4, 5, IH
9. $\Lambda, \Gamma \vdash (if\ [e/x]e_1\ then\ [e/x]e_2\ else\ [e/x]e_3) : \tau'$ 6-8, Rule if
10. $\Lambda, \Gamma \vdash [e/x]e' : \tau'$ 1, 9, definition of $[e/x]e'$

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash s : String}$ (stringVal)

1. $\tau' = String$ assumption
2. $[e/x]s = s$ definition of $[e/x]s$
3. $\Lambda, \Gamma \vdash [e/x]s : \tau'$ 1, 2, Rule stringVal

Case $\dfrac{}{\Lambda, \Gamma' \cup \{y : \tau'\} \cup \{x : \tau\} \vdash y : \tau'}$ (var)

1. $\Lambda, \Gamma' \cup \{y : \tau'\} \vdash e : \tau$ assumption
2. $\Lambda, \Gamma' \cup \{y : \tau'\} \cup \{x : \tau\} \vdash y : \tau'$ assumption
3. $x \neq y \Rightarrow$
   a. $[e/x]y = y$ 3, definition of $[e/x]y$
   b. $\Lambda, \Gamma' \cup \{y : \tau'\} \vdash [e/x]y : \tau'$ 3a, Rule var
4. $x = y \Rightarrow$
   a. $[e/x]y = e$ 4, definition of $[e/x]y$
   a. $\tau = \tau'$ 2, 4
   b. $\Lambda, \Gamma' \cup \{y : \tau'\} \vdash [e/x]y : \tau'$ 1, 4a, 4b

   Result is from 3b and 4c

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Int \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : Int}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 + e_2 : Int}$ (add)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : Int$ assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : Int$ assumption
3. $\Lambda, \Gamma \vdash e : \tau$ assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : Int$ 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : Int$ 2, 3, IH
6. $\Lambda, \Gamma \vdash [e/x]e_1 + [e/x]e_2 : Int$ 4, 5, Rule add
7. $\Lambda, \Gamma \vdash [e/x](e_1 + e_2) : Int$ 6, definition of $[e/x](e_1 + e_2)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau'\ Ref}{\Lambda, \Gamma \cup \{x : \tau\} \vdash !e'' : \tau'}$ (accessRef)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau'\ Ref$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \tau'\ Ref$ 1, 2, IH
4. $\Lambda, \Gamma \vdash !([e/x]e'') : \tau'$ 3, Rule accessRef
5. $\Lambda, \Gamma \vdash [e/x](!e'') : \tau'$ 4, definition of $[e/x](!e'')$

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash unit : Unit}$ (unitVal)

1. $\tau' = Unit$ — assumption
2. $[e/x]unit = unit$ — definition of $[e/x]unit$
3. $\Lambda, \Gamma \vdash [e/x]unit : \tau'$ — 1, 2, Rule unitVal

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash ([] : \tau_{1_{List}}) : \tau_{1_{List}}}$ (listEmptyVal)

1. $\tau' = \tau_{1_{List}}$ — assumption
2. $[e/x]([] : \tau_{1_{List}}) = ([] : \tau_{1_{List}})$ — definition of $[e/x]([] : \tau_{1_{List}})$
3. $\Lambda, \Gamma \vdash [e/x]([] : \tau_{1_{List}}) : \tau'$ — 1, 2, Rule listEmptyVal

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{1_{List}}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash head(e_1) : \tau_1\ Option}$ (head)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{1_{List}}$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_{1_{List}}$ — 1, 2, IH
4. $\Lambda, \Gamma \vdash head([e/x]e_1) : \tau_1\ Option$ — 3, Rule head
5. $\Lambda, \Gamma \vdash [e/x](head(e_1)) : \tau_1\ Option$ — 4, definition of $[e/x](head(e_1))$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{1_{List}}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash tail(e_1) : \tau_{1_{List}}}$ (tail)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{1_{List}}$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_{1_{List}}$ — 1, 2, IH
4. $\Lambda, \Gamma \vdash tail([e/x]e_1) : \tau_{1_{List}}$ — 3, Rule tail
5. $\Lambda, \Gamma \vdash [e/x](tail(e_1)) : \tau_{1_{list}}$ — 4, definition of $[e/x](tail(e_1))$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau''_{List}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash empty(e'') : Bool}$ (empty)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau''_{List}$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $[e/x]emptpy(e'') = empty([e/x]e'')$ — definition of $[e/x]empty(e'')$
4. $\Lambda, \Gamma \vdash [e/x]e'' : \tau''_{List}$ — 1, 2, IH
5. $\Lambda, \Gamma \vdash [e/x]empty(e'') : Bool$ — 3, 4, Rule empty

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_{1_{List}}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 :: e_2 : \tau_{1_{List}}}$ (listCons)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_{1_{List}}$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau_{1_{List}}$ — 2, 3, IH
6. $\Lambda, \Gamma \vdash [e/x]e_1 :: [e/x]e_2 : \tau_{1_{List}}$ — 4, 5, Rule listCons
7. $\Lambda, \Gamma \vdash [e/x](e_1 :: e_2) : \tau_{1_{List}}$ — 6, definition of $[e/x](e_1 :: e_2)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{1_{List}} \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_{1_{List}}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1@e_2 : \tau_{1_{List}}}$ (listAppend)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{1_{List}}$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_{1_{List}}$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_{1_{List}}$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau_{1_{List}}$ — 2, 3, IH
6. $\Lambda, \Gamma \vdash [e/x]e_1@[e/x]e_2 : \tau_{1_{List}}$ — 4, 5, Rule listAppend
7. $\Lambda, \Gamma \vdash [e/x](e_1@e_2) : \tau_{1_{List}}$ — 6, definition of $[e/x](e_1@e_2)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau''}{\Lambda, \Gamma \cup \{x : \tau\} \vdash ref\ e'' : \tau''\ Ref}$ (createRef)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau''$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \tau''$ — 1, 2, IH
4. $\Lambda, \Gamma \vdash ref\ ([e/x]e'') : \tau''\ Ref$ — 3, Rule createRef
5. $\Lambda, \Gamma \vdash [e/x](ref\ e'') : \tau''\ Ref$ — 4, definition of $[e/x](ref\ e'')$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_2}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1;\ e_2 : \tau_2}$ (sequence)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_2$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau_2$ — 2, 3, IH
6. $\Lambda, \Gamma \vdash [e/x]e_1;\ [e/x]e_2 : \tau_2$ — 4, 5, Rule sequence
7. $\Lambda, \Gamma \vdash [e/x](e_1;\ e_2) : \tau_2$ — 6, definition of $[e/x](e_1;\ e_2)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau''\ Ref \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau''}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1\ :=\ e_2 : Unit}$ (assignment)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau''\ Ref$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau''$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau''\ Ref$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau''$ — 2, 3, IH
6. $\Lambda, \Gamma \vdash [e/x]e_1\ :=\ [e/x]e_2 : Unit$ — 4, 5, Rule assignment
7. $\Lambda, \Gamma \vdash [e/x](e_1\ :=\ e_2) : Unit$ — 6, definition of $[e/x](e_1\ :=\ e_2)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau, x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e'' : \tau_2}{\Lambda, \Gamma \cup \{x : \tau\} \vdash fun\ x_1(x_2 : \tau_1) : \tau_2 = e'' : \tau_1 \to \tau_2}$ (fun)

1. $e' = fun\ x_1(x_2 : \tau_1) : \tau_2 = e''$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau, x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e'' : \tau_2$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_2\} \vdash [e/x]e'' : \tau_2$ — 2, 3, IH
5. $\Lambda, \Gamma \vdash fun\ x_1(x_2 : \tau_1)\ : \tau_2 = [e/x]e''\ : \tau_1 \to \tau_2$ — 4, Rule fun
6. $\Lambda, \Gamma \vdash [e/x]e' : \tau_1 \to \tau_2$ — 5, definition of $[e/x]e'$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \mathit{Event}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \mathit{setOutput}(e'') : \mathit{Bool}}$ (setOutput)

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau, y : \tau_1\} \vdash e_2 : \tau_2}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \mathit{let}\ y = e_1\ \mathit{in}\ e_2\ \mathit{end} : \tau_2}$ (let)

1. $e' = (\mathit{let}\ y = e_1\ \mathit{in}\ e_2\ \mathit{end})$ — assumption
2. $\tau' = \tau_2$ — assumption
3. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1$ — assumption
4. $\Lambda, \Gamma \cup \{x : \tau, y : \tau_1\} \vdash e_2 : \tau_2$ — assumption
5. $\Lambda, \Gamma \vdash e : \tau$ — assumption
6. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1$ — 3, 5, IH
7. $\Lambda, \Gamma \cup \{y : \tau_1\} \vdash [e/x]e_2 : \tau_2$ — 4, 5, IH
8. $\Lambda, \Gamma \vdash \mathit{let}\ y = [e/x]e_1\ \mathit{in}\ [e/x]e_2\ \mathit{end} : \tau_2$ — 6, 7, Rule let
9. $\Lambda, \Gamma \vdash [e/x]e' : \tau'$ — 1, 2, 8, definition of $[e/x]\mathit{let}\ y = e_1\ \mathit{in}\ e_2\ \mathit{end}$

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \mathit{outputNotSet}() : \mathit{Bool}}$ (outputNotSet)

1. $\tau' = \mathit{Bool}$ — assumption
2. $[e/x]\mathit{outputNotSet}()$ = outputNotSet() — definition of $[e/x]\mathit{outputNotSet}()$
3. $\Lambda, \Gamma \vdash [e/x]\mathit{outputNotSet}() : \tau'$ — 1, 2, Rule outputNotSet

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_1}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \mathit{call}(e_1, e_2) : \tau_2}$ (call)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \to \tau_2$ — assumption
2. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_1$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e_1 : \tau_1 \to \tau_2$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash [e/x]e_2 : \tau_1$ — 2, 3, IH
6. $\Lambda, \Gamma \vdash \mathit{call}([e/x]e_1, [e/x]e_2) : \tau_2$ — 4, 5, Rule call
7. $\Lambda, \Gamma \vdash [e/x](\mathit{call}(e_1, e_2)) : \tau_2$ — 6, definition of $[e/x](\mathit{call}(e_1, e_2))$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e''.\ell_i : \tau_i}$ (projection)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$ — assumption
2. $i \in \{1, \ldots, n\}$ — assumption
3. $\Lambda, \Gamma \vdash e : \tau$ — assumption
4. $\Lambda, \Gamma \vdash [e/x]e'' : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$ — 1, 3, IH
5. $\Lambda, \Gamma \vdash ([e/x]e'').\ell_i : \tau_i$ — 2, 4, Rule projection
6. $\Lambda, \Gamma \vdash [e/x](e''.\ell_i) : \tau_i$ — 5, definition of $[e/x](e''.\ell_i)$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau'}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \{\ e''\ \}_{s(v)} : \tau'}$ (endLabel)

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau'$ — assumption
2. $\Lambda, \Gamma \vdash e : \tau$ — assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \tau'$ — 1, 2, IH
4. $\Lambda, \Gamma \vdash \{\ [e/x]e''\ \}_{s(v)} : \tau'$ — 3, Rule endLabel
5. $\Lambda, \Gamma \vdash [e/x](\{\ e''\ \}_{s(v)}) : \tau'$ — 4, definition of $[e/x]\{e''\}_{s(v)}$

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \mathit{Obligation}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \mathit{makeCFG}(e'') : \mathit{CFG}}$ (makeCFG)

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \mathit{Bool} \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau''}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \mathit{while}(e_1)\ \{e_2\} : \mathit{Bool}}$ (while)

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

Case $\dfrac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash (in_{\ell_i}\ e_i : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}$ (variant)

1. $e' = (in_{\ell_i}\ e_i : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n))$ — assumption
2. $\tau' = (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — assumption
3. $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_i : \tau_i$ — assumption
4. $\Lambda, \Gamma \vdash e : \tau$ — assumption
5. $i \in \{1, \ldots, n\}$ — assumption
6. $\Lambda, \Gamma \vdash [e/x]e_i : \tau_i$ — 3, 4, IH
7. $\Lambda, \Gamma \vdash (in_{\ell_i}\ [e/x]e_i : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)) : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — 5, 6, Rule variant
8. $\Lambda, \Gamma \vdash [e/x]e' : \tau'$ — 1, 2, 7, definition of $[e/x](in_{\ell_i}\ e_i : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n))$

Case $\dfrac{}{\Lambda, \Gamma \cup \{x : \tau\} \vdash getOutput() : Event\ Option}$ (getOutput)

1. $\tau' = Event\ Option$ — assumption
2. $[e/x]getOutput() = getOutput()$ — definition of $[e/x]getOutput()$
3. $\Lambda, \Gamma \vdash [e/x]getOutput() : \tau'$ — 1, 2, Rule getOutput

□

Lemma 7 (Typing Rule Inversion). *All of the typing rules are invertable.*

Proof. By case analysis of rules deriving $\Lambda, \Gamma \vdash e : \tau$

Case $\dfrac{}{\Lambda, \Gamma \vdash n : Int}$ (intVal)

1. $\Lambda, \Gamma \vdash n : \tau$ — assumption
2. $\Lambda, \Gamma \vdash n : \tau$ is only derivable by Rule intVal — inspection of typing rules
3. $\tau = Int$ — 1, 2, rule intVal

Case $\dfrac{}{\Lambda, \Gamma \vdash b : Bool}$ (boolVal)

1. $\Lambda, \Gamma \vdash b : \tau$ — assumption
2. $\Lambda, \Gamma \vdash b : \tau$ is only derivable by Rule boolVal — inspection of typing rules
3. $\tau = Bool$ — 1, 2, rule boolVal

Case $\dfrac{}{\Lambda, \Gamma \vdash s : String}$ (stringVal)

1. $\Lambda, \Gamma \vdash s : \tau$ — assumption
2. $\Lambda, \Gamma \vdash s : \tau$ is only derivable by Rule stringVal — inspection of typing rules
3. $\tau = String$ — 1, 2, rule stringVal

Case $\dfrac{}{\Lambda, \Gamma \vdash unit : Unit}$ (unitVal)

1. $\Lambda, \Gamma \vdash unit : \tau$ — assumption
2. $\Lambda, \Gamma \vdash unit : \tau$ is only derivable by Rule unitVal — inspection of typing rules
3. $\tau = Unit$ — 1, 2, rule unitVal

Case $\dfrac{\Lambda, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e' : \tau_2}{\Lambda, \Gamma \vdash fun\ x_1(x_2 : \tau_1) : \tau_2\ \{\ e'\ \} : \tau_1 \to \tau_2}$ (fun)

1. $\Lambda, \Gamma \vdash fun\ x_1(x_2 : \tau_1) : \tau_2\ \{\ e'\ \} : \tau$ — assumption
2. $\Lambda, \Gamma \vdash fun\ x_1(x_2 : \tau_1) : \tau_2\ \{\ e'\ \} : \tau$ is only derivable by rule fun — inspection of typing rules
3. $\tau = \tau_1 \to \tau_2,\ \Lambda, \Gamma \cup \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e' : \tau_2$ — 1, 2, Rule fun

Case $\dfrac{}{\Lambda' \cup \{\ell : \tau'\}, \Gamma \vdash \ell : \tau'\ Ref}$ (location)

1. $\Lambda' \cup \{\ell : \tau\}, \Gamma \vdash \ell : \tau$ — assumption
2. 1 is only derivable by Rule location — inspection of typing rules
3. $\tau = \tau'\ Ref$ — 1, 2, Rule location

Case $\dfrac{}{\Lambda, \Gamma' \cup \{x : \tau'\} \vdash x : \tau'}$ (var)

1. $\Lambda, \Gamma' \cup \{x : \tau'\} \vdash x : \tau$ — assumption
2. 1 is only derivable by Rule var — inspection of typing rules
3. $\tau = \tau'$ — 1, 2, Rule var

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : Bool}{\Lambda, \Gamma \vdash e_1 \wedge e_2 : Bool}$ (con)

1. $\Lambda, \Gamma \vdash e_1 \wedge e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 \wedge e_2 : \tau$ is only derivable by rule con — inspection of typing rules
3. $\tau = Bool,\ \Lambda, \Gamma \vdash e_1 : Bool,\ \Lambda, \Gamma \vdash e_2 : Bool$ — 1, 2, Rule con

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : Bool}{\Lambda, \Gamma \vdash e_1 \vee e_2 : Bool}$ (dis)

1. $\Lambda, \Gamma \vdash e_1 \vee e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 \vee e_2 : \tau$ is only derivable by Rule dis — inspection of typing rules
3. $\tau = Bool,\ \Lambda, \Gamma \vdash e_1 : Bool,\ \Lambda, \Gamma \vdash e_2 : Bool$ — 1, 2, Rule dis

Case $\dfrac{\Lambda, \Gamma \vdash e : Bool}{\Lambda, \Gamma \vdash \neg e : Bool}$ (negation)

1. $\Lambda, \Gamma \vdash \neg e : \tau$ — assumption
2. $\Lambda, \Gamma \vdash \neg e : \tau$ is only derivable by rule negation — inspection of typing rules
3. $\tau = Bool,\ \Lambda, \Gamma \vdash e : Bool$ — 1, 2, Rule negation

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_1}{\Lambda, \Gamma \vdash e_1 == e_2 : Bool}$ (equality)

1. $\Lambda, \Gamma \vdash e_1 == e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 == e_2 : \tau$ is only derivable by rule equality — inspection of typing rules
3. $\tau = Bool,\ \Lambda, \Gamma \vdash e_1 : \tau_1,\ \Lambda, \Gamma \vdash e_2 : \tau_1, \tau_1 \in \{Int, Bool, String\}$ — 1, 2, Rule equality

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Int \quad \Lambda, \Gamma \vdash e_2 : Int}{\Lambda, \Gamma \vdash e_1 + e_2 : Int}$ (add)

1. $\Lambda, \Gamma \vdash e_1 + e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 + e_2 : \tau$ is only derivable by rule add — inspection of typing rules
3. $\tau = Int,\ \Lambda, \Gamma \vdash e_1 : Int,\ \Lambda, \Gamma \vdash e_2 : Int$ — 1, 2, Rule add

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_2}{\Lambda, \Gamma \vdash e_1;\ e_2 : \tau_2}$ (sequence)

1. $\Lambda, \Gamma \vdash e_1;\ e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1;\ e_2 : \tau$ is only derivable by rule sequence — 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 sequence

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : \tau_1 \quad \Lambda, \Gamma \vdash e_3 : \tau_1}{\Lambda, \Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau_1}$ (if)

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

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Bool \quad \Lambda, \Gamma \vdash e_2 : \tau_1}{\Lambda, \Gamma \vdash while(e_1)\ \{e_2\} : Bool}$ (while)

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

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Lambda, \Gamma \vdash let\ x = e_1\ in\ e_2\ end : \tau_2}$ (let)

1. $\Lambda, \Gamma \vdash let\ x = e_1\ in\ e_2\ end : \tau$ — assumption
2. $\Lambda, \Gamma \vdash let\ x = e_1\ in\ e_2\ end : \tau$ is only derivable by rule 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 let

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_1}{\Lambda, \Gamma \vdash ref\ e : \tau_1\ Ref}$ (createRef)

1. $\Lambda, \Gamma \vdash ref\ e : \tau$ — assumption
2. $\Lambda, \Gamma \vdash ref\ e : \tau$ is only derivable by rule createRef — inspection of typing rules
3. $\tau = \tau_1\ Ref,\ \Lambda, \Gamma \vdash e : \tau_1$ — 1, 2, Rule createRef

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_1\ Ref}{\Lambda, \Gamma \vdash !e : \tau_1}$ (accessRef)

1. $\Lambda, \Gamma \vdash !e : \tau$ — assumption
2. $\Lambda, \Gamma \vdash !e : \tau$ is only derivable by rule accessRef — inspection of typing rules
3. $\tau = \tau_1,\ \Lambda, \Gamma \vdash e : \tau_1\ Ref$ — 1, 2, Rule accessRef

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1\ Ref \qquad \Lambda, \Gamma \vdash e_2 : \tau_1}{\Lambda, \Gamma \vdash e_1 := e_2 : unit}$ (assignment)

1. $\Lambda, \Gamma \vdash e_1 := e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 := e_2 : \tau$ is only derivable by rule assignment — inspection of typing rules
3. $\tau = unit,\ \Lambda, \Gamma \vdash e_1 : \tau_1\ Ref,\ \Lambda, \Gamma \vdash e_2 : \tau_1$ — 1, 2, Rule assignment

Case $\dfrac{}{\Lambda, \Gamma \vdash ([] : \tau'_{List}) : \tau'_{List}}$ (listEmptyVal)

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

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_{1_{List}}}{\Lambda, \Gamma \vdash e_1 :: e_2 : \tau_{List}}$ (listCons)

1. $\Lambda, \Gamma \vdash e_1 :: e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 :: e_2 : \tau$ is only derivable by rule listCons — inspection of typing rules
3. $\tau = \tau_{1_{List}},\ \Lambda, \Gamma \vdash e_1 : \tau_1,\ \Lambda, \Gamma \vdash e_2 : \tau_{1_{List}}$ — 1, 2, Rule listCons

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_{1_{List}} \quad \Lambda, \Gamma \vdash e_2 : \tau_{1_{List}}}{\Lambda, \Gamma \vdash e_1 \mathbin{@} e_2 : \tau_{List}}$ (listAppend)

1. $\Lambda, \Gamma \vdash e_1 \mathbin{@} e_2 : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e_1 \mathbin{@} e_2 : \tau$ is only derivable by rule listAppend — inspection of typing rules
3. $\tau = \tau_{1_{List}},\ \Lambda, \Gamma \vdash e_1 : \tau_{1_{List}},\ \Lambda, \Gamma \vdash e_2 : \tau_{1_{List}}$ — 1, 2, Rule listAppend

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_{1_{List}}}{\Lambda, \Gamma \vdash head(e) : \tau_1}$ (head)

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

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_{1_{List}}}{\Lambda, \Gamma \vdash tail(e) : \tau_{1_{List}}}$ (tail)

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

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_{List}}{\Lambda, \Gamma \vdash empty(e) : Bool}$ (empty)

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

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma \vdash e_n : \tau_n}{\Lambda, \Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}$ (record)

1. $\Lambda, \Gamma \vdash (\ell_1 = v_1, \ldots, \ell_n = v_n) : \tau$ — assumption
2. $\Lambda, \Gamma \vdash (\ell_1 = v_1, \ldots, \ell_n = v_n) : \tau$ is only derivable by rule record — inspection of typing rules
3. $\tau = (\ell_1 : \tau_1 \times \ldots \times \ell_n : \tau_n),\ \Lambda, \Gamma \vdash v_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma \vdash v_n : \tau_n$ — 1, 2, Rule record

Case $\dfrac{\Lambda, \Gamma \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma \vdash e.\ell_i : \tau_i}$ (projection)

1. $\Lambda, \Gamma \vdash e.\ell_i : \tau$ — assumption
2. $\Lambda, \Gamma \vdash e.\ell_i : \tau$ is only derivable by rule projection — inspection of typing rules
3. $\tau = \tau_i,\ \Lambda, \Gamma \vdash e : (\ell_1 : \tau_1 \times \ldots \times \ell_n : \tau_n)$ — 1, 2, Rule projection

Case $\dfrac{\Lambda, \Gamma \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda, \Gamma \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \cdots + \ell_n) : \tau_n\ : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}$ (variant)

1. $\Lambda, \Gamma \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n) : \tau$ — assumption
2. $\Lambda, \Gamma \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n) : \tau$ is only derivable by rule variant — inspection of typing rules
3. $\tau = (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n),\ \Lambda, \Gamma \vdash e_i : \tau_i,\ i \in \{1, \ldots, n\}$ — 1, 2, Rule variant

Case $\dfrac{\Lambda, \Gamma \vdash e' : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau_x \quad \cdots \quad \Lambda, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau_x}{\Lambda, \Gamma \vdash (case\ e'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau_x}$ (case)

1. $\Lambda, \Gamma \vdash (case\ e'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau$ — assumption
2. $\Lambda, \Gamma \vdash (case\ e'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau$ is only derivable by rule case — inspection of typing rules
3. $\tau = \tau_x,\ \Lambda, \Gamma \vdash e' : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n),\ \Lambda, \Gamma \cup \{x_1 : \tau_1\} \vdash e_1 : \tau_x \quad \cdots \quad \Lambda, \Gamma \cup \{x_n : \tau_n\} \vdash e_n : \tau_x$ — 1, 2, Rule case

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1}{\Lambda, \Gamma \vdash makeTypedVal(\tau_1, e_1) : TypedVal}$ (makeTypedVal)

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

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : TypedVal}{\Lambda, \Gamma \vdash tryCast(\tau_1, e_1) : \tau_1}$ (tryCast)

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

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_1}{\Lambda, \Gamma \vdash \{\ e\ \}_{label} : \tau_1}$ (endLabel)

1. $\Lambda, \Gamma \vdash \{\ e\ \}_{label} : \tau$ — assumption
2. $\Lambda, \Gamma \vdash \{\ e\ \}_{label} : \tau$ is only derivable by rule endLabel — inspection of typing rules
3. $\tau = \tau_1,\ \Lambda, \Gamma \vdash e : \tau_1$ — 1, 2, Rule endLabel

Case $\dfrac{}{\Lambda, \Gamma \vdash getRT() : Res_{List}}$ (getRT)

1. $\Lambda, \Gamma \vdash getRT() : \tau$ — assumption
2. 1 is only derivable by Rule getRT — inspection of typing rules
3. $\tau = Res_{List}$ — 1, 2, Rule getRT

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Obligation}{\Lambda, \Gamma \vdash makeCFG(e_1) : CFG}$ (makeCFG)

1. $\Lambda, \Gamma \vdash makeCFG(e_1) : \tau$ — assumption
2. $\Lambda, \Gamma \vdash makeCFG(e_1) : \tau$ is only derivable by rule makeCFG — inspection of typing rules
3. $\tau = CFG,\ \Lambda, \Gamma \vdash e_1 : Obligation$ — 1, 2, Rule makeCFG

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : Event}{\Lambda, \Gamma \vdash setOutput(e_1) : unit}$ (setOutput)

1. $\Lambda, \Gamma \vdash setOutput(e_1) : \tau$ — assumption
2. $\Lambda, \Gamma \vdash setOutput(e_1) : \tau$ is only derivable by rule setOutput — inspection of typing rules
3. $\tau = unit,\ \Lambda, \Gamma \vdash e_1 : Event$ — 1, 2, Rule setOutput

Case $\dfrac{}{\Lambda, \Gamma \vdash outputNotSet() : Bool}$ (outputNotSet)

1. $\Lambda, \Gamma \vdash outputNotSet() : \tau$ — assumption
2. 1 is only derivable by Rule outputNotSet — inspection of typing rules
3. $\tau = Bool$ — 1, 2, Rule outputNotSet

Case $\dfrac{}{\Lambda, \Gamma \vdash getOutput() : Event\ Option}$ (getOutput)

1. $\Lambda, \Gamma \vdash getOutput() : \tau$ — assumption
2. 1 is only derivable by Rule getOutput — inspection of typing rules
3. $\tau = Event\ Option$ — 1, 2, Rule getOutput

Case $\dfrac{\Lambda, \Gamma \vdash e : (evt : \tau'\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda, \Gamma \vdash monitor(\tau', e) : \tau'\ Event}$ (monitor)

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

Result is from 3, 4

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

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

Case $\dfrac{\Lambda, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda, \Gamma \vdash e_2 : \tau_1}{\Lambda, \Gamma \vdash call(e_1, e_2) : \tau_2}$ (call)

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

□

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

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

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

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

1. $v = n$ assumption

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

1. $v = b$ assumption

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

1. $v = s$ assumption

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

1. $v = unit$ assumption

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

1. $v = fun\ x_1(x_2 : \tau_1) : \tau_2 = e$ assumption

Case $\dfrac{}{(\Lambda' \cup \{\ell : \tau\}), \Gamma \vdash \ell : \tau\ Ref}$ (location)

1. $v = \ell$ assumption

Case $\dfrac{}{\Lambda, \bullet \cup \{x : \tau\} \vdash x : \tau}$ (var)

1. x is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : Bool}{\Lambda, \bullet \vdash e_1 \wedge e_2 : Bool}$ (con)

1. $e_1 \wedge e_2$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : Bool}{\Lambda, \bullet \vdash e_1 \vee e_2 : Bool}$ (dis)

1. $e_1 \vee e_2$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e : Bool}{\Lambda, \bullet \vdash \neg e : Bool}$ (negation)

1. $\neg e$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau \quad \Lambda, \bullet \vdash e_2 : \tau \quad \tau \in \{Int, Bool, String\}}{\Lambda, \bullet \vdash e_1 == e_2 : Bool}$ (equality)

1. $e_1 == e_2$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Int \quad \Lambda, \bullet \vdash e_2 : Int}{\Lambda, \bullet \vdash e_1 + e_2 : Int}$ (add)

1. $e_1 + e_2$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \quad \Lambda, \bullet \vdash e_2 : \tau_2}{\Lambda, \bullet \vdash e_1;\ e_2 : \tau_2}$ (sequence)

1. $e_1;\ e_2$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : \tau \quad \Lambda, \bullet \vdash e_3 : \tau}{\Lambda, \bullet \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}$ (if)

1. $if\ e_1\ then\ e_2\ else\ e_3$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : \tau}{\Lambda, \bullet \vdash while(e_1)\ \{e_2\} : Bool}$ (while)

1. $while(e_1)\ \{e_2\}$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \quad \Lambda, \bullet \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Lambda, \bullet \vdash let\ x = e_1\ in\ e_2\ end : \tau_2}$ (let)

1. $let\ x = e_1\ in\ e_2\ end$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e : \tau}{\Lambda, \bullet \vdash ref\ e : \tau\ Ref}$ (createRef)

1. $ref\ e$ is not a value the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e : \tau Ref}{\Lambda, \bullet \vdash !e : \tau}$ (accessRef)

1. $!e$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau\ Ref \qquad \Lambda, \bullet \vdash e_2 : \tau}{\Lambda, \bullet \vdash e_1 := e_2 : unit}$ (assignment)

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

Case $\dfrac{}{\Lambda, \bullet \vdash ([] : \tau_{List}) : \tau_{List}}$ (listEmptyVal)

1. $v = [] : \tau_{List}$ assumption

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau \quad \Lambda, \bullet \vdash e_2 : \tau_{List}}{\Lambda, \bullet \vdash e_1 :: e_2 : \tau_{List}}$ (listCons)

1. $e_1 = v_1$ *and* $e_2 = v_2 \Rightarrow e_1 :: e_2 = v_1 :: v_2$ assumption
2. $e_1 \neq v_1$ *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

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List} \quad \Lambda, \bullet \vdash e_2 : \tau_{List}}{\Lambda, \bullet \vdash e_1 \mathbin{@} e_2 : \tau_{List}}$ (listAppend)

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

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List}}{\Lambda, \bullet \vdash head(e_1) : \tau}$ (head)

1. $head(e_1)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List}}{\Lambda, \bullet \vdash tail(e_1) : \tau_{List}}$ (tail)

1. $tail(e_1)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \Gamma \vdash e : \tau_{List}}{\Lambda, \Gamma \vdash empty(e) : Bool}$ (empty)

1. $empty(e_1)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \bullet \vdash e_n : \tau_n}{\Lambda, \bullet \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}$ (record)

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

Result from 1, 2

Case $\dfrac{\Lambda, \bullet \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\}}{\Lambda, \bullet \vdash e.\ell_i : \tau_i}$ (projection)

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

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda, \bullet \vdash (in_{\ell_i}\ e_1 : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}$ (variant)

1. $\tau = (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ assumption
2. $v = in_{\ell_i}\ e_1 : \tau$ assumption, 1
3. $i \in \{1, \ldots, n\}$ assumption
4. $e_1 = v_1 \Rightarrow v = in_{\ell_i}\ v_1 : \tau$ assumption, 2
5. $e_i \neq v_i \Rightarrow v$ is not a value the lemma holds vacuously in this case

Result is from 4, 5

Case $\dfrac{\Lambda, \bullet \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda, \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \{x_n : \tau_n\} \vdash e_n : \tau}{\Lambda, \bullet \vdash (case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau}$ (case)

1. $(case\ e\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau'}{\Lambda, \bullet \vdash makeTypedVal(\tau', e_1) : TypedVal}$ (makeTypedVal)

1. $e_1 = v_1 \Rightarrow v = makeTypedVal(\tau', v_1)$ assumption
2. $e_1 \neq v_1 \Rightarrow v$ is not a value the lemma holds vacuously in this case

Result is from 1, 2

Case $\dfrac{\Lambda, \bullet \vdash e_1 : TypedVal}{\Lambda, \bullet \vdash tryCast(\tau, e_1) : \tau}$ (tryCast)

1. $tryCast(\tau, e_1)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e : \tau}{\Lambda, \bullet \vdash \{\ e\ \}_{label} : \tau}$ (endLabel)

1. $\{\ e\ \}_{label}$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{}{\Lambda, \Gamma \vdash getRT() : Res_{List}}$ (getRT)

1. $getRT()$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Obligation}{\Lambda, \bullet \vdash makeCFG(e_1) : CFG}$ (makeCFG)

1. $makeCFG(e_1)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e : Event}{\Lambda, \bullet \vdash setOutput(e) : unit}$ (setOutput)

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

Case $\dfrac{}{\Lambda, \bullet \vdash outputNotSet() : Bool}$ (outputNotSet)

1. $outputNotSet()$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{}{\Lambda, \bullet \vdash getOutput() : (event : Event + none : Unit)}$ (getOutput)

1. $getOutput()$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda, \Gamma \vdash monitor(\tau_1, e_1) : \tau_1\ Event}$ (monitor)

1. $monitor(\tau_1, e_1)$ is not a value so the lemma holds vacuously in this case

Case $\dfrac{\Lambda, \bullet \vdash e_1 : String \quad \Lambda, \bullet \vdash e_1 \in F.name \quad \Lambda, \bullet \vdash F[e_1].fun : \tau_1 \to \tau_2 \quad \Lambda, \bullet \vdash e_2 : TypedVal}{\Lambda, \bullet \vdash invoke(e_1, e_2) : TypedVal}$ (invoke)

1. $invoke(e_1, e_2)$ is not a value so the lemma holds vacuously in this case

□

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda, \bullet \vdash e_2 : \tau_1}{\Lambda, \bullet \vdash call(e_1, e_2) : \tau_2}$ (call)

1. $call(e_1, e_2)$ is not a value so the lemma holds vacuously in this case

Lemma 9 (Progress). $\Lambda \vdash (C, e) : \tau \Rightarrow e$ value $\vee\ \exists C', e' : (C, e) \longrightarrow (C', e')$

We will instead prove the equivalent statement $\Lambda \vdash C$ **ok** $\wedge\ \Lambda, \bullet \vdash e : \tau \Rightarrow e$ value $\vee\ \exists C', e' : (C, e) \longrightarrow (C', e')$. 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 $\dfrac{}{\Lambda, \bullet \vdash n : Int}$ (intVal)

1. $n\ value$ definition of values

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

1. b value definition of values

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

1. $s\ value$ definition of values

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

1. $unit\ value$ definition of values

Case $\dfrac{\Lambda, \bullet \vdash e_2 : \tau_2}{\Lambda, \bullet \vdash fun\ x_1(x_2 : \tau_1) : \tau_2 = e_2 : \tau_1 \to \tau_2}$ (fun)

1. $(fun\ x_1(x_2 : \tau_1) : \tau_2 = e_2)\ value$ definition of values

Case $\dfrac{}{\Lambda \cup \{\ell : \tau\}, \bullet \vdash \ell : \tau\ Ref}$ (location)

1. $\ell\ value$ definition of values

Case $\dfrac{}{\Lambda, \bullet \vdash x : \tau}$ (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 $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : Bool}{\Lambda, \bullet \vdash e_1 \wedge e_2 : Bool}$ (con)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash e_1 : Bool$ assumption
3. $\Lambda, \bullet \vdash e_2 : Bool$ assumption
4. $e_1 = v_1\ \ or\ \ (C, e_1) \longrightarrow (C', e_1')$ 1, 2, IH
5. $e_2 = v_2\ \ or\ \ (C, e_2) \longrightarrow (C', e_2')$ 1, 3, IH
6. $e_1 = v_1 \Rightarrow e_1 \in \{true,\ false\}$ 2, Lemma Canonical Forms
7. $(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e_1 \wedge e_2) \longrightarrow (C', e_1' \wedge e_2)$ Rule andE1
8. $(e_1 = v_1)\ and\ (C, e_2) \longrightarrow (C', e_2') \Rightarrow (C, e_1 \wedge e_2) \longrightarrow (C', v_1 \wedge e_2')$ Rule andE2
9. $(e_1 = true\ \ and\ \ e_2 = v_2) \Rightarrow (C, e_1 \wedge e_2) \longrightarrow (C, v_2)$ Rule andTrue
10. $(e_1 = false\ \ and\ \ e_2 = v_2) \Rightarrow (C, e_1 \wedge e_2) \longrightarrow (C, false)$ Rule andFalse
11. $\exists C', e' : (C, e_1 \wedge e_2) \longrightarrow (C', e')$ 7-10

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : Bool}{\Lambda, \bullet \vdash e_1 \vee e_2 : Bool}$ (dis)

1. This case is analogous to case con.

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool}{\Lambda, \bullet \vdash \neg e_1 : Bool}$ (negation)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash e_1 : Bool$ assumption
3. $e_1 = v_1 \vee (C, e_1) \longrightarrow (C', e_1')$ 1, 2, IH
4. $e_1 = v_1 \Rightarrow e_1 \in \{true, false\}$ 2, Lemma Canonical Forms
5. $e_1 = true \Rightarrow (C, \neg e_1) \longrightarrow (C, false)$ notTrue
6. $e_1 = false \Rightarrow (C, \neg e_1) \longrightarrow (C, true)$ notFalse
7. $(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, \neg e_1) \longrightarrow (C', \neg e_1')$ notE
8. $\exists C', e' : (C, \neg e_1) \longrightarrow (C', e')$ 5-7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau \quad \Lambda, \bullet \vdash e_2 : \tau \quad \tau \in \{Int, Bool, String\}}{\Lambda, \bullet \vdash e_1 == e_2 : Bool}$ (equality)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau$	assumption
4.	$\tau \in \{Int, Bool, String\}$	assumption
5.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
6.	$e_2 = v_2$ *or* $(C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
7.	$e_1 \longrightarrow e_1' \Rightarrow (C, e_1 == e_2) \longrightarrow (C', e_1' == e_2)$	Rule eqE1
8.	$e_1 = v_1$ *and* $e_2 \longrightarrow e_2' \Rightarrow (C, e_1 == e_2) \longrightarrow (C', v_1 == e_2')$	Rule eqE2
9.	$e_1 = v_1$ *and* $e_2 = v_2 \Rightarrow$	
a.	$\tau = Int \Rightarrow$	
i.	$v_1 = n_1$	Lemma Canonical Forms, 2, 9, 9a
ii.	$v_2 = n_2$	Lemma Canonical Forms, 3, 9, 9a
iii.	$n_1 = n_2 \Rightarrow (C, e_1 == e_2) \longrightarrow (C, true)$	9, (i), rule eqIntTrue
iv.	$n_1 \neq n_2 \Rightarrow (C, e_1 == e_2) \longrightarrow (C, false)$	9, (ii), rule eqIntFalse
b.	$\tau = Bool \Rightarrow$	
i.	$v_1 \in \{true, false\}$	Lemma Canonical Forms, 2, 9, 9b
ii.	$v_2 \in \{true, false\}$	Lemma Canonical Forms, 3, 9, 9b
iii.	$v_2 = b_2$	(ii), definition of b
iv.	$v_1 = true \Rightarrow (C, e_1 == e_2) \longrightarrow (C, b_2)$	9, (iii), rule eqBoolTrue
v.	$v_1 = false \Rightarrow (C, e_1 == e_2) \longrightarrow (C, \neg b_2)$	9, (iii), rule eqBoolFalse
vi.	$\exists C', e' : (C, e_1 == e_2) \longrightarrow (C', e')$	(i), (iv), (v)
c.	$\tau = String \Rightarrow$	
i.	$v_1 = s_1$	Lemma Canonical Forms, 2, 9, 9c
ii.	$v_2 = s_2$	Lemma Canonical Forms, 3, 9, 9c
iii.	$s_1 = s_2 \Rightarrow (C, e_1 == e_2) \longrightarrow (C, true)$	9, (i), rule eqStrTrue
iv.	$s_1 \neq s_2 \Rightarrow (C, e_1 == e_2) \longrightarrow (C, false)$	9, (ii), rule eqStrFalse
d.	$\exists C', e' : (C, e) \longrightarrow (C', e')$	4, a(iii, iv), b(vi), c(iii, iv)
10.	$e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$	7, 8, 9d

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Int \quad \Lambda, \bullet \vdash e_2 : Int}{\Lambda, \bullet \vdash e_1 + e_2 : Int}$ (add)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : Int$	assumption
3.	$\Lambda, \bullet \vdash e_2 : Int$	assumption
4.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$e_2 = v_2$ *or* $(C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
6.	$e_1 = v_1 \Rightarrow e_1 = n_1$	2, Lemma Canonical Forms
7.	$e_2 = v_2 \Rightarrow e_2 = n_2$	3, Lemma Canonical Forms
8.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e_1 + e_2) \longrightarrow (C', e_1' + e_2)$	Rule addE1
9.	$(e_1 = n_1$ *and* $(C, e_2) \longrightarrow (C', e_2')) \Rightarrow (C, e_1 + e_2) \longrightarrow (C', n_1 + e_2')$	Rule addE2
10.	$(e_1 = n_1$ *and* $e_2 = n_2) \Rightarrow (C, e_1 + e_2) \longrightarrow (C, n_1 +_a n_2)$	6, 7, Rule addValue
11.	$e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$	8-10

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \quad \Lambda, \bullet \vdash e_2 : \tau_2}{\Lambda, \bullet \vdash e_1;\ e_2 : \tau_2}$ (sequence)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau_1$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau_2$	assumption
4.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, (e_1; e_2)) \longrightarrow (C', (e_1'; e_2))$	Rule sequenceE1
6.	$e_1 = v_1 \Rightarrow (C, (e_1; e_2)) \longrightarrow (C, e_2)$	Rule sequenceE2
7.	$\exists C', e' : (C, e) \longrightarrow (C', e')$	3-5

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : \tau \quad \Lambda, \bullet \vdash e_3 : \tau}{\Lambda, \bullet \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}$ (if)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : Bool$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau$	assumption
4.	$\Lambda, \bullet \vdash e_3 : \tau$	assumption
5.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
6.	$e_1 = v_1 \Rightarrow e_1 \in \{true, false\}$	3, Lemma Canonical Forms
7.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, if\ e_1\ then\ e_2\ else\ e_3) \longrightarrow (C', if\ e_1'\ then\ e_2\ else\ e_3)$	Rule ifE
8.	$e_1 = true \Rightarrow (C, if\ e_1\ then\ e_2\ else\ e_3) \longrightarrow (C, e_2)$	Rule ifTrue
9.	$e_1 = false \Rightarrow (C, if\ e_1\ then\ e_2\ else\ e_3) \longrightarrow (C, e_3)$	Rule ifFalse
10.	$\exists C', e' : (C, if\ e_1\ then\ e_2\ else\ e_3) \longrightarrow (C', e')$	7-9

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Bool \quad \Lambda, \bullet \vdash e_2 : \tau}{\Lambda, \bullet \vdash while(e_1)\{e_2\} : Bool}$ (while)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$(C, while(e_1)\{e_2\}) \Rightarrow (C, if\ e_1\ then\ while(e_1)\{e_2\}\ else\ false$	1, Rule whileE

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \quad \Lambda, \bullet \vdash e_2 : \tau_2}{\Lambda, \bullet \vdash let\ x = e_1\ in\ e_2\ end : \tau_2}$ (let)

1.	$e = (let\ x = e_1\ in\ e_2\ end)$	assumption
2.	$\Lambda \vdash C$ **ok**	assumption
3.	$\Lambda, \bullet \vdash e_1 : \tau_1$	assumption
4.	$\Lambda, \bullet \cup \{x : \tau_1\} \vdash e_2 : \tau_2$	assumption
5.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	2, 3, IH
6.	$e_1 = v_1 \Rightarrow (C, e) \longrightarrow (C, [v_1/x]e_2)$	Rule letValue
7.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow let\ x = e_1\ in\ e_2\ end \longrightarrow let\ x = e_1'\ in\ e_2\ end$	Rule letE
8.	$\exists C', e', (C, e) \longrightarrow (C', e')$	5-7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau}{\Lambda, \bullet \vdash ref\ e_1 : \tau\ Ref}$ (createRef)

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

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau\ Ref}{\Lambda, \bullet \vdash !e_1 : \tau}$ (accessRef)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$e_1 : \tau\ Ref$	assumption
3.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
4.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, !e_1) \longrightarrow (C', !e_1')$	3, Rule derefE
5.	$e_1 = v \Rightarrow e_1 = \ell_i$	2, Lemma Canonical Forms
6.	$\Lambda = \Lambda' \cup \{\ell : \tau\}$	2, 5, Rule location
7.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-Ok	Inspection of Rule C-Ok
8.	$C = (M, R, inOb, rt, out, \tau_{out}) \wedge M : \Lambda$	1, 7, Inspection of Rule C-Ok
9.	$M : \Lambda$ is only derivable by Rule TMem	Inspection of Rule M:Λ
10.	$M = M' \cup \{(l, v)\}$	6, 8, 9, Rule TMem
11.	$e_1 = v_1 \Rightarrow (C, !e_1) \longrightarrow (C, v)$	8, 10, Rule derefV
12.	$\exists C', e', (C, e) \longrightarrow (C', e')$	3, 4, 11

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau\ Ref \qquad \Lambda, \bullet \vdash e_2 : \tau}{\Lambda, \bullet \vdash e_1 := e_2 : unit}$ (assignment)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau\ Ref$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau$	assumption
4.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$e_2 = v_2$ *or* $(C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
6.	$e_1 = v_1 \Rightarrow e_1 = \ell_1$	2, Lemma Canonical Forms
7.	$\Lambda\bullet \vdash \ell_1 : \tau\ Ref$ is only derivable by Rule location	Inspection of the Rule location
8.	$\Lambda = \Lambda' \cup \{\ell_1 : \tau\}$	2, 6, 7, Inversion of Rule location
9.	$C = (M, R, inOb, rt, out, \tau_{out}) \wedge M : \Lambda$	1, Lemma C-Inversion
10.	$M : \Lambda$ is only derivable by TMem	Inspection of Rule TMem
11.	$M = M' \cup \{(\ell_1, v_{old})\}$	8, 9, 10, rule TMem
12.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e_1 := e_2) \longrightarrow (C', e_1' := e_2)$	Rule assignE1
13.	$e_1 = v_1 \wedge (C, e_2) \longrightarrow (C', e_2') \Rightarrow (C, e_1 := e_2) \longrightarrow (C', v_1 := e_2')$	9, Rule assignE2
14.	$e_1 = v_1 \wedge e_2 = v_2 \Rightarrow (C, e_1 := e_2) \longrightarrow ((M' \cup \{\ell_1, v_2\}, R, inOb, rt, out, \tau_{out}), unit)$	6, 9, Rule assignValue
15.	$\exists C', e', (C, e) \longrightarrow (C', e')$	4-5, 12-14

Case $\dfrac{}{\Lambda, \bullet \vdash ([] : \tau_{List}) : \tau_{List}}$ (listEmptyVal)

1.	$e = ([] : \tau_{List}) = v$	definition of list value

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau \quad \Lambda, \bullet \vdash e_2 : \tau_{List}}{\Lambda, \bullet \vdash e_1 :: e_2 : \tau_{List}}$ (listCons)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau_{List}$	assumption
4.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$e_2 = v_2$ *or* $(C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
6.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e_1 :: e_2) \longrightarrow (C', e_1' :: e_2)$	listPrependE1
7.	$(C, e_2) \longrightarrow (C', e_2') \Rightarrow (C, e_1 :: e_2) \longrightarrow (C', v_1 :: e_2')$	listPrependE2
8.	$(e_1 = v_1$ *and* $e_2 = v_2$ *and* $v_1 :: v_2 = v) \Rightarrow (C, e_1 :: e_2) \longrightarrow (C, v)$	Definition of values
9.	$e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$	6-8

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List} \quad \Lambda, \bullet \vdash e_2 : \tau_{List}}{\Lambda, \bullet \vdash e_1 \mathbin{@} e_2 : \tau_{List}}$ (listAppend)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau_{List}$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau_{List}$	assumption
4.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$e_2 = v_2$ *or* $(C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
6.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e_1 \mathbin{@} e_2) \longrightarrow (C', e_1' \mathbin{@} e_2)$	Rule listAppandE1
7.	$(e_1 = v_1$ *and* $(C, e_2) \longrightarrow (C', e_2')) \Rightarrow (C, e_1@e_2) \longrightarrow (C', v_1@e_2')$	Rule listAppendE2
8.	$e_1 = v_1 \Rightarrow e_1 = v_1' :: \cdots :: v_n'$ *or* $e_1 = [] : \tau_{List}$	2, Lemma Canonical Forms
9.	$e_2 = v_2 \Rightarrow e_2 = v_{n+1}' :: \cdots :: v_m'$ *or* $e_2 = [] : \tau_{List}$	3, Lemma Canonical Forms
10.	$(e_1 = [] : \tau_{List}$ *and* $e_2 = v_2) \Rightarrow (C, (e_1@e_2)) \longrightarrow (C, v_2)$	Rule listAppendNil
11.	$e_1 = v_1' :: \cdots :: v_n'$ *and* $e_2 = []) \Rightarrow (C, (e_1@e_2)) \longrightarrow (C, v_1)$	Rule listAppendValue
12.	$(e_1 = v_1' :: \cdots :: v_n'$ *and* $e_2 = v_{n+1}' :: \cdots :: v_m') \Rightarrow (C, (e_1@e_2)) \longrightarrow (C, v_1' :: \cdots :: v_m')$	Rule listAppendValue
13.	$e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$	6, 7, 10-12

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List}}{\Lambda, \bullet \vdash head(e_1) : \tau}$ (head)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau_{List}$	assumption
3.	$e_1 = v$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
4.	$e_1 = v \Rightarrow e_1 = v_1 :: v_2$ *or* $e_1 = [] : \tau_{List}$	1, Lemma Canonical Forms
5.	$e_1 = [] : \tau_{List} \Rightarrow (C, head(e_1)) \longrightarrow (C, in_{none}(unit) : \tau\ Option)$	listHeadEmpty
6.	$e_1 = v_1 :: v_2 \Rightarrow (C, head(e_1)) \longrightarrow (C, in_{some}(v_1) : \tau\ Option)$	listHeadValue
7.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, head(e_1)) \longrightarrow (C', head(e_1'))$	listHeadE
8.	$\exists C', e', (C, e) \longrightarrow (C', e')$	5-7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List}}{\Lambda, \bullet \vdash tail(e_1) : \tau_{List}}$ (tail)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau_{List}$	assumption
3.	$e_1 = v$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
4.	$e_1 = v \Rightarrow e_1 = v_1 :: v_2$ *or* $e_1 = [] : \tau_{List}$	2, Lemma Canonical Forms
5.	$e_1 = [] : \tau_{List} \Rightarrow (C, tail(e_1)) \longrightarrow (C, [] : \tau_{List})$	Rule listTailNil
6.	$e_1 = v_1 :: v_2 \Rightarrow (C, tail(e_1)) \longrightarrow (C, v_2)$	Rule listTailCons
7.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, tail(e_1)) \longrightarrow (C', tail(e_1'))$	Rule listTailE
8.	$\exists C', e', (C, e) \longrightarrow (C', e')$	5-7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_{List}}{\Lambda, \bullet \vdash empty(e_1) : Bool}$ (empty)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau_{List}$	assumption
3.	$e_1 = v$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
4.	$e_1 = v \Rightarrow e_1 = v_1 :: v_2$ *or* $e_1 = [] : \tau_{List}$	2, Lemma Canonical Forms
5.	$e_1 = [] : \tau_{List} \Rightarrow (C, empty(e_1)) \longrightarrow (C, true)$	Rule listTailNil
6.	$e_1 = v_1 :: v_2 \Rightarrow (C, empty(e_1)) \longrightarrow (C, false)$	Rule listTailCons
7.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, tail(e_1)) \longrightarrow (C', tail(e_1'))$	Rule listTailE
8.	$\exists C', e', (C, e) \longrightarrow (C', e')$	5-7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \bullet \vdash e_n : \tau_n}{\Lambda, \bullet \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)}$ (record)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$e = (\ell_1 = e_1, \ldots, \ell_n = e_n)$	assumption
3.	$\Lambda, \bullet \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \bullet \vdash e_n : \tau_n$	assumption
4.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1'), \ldots, e_n = v_n$ *or* $(C, e_n) \longrightarrow (C', e_n')$	3, IH
5.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e) \longrightarrow (C', (\ell_1 = e_1', \ldots, \ell_n = e_n))$	1, 2, variantE
	$\vdots$	$\vdots$
	$e_1 = v_1, \ldots, e_{i-1} = v_{i-1}, (C, e_i) \longrightarrow (C', e_i') \Rightarrow (C, e) \longrightarrow (C', (\ell_1 = v_1, \ldots, \ell_i = e_i', \ldots, \ell_n = v_n))$	1, 2, variantE
	$\vdots$	$\vdots$
	$e_1 = v_1, \ldots, e_{n-1} = v_{n-1}, (C, e_n) \longrightarrow (C', e_n') \Rightarrow (C, e) \longrightarrow (C', (\ell_1 = v_1, \ldots, \ell_n = e_n'))$	1, 2, variantE
6.	$e_1 = v_1, \ldots, e_n = v_n \Rightarrow e = v = (\ell_1 = v_1, \ldots, \ell_n = v_n)$	1, definition of values
7.	$e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$	5-6

Case $\dfrac{\Lambda, \bullet \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \quad i \in \{1, \ldots, n\}}{\Lambda, \bullet \vdash e_1.\ell_i : \tau_i}$ (projection)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$ — assumption
3. $i \in \{1, \ldots, n\}$ — assumption
4. $e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$ — 1, 2, IH
5. $e_1 = v_1 \Rightarrow e_1 = (\ell_1 : v_1', \ldots, \ell_n : v_n')$ — 2, Lemma Canonical Forms
6. $(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, e_1.\ell_i) \longrightarrow (C', e_1'.\ell_i)$ — Rule projectionE
7. $e_1 = (\ell_1 : v_1', \ldots, \ell_n : v_n') \Rightarrow (C, e_1.\ell_i) \longrightarrow (C, v_i')$ — Rule projectionV
8. $e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$ — 6-7

Case $\dfrac{\Lambda, \bullet \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Lambda, \bullet \vdash (in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}$ (variant)

1. $e = in_{\ell_i}\ e_i : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n$ — assumption
2. $\Lambda \vdash C$ **ok** — assumption
3. $\Lambda, \bullet \vdash e_i : \tau_i$ — assumption
3. $e_i = v_i$ *or* $(C, e_i) \longrightarrow (C', e_i')$ — 1, 2, IH
4. $(C, e_i) \longrightarrow (C', e_i') \Rightarrow (C, e) \longrightarrow (C', in_{\ell_i}\ e_i' : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — 1, Rule variantE
5. $e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$ — 3, 4

Case $\dfrac{\Lambda, \bullet \vdash e_x : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \quad \Lambda, \bullet \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \bullet \cup \{x_n : \tau_n\} \vdash e_n : \tau}{\Lambda, \bullet \vdash (case\ e_x\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau}$ (case)

1. $\Lambda \vdash C$ **ok** — assumption
2. $e = case\ e_x\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n$ — assumption
3. $\Lambda, \bullet \vdash e_x : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — assumption
4. $\Lambda, \bullet \cup \{x_1 : \tau_1\} \vdash e_1 : \tau \quad \cdots \quad \Lambda, \bullet \cup \{x_n : \tau_n\} \vdash e_n : \tau$ — assumption
5. $e_x = v_x$ *or* $(C, e_x) \longrightarrow (C', e_x')$ — 1, 2, IH
6. $e_x = v_x \Rightarrow \exists i, v_1 : i \in \{1, \ldots, n\} \wedge e_x = in_{\ell_i}\ v_1 : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n$ — 3, Lemma Canonical Forms
7. $(C, e_x) \longrightarrow (C', e_x') \Rightarrow (C, e) \longrightarrow (C', (case\ e_x'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n))$ — caseE
8. $e_x = in_{\ell_i}\ v_1 : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n \Rightarrow (C, e) \longrightarrow (C, [v_i/x_i]e_i)$ — 6, caseV
9. $\exists C', e', (C, e) \longrightarrow (C', e')$ — 7-8

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1}{\Lambda, \bullet \vdash makeTypedVal(\tau_1, e_1) : TypedVal}$ (makeTypedVal)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash e_1 : \tau_1$ — assumption
3. $e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$ — 1, 2, IH
4. $(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, makeTypedVal(\tau_1, e_1)) \longrightarrow (C', makeTypedVal(\tau_1, e_1'))$ — Rule makeTypedValE
5. $e_1 = v_1 \Rightarrow makeTypedVal(\tau_1, e_1) = v$ — Definition of values
6. $e = v$ *or* $\exists C', e', (C, e) \longrightarrow (C', e')$ — 4-5

Case $\dfrac{\Lambda, \bullet \vdash e_1 : TypedVal}{\Lambda, \bullet \vdash tryCast(\tau_1, e_1) : \tau_1\ Option}$ (tryCast)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash e_1 : TypedVal$ — assumption
3. $e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$ — 1, 2, IH
4. $(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, tryCast(\tau_1, e_1)) \longrightarrow (C', tryCast(\tau_1, e_1'))$ — Rule tryCastE
5. $e_1 = v_1 \Rightarrow e_1 = makeTypedVal(\tau_2, v_2)$ — 2, Lemma Canonical Forms
6. $e_1 = makeTypedVal(\tau_2, v_2) \wedge \tau_1 = \tau_2 \Rightarrow (C, tryCast(\tau_1, e_1) \longrightarrow (C, in_{some}\ v_2 : \tau_1\ Option)$ — 5, rule TryCastVOk
7. $e_1 = makeTypedVal(\tau_2, v_2) \wedge \tau_1 \neq \tau_2 \Rightarrow (C, tryCast(\tau_1, e_1) \longrightarrow (C, in_{none}\ unit : \tau_1\ Option)$ — 5, rule TryCastVBad
6. $\exists C', e', (C, e) \longrightarrow (C', e')$ — 4-7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau}{\Lambda, \bullet \vdash \{\ e_1\ \}_{label} : \tau}$ (endLabel)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-Ok	Inspection of Rule C-Ok
3.	$C = (M, R, inOb, rt, out, \tau_{out})$	1, 2, Inversion of Rule C-Ok
4.	$\Lambda, \bullet \vdash e_1 : \tau$	assumption
5.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 4, IH
6.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, \{\ e\ \}_{s(v_2)}) \longrightarrow (C', \{\ e'\ \}_{s(v_2)})$	Rule endLabelE
7.	$e_1 = v_1 \wedge s \neq$ "$monitor''$ $\Rightarrow (C, \{\ e\ \}_{s(v_2)}) \xrightarrow{end_{s(v_2):v_1}} (C, v_1)$	Rule endLabelValue
8.	$e_1 = v_1 \wedge s =$ "$monitor''$ $\Rightarrow (C, \{\ e\ \}_{s(v_2)}) \xrightarrow{end_{s(v_2):v_1}} ((M, R, false, rt, out, \tau_{out}), v_1)$	Rule endLabelMonitor
9.	$\exists C', e' : (C, e) \longrightarrow (C', e')$	5-8

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau}{\Lambda, \bullet \vdash getRt() : Res_{List}}$ (getRT)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-Ok	Inspection of Rule C-Ok
3.	$C = (M, R, inOb, rt, out, \tau_{out})$	1, 2, Inversion of Rule C-Ok
4.	$(C, getRT()) \longrightarrow (C, rt)$	3

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Obligation}{\Lambda, \bullet \vdash makeCFG(e_1) : CFG}$ (makeCFG)

1.	$\Lambda \vdash C$ **ok**	assumption
1.	$\Lambda, \bullet \vdash e_1 : Obligation$	assumption
2.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, IH
3.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, makeCFG(e_1)) \longrightarrow (C', makeCFG(e_1'))$	makeCFGE
4.	$e_1 = v_1 \Rightarrow (C, makeCFG(e_1)) \xrightarrow{begin_{makeCFG(v_1)}} (C, \{makeCFG_\alpha(v)\}_{makeCFG(v_1)})$	makeCFGValue
5.	$\exists C', e', (C, e) \longrightarrow (C', e')$	3, 4

Case $\dfrac{\Lambda, \bullet \vdash e_1 : Event}{\Lambda, \bullet \vdash setOutput(e_1) : Bool}$ (setOutput)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$e_1 : Event$	assumption
3.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-Ok	Inspection of Rule C-Ok
4.	$C = (M, R, inOb, rt, out, \tau_{out})$	1, 3, Inversion of Rule C-Ok
5.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, 3, Inversion of Rule C-Ok
6.	out is a value	4, definition of syntax
7.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, IH
8.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, setOutput(e_1)) \longrightarrow (C', setOutput(e_1'))$	Rule setOutputE
9.	$e_1 = v_1 \Rightarrow$	
a.	$out = in_{none}\ unit : \tau_{out}\ Event\ Option \vee \exists v_2 : out = in_{some}\ v_2 : \tau\ Event\ Option$	5, 6, Lemma CF
b.	$out = in_{some}\ v_2 : \tau\ Event\ Option \Rightarrow (C, setOutput(e_1)) \longrightarrow (C, false)$	4, 9, Rule setOutputSet
c.	$out = in_{none}\ unit : \tau\ Event\ Option \Rightarrow$	
i.	$e_1 = in_{act}\ v_3 : Event \vee e_1 = in_res\ res(v_4, makeTypedVal(\tau, v_5)) : Event$	2, 4, 9, definition of types, Lemma CF
ii.	$e_1 = in_{act}\ v_3 : Event \Rightarrow (C, setOutput(e_1)) \longrightarrow$ $((M, R, inOb, rt, in_{some}(in_{act}\ v : \tau_{out}\ Event) : \tau_{out}\ Event\ Option, \tau_{out}), true)$	4, 9, c, Rule setOutputAct
iii.	$e_1 = in_{res}\ res(v_4, makeTypedVal(\tau, v_5)) \wedge \tau = \tau_{out} \Rightarrow (C, setOutput(e_1)) \longrightarrow$ $((M, R, inOb, rt, in_{some}(in_{res}\ res(v_1, v_2) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option, \tau_{out}), true)$	4, 9, c, Rule setOutputNotSetResGood
iv.	$e_1 = in_{res}\ res(v_4, makeTypedVal(\tau, v_5)) \wedge \tau \neq \tau_{out} \Rightarrow (C, setOutput(e_1)) \longrightarrow (C, false)$	4, 9, c, Rule setOutputNotSetResBad
v.	$\exists C', e' : (C, setOutput(e_1)) \longrightarrow (C, e')$	i-iv
d.	$out = in_{none}\ unit : \tau\ Event\ Option \Rightarrow \exists C', e' : (C, setOutput(e_1)) \longrightarrow (C, e')$	c, c(v)
e.	$\exists C', e' : (C, setOutput(e_1)) \longrightarrow (C, e')$	a, b, d
10.	$e_1 = v_1 \Rightarrow \exists C', e' : (C, setOutput(e_1)) \longrightarrow (C, e')$	9, 9(d)
11.	$\exists C', e' : (C, setOutput(e_1)) \longrightarrow (C, e')$	7, 8, 10

Case $\dfrac{}{\Lambda, \bullet \vdash outputNotSet() : Bool}$ (outputNotSet)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-Ok	Inspection of Rule C-Ok
3.	$C = (M, R, inOb, rt, out, \tau_{out})$	1, 2, Inversion of Rule C-Ok
4.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, 2, Inversion of Rule C-Ok
5.	out is a value	3, definition of syntax
6.	$out = in_{none}\ unit : \tau_{out}\ Event\ Option \vee \exists v_2 : out = in_{some}\ v_2 : \tau\ Event\ Option$	4, 5, Lemma CF
7.	$out = in_{none}\ unit : \tau_{out}\ Event\ Option \Rightarrow (C, outputNotSet()) \longrightarrow (C, true)$	3, Rule outputNotSetTrue
8.	$out = in_{some}\ v_2 : \tau_{out}\ Event\ Option \Rightarrow (C, outputNotSet()) \longrightarrow (C, false)$	3, Rule outputNotSetFalse
9.	$\exists C', e' : (C, outputNotSet()) \longrightarrow (C, e')$	6-8

Case $\dfrac{}{\Lambda, \bullet \vdash getOutput() : Event\ Option}$ (getOutput)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-Ok	Inspection of Rule C-Ok
3.	$C = (M, R, inOb, rt, out, \tau_{out})$	1, 2, Inversion of Rule C-Ok
4.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, 2, Inversion of Rule C-Ok
5.	out is a value	3, definition of syntax
6.	$out = in_{none}\ unit : \tau_{out}\ Event\ Option \vee \exists v_2 : out = in_{some}\ v_2 : \tau\ Event\ Option$	4, 5, Lemma CF
7.	$out = in_{none}\ unit : \tau_{out}\ Event\ Option \Rightarrow (C, getOutput()) \longrightarrow (C, in_{none}\ unit : Event\ Option)$	Rule getOutputNone
8.	$out = in_{some}\ v_2 : \tau_{out}\ Event\ Option \Rightarrow$	
a.	$v_2 = in_{act}\ v_3 : \tau_{out}\ Event\ Option \vee v_2 = in_{res}\ res(v_4, v_5) : \tau_{out}\ Event\ Option$	4, 5, 8, Lemma CF
b.	$v_2 = in_{act}\ v_3 : \tau_{out}\ Event\ Option \Rightarrow (C, getOutput()) \longrightarrow (C, in_{some}\ (in_{act}\ v_3 : Event) : Event\ Option)$	Rule getOutputSomeAct
c.	$v_2 = in_{res}\ res(v_4, v_5) : \tau_{out}\ Event\ Option \Rightarrow (C, getOutput()) \longrightarrow (C, in_{some}\ (in_{res}\ res(v_4, makeTypedVal(\tau_{out}, v_5)) : Event) : Event\ Option)$	Rule getOutputSomeRes
d.	$\exists C', e' : (C, getOutput()) \longrightarrow (C, e')$	a-c
9.	$out = in_{some}\ v_2 : \tau_{out}\ Event\ Option \Rightarrow \exists C', e' : (C, getOutput()) \longrightarrow (C, e')$	8, 8(d)
10.	$\exists C', e' : (C, getOutput()) \longrightarrow (C, e')$	6, 7, 9

Case $\dfrac{\Lambda, \bullet \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)}{\Lambda, \bullet \vdash monitor(\tau_1, e_1) : \tau_1\ Event}$ (monitor)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$	assumption
3.	$e_1 = v_1$ *or* $(C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
4.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, monitor(\tau_1, e_1)) \longrightarrow (C', monitor(\tau_1, e_1'))$	Rule monitorE
5.	$\Lambda \vdash C$ **ok** is only derivable by Rule C-ok	Inspection of Rule C-Ok
6.	$C = (M, R, inOb, rt, out, \tau_{out})$	1, 5, Rule C-Ok
7.	$e_1 = v_1 \Rightarrow (C, monitor(\tau_1, v_1)) \xrightarrow{begin_{monitor(v_1)}} ((M, R, true, inOb, rt, out, \tau_{out}), \{e_{monitor}(\tau_1, v_1)\}_{monitor(v_1)})$	6, Rule monitorV
8.	$\exists C', e', (C, e) \longrightarrow (C', e')$	3, 4, 7

Case $\dfrac{\Lambda, \bullet \vdash e_1 : String \quad \Lambda, \bullet \vdash e_2 : TypedVal}{\Lambda, \bullet \vdash invoke(e_1, e_2) : TypedVal\ Option}$ (invoke)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : String$	assumption
3.	$\Lambda, \bullet \vdash e_2 : TypedVal$	assumption
4.	$e_1 = v_1 \ \ or \ \ (C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$e_1 = v_1 \Rightarrow e_1 = s$	2, Lemma Canonical Forms
6.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, invoke(e_1, e_2)) \longrightarrow (C', invoke(e_1', e_2))$	Rule invokeE1
7.	$e_2 = v_2 \ \ or \ \ (C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
8.	$e_1 = s_1 \wedge e_2 \longrightarrow e_2' \Rightarrow (C, invoke(e_1, e_2)) \longrightarrow (C', invoke(e_1, e_2'))$	5, Rule invokeE2
9.	$e_1 = s_1 \wedge e_2 = v_2 \Rightarrow$	
a.	$(\forall s, f : (s, f) \notin F) \vee (\exists x_1, x_2, \tau_1, \tau_2, e : (s, (fun\ x_1(x_2 : \tau_1) : \tau_2 = e)) \in F)$	definition of syntax
b.	$(\forall s, f : (s, f) \notin F) \Rightarrow (C, invoke(e_1, e_2)) \longrightarrow (C, in_{none}\ unit : TypedVal\ Option$	5, 9, invokeValNotExists
c.	$(\exists x_1, x_2, \tau_1, \tau_2, e : (s, (fun\ x_1(x_2 : \tau_1) : \tau_2 = e)) \in F) \Rightarrow$	
i.	$v_2 = makeTypedVal(\tau_3, v_2')$	3, 9, Lemma Canonical Forms
ii.	$\tau_1 = \tau_3 \Rightarrow (C, invoke(e_1, e_2)) \longrightarrow (C, in_{some}\ makeTypedVal(\tau_2,$ $call((fun\ x_1(x_2 : \tau_1) : \tau_2 = e), v_2')) : TypedVal\ Option)$	9, c, i, Rule invokeValueExistsOk
iii.	$\tau_1 \neq \tau_3 \Rightarrow (C, invoke(e_1, e_2)) \longrightarrow (C, in_{none}\ unit : TypedVal\ Option)$	9, c, i, Rule invokeValueExistsBad
iv.	$\exists C', e', (C, e) \longrightarrow (C', e')$	ii, iii
d.	$(\exists x_1, x_2, \tau_1, \tau_2, e : (s, (fun\ x_1(x_2 : \tau_1) : \tau_2 = e)) \in F) \Rightarrow \exists C', e', (C, e) \longrightarrow (C', e')$	c, c(iv)
e.	$\exists C', e', (C, e) \longrightarrow (C', e')$	a, b, d
10.	$e_1 = s_1 \wedge e_2 = v_2 \Rightarrow \exists C', e', (C, e) \longrightarrow (C', e')$	9, 9(e)
11.	$\exists C', e', (C, e) \longrightarrow (C', e')$	4, 6, 7, 8, 10

Case $\dfrac{\Lambda, \bullet \vdash e_1 : \tau_1 \to \tau_2 \quad \Lambda, \bullet \vdash e_2 : \tau_1}{\Lambda, \bullet \vdash call(e_1, e_2) : \tau_2}$ (call)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 : \tau_1 \to \tau_2$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau_1$	assumption
4.	$e_1 = v_1 \ \ or \ \ (C, e_1) \longrightarrow (C', e_1')$	1, 2, IH
5.	$e_2 = v_2 \ \ or \ \ (C, e_2) \longrightarrow (C', e_2')$	1, 3, IH
6.	$(C, e_1) \longrightarrow (C', e_1') \Rightarrow (C, call(e_1, e_2)) \longrightarrow (C, call(e_1', e_2))$	callE1
7.	$e_1 = v_1 \Rightarrow \exists x_1, x_2, e_z : e_1 = (fun\ x_1(x_2 : \tau_1) = e_z)$	2, Lemma Canonical Forms
8.	$e_1 = v_1 \ \ and \ \ (C, e_2) \longrightarrow (C', e_2') \Rightarrow (C, call(e_1, e_2)) \longrightarrow (C, call(v_1, e_2'))$	7, callE2
9.	$\Lambda \vdash C$ **ok** is only derivable by rule C-ok	inspection of the rules
10.	$C = (M, (F, pols, os, vc), inOb, rt, out, \tau_{out})$ $\Lambda, \bullet \vdash inOb : Bool$	9, inversion of Rule C-ok
11.	$\exists v, inOb = v$	definition of C
12.	$inOb \in \{true, false\}$	10, 11, Lemma Canonical Forms
13.	$e_1 = v_1 \ \ and \ \ e_2 = v_2 \Rightarrow$	
a.	$\exists s : (s, v_1) \in F \Rightarrow$	
i.	$inOb = true \Rightarrow$ $(C, call(e_1, e_2)) \xrightarrow{begin_{x_1(v2)}}$ $((M, (F, pols, os, vc), true,$ $rt \ @ \ res(act(s, makeTypedVal(\tau_1, v_2)), makeTypedVal(\tau_2, [v_1/x_1, v_2/x_2]e))$ $:: [] : Res_{List}, out, \tau_{out})$ $\{[v_1/x_1, v_2/x_2]e\}_{x_1(v_2)})$	7, 13, a, callFromObligation
ii.	$inOb = false \Rightarrow$ $(C, call(e_1, e_2)) \xrightarrow{being_{s(v)}}$ $((M, (F, pols, os, vc), false, rt, in_{none}\ unit : \tau_2\ Event\ Option, \tau_2), e_{procEvt})$	7, 13, a, callFromApplication
b.	$f \notin range(F) \Rightarrow$	
i.	$(\ldots, onTrigger = v_1, \ldots) \in pols \wedge v_1 = (fun\ x_1(x_2 : Event) : Unit = e) \Rightarrow$ $(C, call(e_1, e_2)) \xrightarrow{begin_{x_1(v_2)}}$ $((M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out}), \{[v_1/x_1, v_2/x_2]e\}_{x_1(x_2)}$	7, 13, b, callOnTrigger
ii.	$(\ldots, onObligation = v_1, \ldots) \in pols \wedge v_1 = (fun\ x_1(x_2 : Res_{List}) : Unit = e) \Rightarrow$ $(C, call(e_1, e_2)) \xrightarrow{begin_{x_1(v_2)}}$ $((M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out}), \{[v_1/x_1, v_2/x_2]e\}_{x_1(x_2)}$ $(C, call(e_1, e_2)) \longrightarrow (C, \{[v/x]e\}_{s(v)})$	7, 13, b, callOnObligation
iii.	$(\forall pol \in pols : pol = (\ldots, onTrigger = f_1, onObligation = f_2, \ldots) \Rightarrow v_1 \notin \{f_1, f_2\}) \Rightarrow$ $(C, call(e_1, e_2)) \xrightarrow{begin_{x_1(v_2)}} (C, \{[v_2/x_2, v_1/x_1]e\}_{x_1(v_2)}$	7, 13, b, callNonMonitoredfunction
14.	$\exists C', e' : (C, call(e_1, e_2)) \longrightarrow (C', e')$	13a(i), 13a(ii), 13b(i), 13b(ii), 13b(iii)

Lemma 10 (Monitor Type). $\Lambda, \bullet \vdash v : (evt : \tau\ Event \times pols : Pol_{List} \times os : OS \times vc : VC) \Rightarrow \Lambda, \bullet \vdash e_{monitor}(\tau, v) : \tau\ Event$

The proof of this lemma is trivial though uninteresting. The proof technique is to derive the proof tree of $\Lambda, \bullet \vdash e_{monitor}(\tau, v) : \tau\ Event$

Lemma 11 (Preservation). $\Lambda \vdash (C, e) : \tau \wedge (C, e) \longrightarrow (C', e') \Rightarrow \exists \Lambda' : \Lambda' \vdash (C', e') : \tau \ \wedge \ \Lambda \subseteq \Lambda'$

We will instead prove the equivalent statement:
$(\Lambda \vdash C$ **ok** $\ \wedge \ \Lambda, \bullet \vdash e : \tau \ \wedge \ (C, e) \longrightarrow (C', e')) \Longrightarrow \exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \ \Lambda', \bullet \vdash e' : \tau \ \wedge \ \Lambda \subseteq \Lambda')$.
It can be shown that these two statements are equivalent by inversion of rule TConfig.
It is assumed throughout this proof that any Λ is a subset of itself.

Proof. By induction on the derivation of $(C, e) \longrightarrow (C', e')$

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 \wedge e_2) \longrightarrow (C', e_1' \wedge e_2)}$ (andE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 \wedge e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : Bool$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : Bool$	2, Inversion Lemma
6.	$\tau = Bool$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : Bool \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
8.	$\Lambda', \bullet \vdash e_2 : Bool$	5, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash e_1' \wedge e_2 : \tau$	6-8, Rule con

Result is from 7, 9

Case $\dfrac{}{(C, true \wedge e_2) \longrightarrow (C, e_2)}$ (andTrue)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash true \wedge e_2 : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash e_2 : \tau$	2, 3, Inversion Lemma

Result is from 1, 4

Case $\dfrac{}{(C, false \wedge e_2) \longrightarrow (C, false)}$ (andFalse)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash false \wedge e_2 : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash false : \tau$	3, Rule boolVal

Result is from 1, 4

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 \vee e_2) \longrightarrow (C', e_1' \vee e_2)}$ (orE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 \vee e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : Bool$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : Bool$	2, Inversion Lemma
6.	$\tau = Bool$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : Bool \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
8.	$\Lambda', \bullet \vdash e_2 : Bool$	5, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash e_1' \vee e_2 : \tau$	6-8, Rule or

Result is from 7, 9

Case $\dfrac{}{(C, true \vee e_2) \longrightarrow (C, true)}$ (orTrue)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash true \vee e_2 : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash true : \tau$	3, Rule boolVal

Result is from 1, 4

Case $\dfrac{}{(C, false \vee e_2) \longrightarrow (C, e_2)}$ (orFalse)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash false \vee e_2 : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash e_2 : \tau$	2, 3, Inversion Lemma

Result is from 1, 4

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, \neg e_1) \longrightarrow (C', \neg e_1')}$ (notE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash \neg e_1 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : Bool$	2, Inversion Lemma
5.	$\tau = Bool$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : bool \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash \neg e_1' : \tau$	5, 6, Rule negation

Result is from 6, 7

Case $\dfrac{}{(C, \neg false) \longrightarrow (C, true)}$ (notFalse)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash \neg false : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash true : \tau$	3, Rule boolVal

Result is from 1, 4

Case $\dfrac{}{(C, \neg true) \longrightarrow (C, false)}$ (notTrue)

1. This case is analogous to case notFalse

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 == e_2) \longrightarrow (C', e_1' == e_2)}$ (eqE1)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 == e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau_1$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau_1$	2, Inversion Lemma
6.	$\tau_1 = \{Int, Bool, String\}$	2, Inversion Lemma
7.	$\tau = Bool$	2, Inversion Lemma
8.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : \tau_1 \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
9.	$\Lambda', \bullet \vdash e_2 : \tau_1$	5, 8, Λ-weakening Lemma
10.	$\Lambda', \bullet \vdash e_1' == e_2 : \tau$	6-9, Rule equality

Result is from 8, 10

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, v_1 == e_2) \longrightarrow (C', v_1 == e_2')}$ (eqE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash v_1 == e_2 : \tau$	assumption
3.	$(C, e_2) \longrightarrow (C', e_2')$	assumption
4.	$\Lambda, \bullet \vdash v_1 : \tau_1$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau_1$	2, Inversion Lemma
6.	$\tau_1 = \{Int, \ Bool, \ String\}$	2, Inversion Lemma
7.	$\tau = Bool$	2, Inversion Lemma
8.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_2' : \tau_1 \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 5, IH
9.	$\Lambda', \bullet \vdash v_1 : \tau_1$	4, 8, Λ-weakening Lemma
10.	$\Lambda', \bullet \vdash v_1 == e_2' : \tau$	6-9, Rule Equality

Result is from 8, 10

Case $\dfrac{n_1 = n_2}{(C, n_1 == n_2) \longrightarrow (C, true)}$ (eqIntTrue)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash n_1 == n_2 : \tau$ assumption
3. $\tau = Bool$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash true : \tau$ 3, Rule boolVal

Result is from 1, 4

Case $\dfrac{n_1 \neq n_2}{(C, n_1 == n_2) \longrightarrow (C, false)}$ (eqIntFalse)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash n_1 == n_2 : \tau$ assumption
3. $\tau = Bool$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash false : \tau$ 3, Rule boolVal

Result is from 1, 4

Case $\dfrac{s_1 = s_2}{(C, s_1 == s_2) \longrightarrow (C, true)}$ (eqStrTrue)

1. This case is analagous to case eqIntTrue

Case $\dfrac{s_1 \neq s_2}{(C, s_1 == s_2) \longrightarrow (C, false)}$ (eqStrFalse)

1. This case is analagous to case eqIntFalse

Case $\dfrac{}{(C, true == b_2) \longrightarrow (C, b_2)}$ (eqBoolTrue)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash true == b_2 : \tau$ assumption
3. $\tau = Bool$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash b_2 : \tau$ 2, 3, Inversion Lemma

Result is from 1, 4

Case $\dfrac{}{(C, false == b_2) \longrightarrow (C, \neg b_2)}$ (eqBoolFalse)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash false == b_2 : \tau$ assumption
3. $\tau = Bool$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash b_2 : Bool$ 2, Inversion Lemma
5. $\Lambda, \bullet \vdash \neg b_2 : \tau$ 3, 4, Rule negation

Result is from 1, 5

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 + e_2) \longrightarrow (C', e_1' + e_2)}$ (addE1)

1. $\Lambda \vdash C$ **ok** assumption
2. $\Lambda, \bullet \vdash e_1 + e_2 : \tau$ assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ assumption
4. $\Lambda, \bullet \vdash e_1 : Int$ 2, Inversion Lemma
5. $\Lambda, \bullet \vdash e_2 : Int$ 2, Inversion Lemma
6. $\tau = Int$ 2, Inversion Lemma
7. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : Int \wedge \ \Lambda \subseteq \Lambda')$ 1, 3, 4, IH
8. $\Lambda', \bullet \vdash e_2 : Int$ 5, 7, Λ-weakening Lemma
9. $\Lambda', \bullet \vdash e_1' + e_2 : Int$ 6-8, Rule add

Result is from 7, 9

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, n_1 + e_2) \longrightarrow (C', n_1 + e_2')}$ (addE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash n_1 + e_2 : \tau$	assumption
3.	$(C, e_2) \longrightarrow (C', e_2')$	assumption
4.	$\Lambda, \bullet \vdash e_2 : Int$	2, Inversion Lemma
5.	$\tau = Int$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_2' : Int \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash n_1 : Int$	Rule IntVal
8.	$\Lambda', \bullet \vdash n_1 + e_2' : Int$	5-7, Rule add

Result is from 6, 8

Case $\dfrac{n_1 +_\alpha n_2 = n}{(C, n_1 + n_2) \longrightarrow (C, n)}$ (addValue)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash n_1 + n_2 : \tau$	assumption
3.	$\tau = Int$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash n : \tau$	3, Rule intVal

Result is from 1, 4

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1; e_2) \longrightarrow (C', e_1'; e_2)}$ (sequenceE1)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1; e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau'$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : \tau' \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash e_2 : \tau$	5, 6, Λ-weakening Lemma
8.	$\Lambda', \bullet \vdash e_1'; e_2 : \tau$	6, 7, Rule sequence

Result is from 6, 8

Case $\dfrac{}{(C, v_1; e_2) \longrightarrow (C, e_2)}$ (sequenceE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash v_1; e_2 : \tau$	assumption
3.	$\Lambda, \bullet \vdash e_2 : \tau$	2, Inversion Lemma

Result is from 1, 3

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, if\ e_1\ then\ e_2\ else\ e_3) \longrightarrow (C', if\ e_1'\ then\ e_2\ else\ e_3)}$ (ifE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : Bool$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau$	2, Inversion Lemma
6.	$\Lambda, \bullet \vdash e_3 : \tau$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : Bool \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
8.	$\Lambda', \bullet \vdash e_2 : \tau$	5, 7, Λ-weaking Lemma
9.	$\Lambda', \bullet \vdash e_3 : \tau$	6, 7, Λ-weaking Lemma
10.	$\Lambda', \bullet \vdash if\ e_1'\ then\ e_2\ else\ e_3 : \tau$	7-9, Rule if

Result is from 7, 10

Case $\dfrac{}{(C, if\ true\ then\ e_2\ else\ e_3) \longrightarrow (C, e_2)}$ (ifTrue)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash if\ true\ then\ e_2\ else\ e_3 : \tau$ — assumption
3. $\Lambda, \bullet \vdash e_2 : \tau$ — 2, Inversion Lemma

Result is from 1, 3

Case $\dfrac{}{(C, if\ false\ then\ e_2\ else\ e_3) \longrightarrow (C, e_3)}$ (ifFalse)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash if\ false\ then\ e_2\ else\ e_3 : \tau$ — assumption
3. $\Lambda, \bullet \vdash e_3 : \tau$ — 2, Inversion Lemma

Result is from 1, 3

Case $\dfrac{}{(C, while(e_1)\ \{e_2\}) \longrightarrow (C, if\ e_1\ then\ (e_2;\ while(e_1)\ \{e_2\})\ else\ false)}$ (whileE)

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

Result is from 1, 8

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, let\ x = e_1\ in\ e_2\ end) \longrightarrow (C', let\ x = e_1'\ in\ e_2\ end)}$ (letE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash let\ x = e_1\ in\ e_2\ end : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : \tau_1$ — 2, Inversion Lemma
5. $\Lambda, \{x : \tau_1\} \vdash e_2 : \tau_2$ — 2, Inversion Lemma
6. $\tau = \tau_2$ — 2, Inversion Lemma
7. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau_1 \wedge\ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
8. $\Lambda', \{x : \tau_1\} \vdash e_2 : \tau_2$ — 5, 7, Λ-weakening Lemma
9. $\Lambda', \bullet \vdash let\ x = e_1'\ in\ e_2\ end : \tau$ — 6, 7, 8, Rule let

Result is from 7, 9

Case $\dfrac{}{(C, let\ x = v\ in\ e_2\ end) \longrightarrow (C, [v/x]e_2)}$ (letValue)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash let\ x = v\ in\ e_2\ end : \tau$ — assumption
3. $\Lambda, \bullet \vdash v : \tau_1$ — 2, Inversion Lemma
4. $\Lambda, \{x : \tau_1\} \vdash e_2 : \tau_2$ — 2, Inversion Lemma
5. $\tau = \tau_2$ — 2, Inversion Lemma
6. $\Lambda, \bullet \vdash [v/x]e_2 : \tau$ — 3-5, Substitution Lemma

Result is from 1, 6

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, ref\ e_1) \longrightarrow (C', ref\ e_1')}$ (refE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash ref\ e_1 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau_1$	2, Inversion Lemma
5.	$\tau = \tau_1\ Ref$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau_1 \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash ref\ e_1' : \tau$	5, 6, Rule createRef

Result is from 6, 7

Case $\dfrac{\ell \notin dom(M)}{((M, R, inOb, rt, out, \tau_{out}), ref\ v) \longrightarrow ((M \cup \{(\ell, v)\}, R, inOb, rt, out, \tau_{out}), \ell)}$ (refValue)

1.	let $(M, R, inOb, rt, out, \tau_{out}) = C$	assumption
2.	let $(M \cup \{(\ell, v)\}, R, inOb, rt, out, \tau_{out}) = C'$	assumption
3.	$\Lambda \vdash C$ **ok**	assumption
4.	$\Lambda, \bullet \vdash ref\ v : \tau$	assumption
5.	$\Lambda, \bullet \vdash v : \tau'$	4, Inversion Lemma
6.	$\tau = \tau'\ Ref$	4, Inversion Lemma
7.	$M : \Lambda$	1, 3, C-Inversion Lemma
8.	8 is only derivable by Rule TMem	Inspection of $M : \Lambda$ rules
9.	$M = \{(\ell_1, v_1), \ldots, (\ell_n, v_n)\}$	7, 8, Inversion of Rule TMem
10.	$\Lambda = \{(\ell_1 : \tau_1), \ldots, (\ell_n : \tau_n)\}$	7, 8, Inversion of Rule TMem
11.	$\forall i \in \{1, \ldots, n\}. \Lambda, \bullet \vdash v_i : \tau_i$	7, 8, Inversion of RuleTMem
12.	let $M' = M \cup \{(\ell, v)\}$	assumption
13.	let $\Lambda' = \Lambda \cup \{(\ell : \tau')\}$	assumption
14.	$\Lambda \subseteq \Lambda'$	Definition of $\subseteq$
15.	$\forall i \in \{1, \ldots, n\}. \Lambda', \bullet \vdash v_i : \tau_i$	11, 14, Λ-weakening Lemma
16.	$\Lambda', \bullet \vdash v : \tau'$	5, 14, Λ-weakening Lemma
17.	$M' : \Lambda'$	12, 13, 15, 16
18.	$\Lambda', \bullet \vdash C'$ **ok**	2, 3, 12-14, 17, C-weakening Lemma
19.	$\Lambda', \bullet \vdash \ell : \tau$	6, 14, Rule location

Result is from 14, 18, 19

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, !e_1) \longrightarrow (C', !e_1')}$ (derefE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash !e_1 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau\ Ref$	2, Inversion Lemma
5.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau\ Ref \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
6.	$\Lambda', \bullet \vdash !e_1' : \tau$	5, rule accessRef

Result is from 5, 6

Case $\dfrac{}{(M' \cup \{(\ell, v)\}, \ldots), !\ell) \longrightarrow (M' \cup \{(\ell, v)\}, \ldots), v)}$ (derefValue)

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

Result is from 1, 8

Case $\dfrac{(C, e_1) \longrightarrow (C, e_1')}{(C, e_1 := e_2) \longrightarrow (C', e_1' := e_2)}$ (assignE1)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 := e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau_1\ Ref$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau_1$	2, Inversion Lemma
6.	$\tau = Unit$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau_1\ Ref \wedge \Lambda \subseteq \Lambda')$	1, 3, 4, IH
8.	$\Lambda', \bullet \vdash e_2 : \tau_1$	5, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash e_1' := e_2 : \tau$	6-8, Rule assignment

Result is from 7, 9

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, \ell_1 := e_2) \longrightarrow (C', \ell_1 := e_2')}$ (assignE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash \ell_1 := e_2 : \tau$	assumption
3.	$(C, e_2) \longrightarrow (C', e_2')$	assumption
4.	$\Lambda, \bullet \vdash \ell_1 : \tau_1\ Ref$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau_1$	2, Inversion Lemma
6.	$\tau = Unit$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\ \wedge \Lambda', \bullet \vdash e_2' : \tau_1 \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 5, IH
8.	$\Lambda', \bullet \vdash \ell_1 : \tau_1\ Ref$	4, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash \ell_1 := e_2' : \tau$	6-8, Rule assignment

Result is from 7, 9

Case $\dfrac{}{((M' \cup \{(\ell, v)\}, R, inOb, rt, out, \tau_{out}), \ell := v') \longrightarrow ((M' \cup \{(\ell, v')\}, R, inOb, rt, out, \tau_{out}), unit)}$ (assignValue)

1.	$\Lambda \vdash (M' \cup \{(\ell, v)\}, R, inOb, rt, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash \ell := v' : \tau$	assumption
3.	$M' \cup \{(\ell, v)\} : \Lambda$	1, C-Inversion Lemma
4.	$\Lambda \vdash R$ **ok**	1, C-Inversion Lemma
5.	$\Lambda, \bullet \vdash inOb : Bool$	1, C-Inversion Lemma
6.	$\Lambda, \bullet \vdash rt : Res_{List}$	1, C-Inversion Lemma
7.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, C-Inversion Lemma
8.	$\Lambda, \bullet \vdash \ell : \tau'\ Ref$	2, Inversion Lemma
9.	$\Lambda, \bullet \vdash v' : \tau'$	2, Inversion Lemma
10.	$\tau = Unit$	2, Inversion Lemma
11.	3 is only derivable by Rule TMem	Inspection of $M : \Lambda$ rules
12.	$\Lambda = \Lambda' \cup \{\ell : \tau'\}$	3, 11, Inversion of Rule TMem
13.	$\forall i \in \{1, \ldots, n\}.\Lambda, \bullet \vdash v_i : \tau_i$	3, 11, Inversion of Rule TMem
14.	$\forall i \in \{1, \ldots, n-1\}.\Lambda, \bullet \vdash v_i : \tau_i$	13, Def of $\forall$, Def of $\leq$
15.	$M' \cup \{(\ell, v')\} : \Lambda$	9, 14, Rule TMem
16.	$\Lambda \vdash (M' \cup \{\ell, v')\}, R, inOb, rt, out, \tau_{out})$ **ok**	4-7, 15, Rule C-ok
17.	$\Lambda, \bullet \vdash unit : \tau$	10, Rule unitVal

Result is from 16, 17

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 :: e_2) \longrightarrow (C', e_1' :: e_2)}$ (listPrependE1)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 :: e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\tau = \tau_{1_{List}}$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_1 : \tau_1$	2, Inversion Lemma
6.	$\Lambda, \bullet \vdash e_2 : \tau_{1_{List}}$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : \tau_1 \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 5, IH
8.	$\Lambda', \bullet \vdash e_2 : \tau_{1_{List}}$	5, 7, Lemma Λ-Weakening
9.	$\Lambda', \bullet \vdash e_1' :: e_2 : \tau$	4, 7, 8, Rule listCons
	Result is from 7, 9	

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, v_1 :: e_2) \longrightarrow (C', v_1 :: e_2')}$ (listPrependE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash v_1 :: e_2 : \tau$	assumption
3.	$(C, e_2) \longrightarrow (C', e_2')$	assumption
4.	$\tau = \tau_{1_{List}}$	1, Inversion Lemma
5.	$\Lambda, \bullet \vdash v_1 : \tau_1$	1, Inversion Lemma
6.	$\Lambda, \bullet \vdash e_2 : \tau_{1_{List}}$	1, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_2' : \tau_{1_{List}} \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 6, IH
8.	$\Lambda', \bullet \vdash v_1 : \tau_1$	5, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash v_1 :: e_2' : \tau$	4, 7, 8, Rule listCons
	Result is from 7, 8	

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1 \ @ \ e_2) \longrightarrow (C', e_1' \ @ \ e_2)}$ (listAppendE1)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash e_1 @ e_2 : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau'_{List}$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau'_{List}$	2, Inversion Lemma
6.	$\tau = \tau'_{List}$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : \tau'_{List} \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
8.	$\Lambda', \bullet \vdash e_2 : \tau'_{List}$	5, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash e_1' \ @ \ e_2 : \tau$	6-8, Rule listAppend
	Result is from 7, 9	

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, v_1 \ @ \ e_2) \longrightarrow (C', v_1 \ @ \ e_2')}$ (listAppendE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash v_1 @ e_2 : \tau$	assumption
3.	$(C, e_2) \longrightarrow (C', e_2')$	assumption
4.	$\Lambda, \bullet \vdash v_1 : \tau'_{List}$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : \tau'_{List}$	2, Inversion Lemma
6.	$\tau = \tau'_{List}$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_2' : \tau'_{List} \wedge \ \Lambda \subseteq \Lambda')$	1,3, 5, IH
8.	$\Lambda', \bullet \vdash v_1' : \tau'_{List}$	4, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash v_1 \ @ \ e_2' : \tau$	6-8, Rule listAppend
	Result is from 7, 9	

Case $\dfrac{}{(C, ([] : \tau'_{List}) \mathbin{@} v_3) \longrightarrow (C, v_3)}$ (appendNil)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash ([] : \tau'_{List} \mathbin{@} v_3) : \tau$	assumption
3.	$\tau = \tau'_{List}$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash v_3 : \tau$	2, 3, Inversion Lemma

Result is from 1, 4

Case $\dfrac{}{(C, (v_1 :: v_2) \mathbin{@} v_3) \longrightarrow (C, v_1 :: (v_2 \mathbin{@} v_3))}$ (appendCons)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash (v_1 :: v_2) \mathbin{@} v_3) : \tau$	assumption
3.	$\Lambda, \bullet \vdash (v_1 :: v_2) : \tau'_{List}$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash v_3 : \tau'_{List}$	2, Inversion Lemma
5.	$\tau = \tau'_{List}$	2, Inversion Lemma
6.	$\Lambda, \bullet \vdash v_1 : \tau'$	3, Inversion Lemma
7.	$\Lambda, \bullet \vdash v_2 : \tau'_{List}$	3, Inversion Lemma
8.	$\Lambda, \bullet \vdash (v_2 \mathbin{@} v_3) : \tau'_{List}$	4, 7, Rule listAppend
9.	$\Lambda, \bullet \vdash v_1 :: (v_2 \mathbin{@} v_3) : \tau$	5, 6, 8, Rule listCons

Result is from 1, 9

Case $\dfrac{(C, e_1) \longrightarrow (C', e'_1)}{(C, head(e_1)) \longrightarrow (C', head(e'_1))}$ (listHeadE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash head(e_1) : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e'_1)$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau'_{List}$	2, Inversion Lemma
5.	$\tau = \tau'\ Option$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e'_1 : \tau'_{List} \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash head(e'_1) : \tau$	5, 6, Rule head

Result is from 6, 7

Case $\dfrac{}{(C, head(v_1 :: \cdots :: [] : \tau'_{List})) \longrightarrow (C, in_{some}(v_1) : \tau'\ Option)}$ (listHeadCons)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash head(v_1 :: \cdots :: [] : \tau'_{List}) : \tau'\ Option$	assumption
3.	$\Lambda, \bullet \vdash (v_1 :: \cdots :: [] : \tau'_{List}) : \tau'_{List}$	2, Inversion Lemma
4.	$\tau = \tau'\ Option$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash v_1 : \tau'$	3, Inversion Lemma
6.	$\Lambda, \bullet \vdash (in_{some}(v_1) : \tau'\ Option) : \tau$	4, 5, Def of types, Rule variant

Result is from 1, 6

Case $\dfrac{}{(C, head([] : \tau'_{List})) \longrightarrow (C, in_{none}(unit) : \tau'\ Option)}$ (listHeadNil)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash head([] : \tau'_{List}) : \tau$	assumption
3.	$\tau = \tau'\ Option$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash unit : Unit$	Rule unitVal
5.	$\Lambda, \bullet \vdash (in_{none}(unit) : \tau'\ Option) : \tau$	3, 4, Def of types, Rule variant

Result is from 1, 5

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, tail(e_1)) \longrightarrow (C', tail(e_1'))}$ (listTailE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash tail(e_1) : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau'_{List}$	2, Inversion Lemma
5.	$\tau = \tau'_{List}$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau'_{List} \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash tail(e_1') : \tau$	5, 6, Rule tail

Result is from 6, 7

Case $\dfrac{}{(C, tail(v_1 :: v_2)) \longrightarrow (C, v_2)}$ (listTailCons)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash tail(v_1 :: v_2) : \tau$	assumption
3.	$\Lambda, \bullet \vdash (v_1 :: v_2) : \tau'_{List}$	2, Inversion Lemma
4.	$\tau = \tau'_{List}$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash v_2 : \tau$	3, 4, Inversion Lemma

Result is from 1, 5

Case $\dfrac{}{(C, tail([] : \tau'_{List})) \longrightarrow (C, [] : \tau'_{List})}$ (listTailNil)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash tail([] : \tau'_{List}) : \tau$	assumption
3.	$\tau = \tau'_{List}$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash ([] : \tau'_{List}) : \tau$	2, 3, Inversion Lemma

Result is from 1, 4

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, empty(e_1)) \longrightarrow (C', empty(e_1'))}$ (listEmptyE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash empty(e_1) : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : \tau'_{List}$	2, Inversion Lemma
5.	$\tau = Bool$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau'_{List} \wedge \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash empty(e_1') : \tau$	5, 6, Rule empty

Result is from 6, 7

Case $\dfrac{}{(C, empty([] : \tau'_{List})) \longrightarrow (C, true)}$ (listEmptyNil)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash empty([] : \tau'_{List}) : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash true : \tau$	3, Rule boolVal

Result is from 1, 4

Case $\dfrac{}{(C, empty(v_1 :: v_2)) \longrightarrow (C, false)}$ (listEmptyCons)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash empty(v_1 :: v_2) : \tau$	assumption
3.	$\tau = Bool$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash false : \tau$	3, Rule boolVal

Result is from 1, 4

Case $\dfrac{\forall j(1 \le j < i).e_j = v_j \quad (C, e_i) \longrightarrow (C', e_i') \quad i \in \{1, \ldots, n\}}{(C, (l_1 = e_1, \ldots, l_n = e_n)) \longrightarrow (C', (l_1 = e_1, \ldots, l_i = e_i', \ldots, l_n = e_n))}$ (recordE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash (l_1 = e_1, \ldots, l_n = e_n) : \tau$ — assumption
3. $(C, e_i) \longrightarrow (C', e_i')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : \tau_1 \ldots \Lambda, \bullet \vdash e_n : \tau_n$ — 2, Inversion Lemma
5. $\tau = (\tau_1 \times \cdots \times \tau_n)$ — 2, Inversion Lemma
6. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_i' : \tau_i \wedge \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
7. $\Lambda', \bullet \vdash e_1 : \tau_1 \ldots \Lambda', \bullet \vdash e_n : \tau_n$ — 4, 6, Λ-weakening Lemma
8. $\Lambda', \bullet \vdash (l_1 = e_1, \ldots, l_i = e_i', \ldots, l_n = e_n) : \tau$ — 5-7, Rule variant

Result is from 6, 8

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, e_1.\ell_i) \longrightarrow (C', e_1'.\ell_i)}$ (projectionE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash e_1.\ell_i : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$ — 2, InversionLemma
5. $\tau = \tau_i$ — 2, Inversion Lemma
6. $i \in \{1, \ldots, n\}$ — 2, Inversion Lemma
7. $\exists \Lambda' : (\Lambda' \vdash C'\ ok \ \wedge \ \Lambda', \bullet \vdash e_1' : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \ \wedge \ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
8. $\Lambda', \bullet \vdash e_1'.\ell_i : \tau$ — 5-7, Rule projection

Result is from 7, 8

Case $\dfrac{i \in \{1, \ldots, n\}}{(C, (\ell_1 = v_1, \ldots, \ell_n = v_n).\ell_i) \longrightarrow (C, v_i)}$ (projectionV)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash (\ell_1 = v_1, \ldots, \ell_n = v_n).\ell_i : \tau$ — assumption
3. $\Lambda, \bullet \vdash (\ell_1 = v_1, \ldots, \ell_n = v_n) : (\ell_1 : \tau_1, \ldots, \ell_n : \tau_n)$ — 2, Inversion Lemma
4. $\tau = \tau_i$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash v_i : \tau$ — 3, 4, Inversion Lemma

Result is from 1, 5

Case $\dfrac{(C, e_i) \longrightarrow (C', e_i')}{(C, in_{\ell_i}\ e_i : \tau') \longrightarrow (C', in_{\ell_i}\ e_i' : \tau')}$ (variantE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash (in_{\ell_i}\ e_i : \tau') : \tau$ — assumption
3. $(C, e_i) \longrightarrow (C', e_i')$ — assumption
4. $i \in \{1, \ldots, n\}$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash e_i : \tau_i$ — 2, Inversion Lemma
6. $\tau = \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n$ — 2, Inversion Lemma
7. $\tau' = \tau$ — 2, Inversion Lemma
8. $\exists \Lambda' : (\Lambda' \vdash C'\ ok \ \wedge \ \Lambda', \bullet \vdash e_i' : \tau_i \ \wedge \ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
9. $\Lambda', \bullet \vdash (in_{\ell_i}\ e_i' : \tau') : \ \tau$ — 5-8, Rule variant

Result is from 8, 9

Case $\dfrac{(C, e_c) \longrightarrow (C', e_c')}{(C, (case\ e_c\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n)) \longrightarrow (C', (case\ e_c'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n))}$ (caseE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash (case\ e_c\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau$ — assumption
3. $(C, e_c) \longrightarrow (C', e_c')$ — assumption
4. $\Lambda, \bullet \vdash e_c \ : \ (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — 2, Inversion Lemma
5. $\Lambda, \{x_1 : \tau_1\} \vdash e_1 : \ \tau, \ldots, \Lambda, \{x_1 : \tau_1\} \vdash e_n : \tau$ — 2, Inversion Lemma
6. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_c' : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n) \wedge \ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
7. $\Lambda', \{x_1 : \tau_1\} \vdash e_1 : \ \tau, \ldots, \Lambda', \{x_1 : \tau_1\} \vdash \ell_n : \tau$ — 5, 6, Λ-weakening Lemma
8. $(\Lambda', \bullet \vdash case\ e_c'\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau$ — 6, 7, Rule Case

Result is from 6, 8

Case $\dfrac{i \in \{1, \ldots, n\}}{(C, (case\ (in_{\ell_i} v_i : \tau')\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n)) \longrightarrow (C, [v_i/x_i]e_i)}$ (caseV)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash (case\ (in_{\ell_i} v_i : \tau')\ of\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n) : \tau$ — assumption
3. $\Lambda, \bullet \vdash (in_{\ell_i} v_i : \tau') : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ — 2, Inversion Lemma
4. $\Lambda, \{x_i : \tau_i\} \vdash e_i : \tau$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash v_i : \tau_i$ — 3, Inversion Lemma
6. $\Lambda, \bullet \vdash [v_i/x_i]e_i : \tau$ — 4, 5, Lemma substitution

Result is from 1, 6

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, makeTypedVal(\tau_1, e_1)) \longrightarrow (C', makeTypedVal(\tau_1, e_1'))}$ (makeTypedValE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash makeTypedVal(\tau_1, e_1) : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : \tau_1$ — 2, Inversion Lemma
5. $\tau = TypedVal$ — 2, Inversion Lemma
6. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau_1 \wedge\ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
7. $\Lambda', \bullet \vdash makeTypedVal(\tau_1, e_1') : \tau$ — 5, 6, Rule makeTypedVal

Result is from 6, 7

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, tryCast(\tau_1, e_1)) \longrightarrow (C', tryCast(\tau_1, e_1'))}$ (tryCastE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash tryCast(\tau_1, e_1) : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : TypedVal$ — 2, Inversion Lemma
5. $\tau = \tau_1\ Option$ — 2, Inversion Lemma
6. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : TypedVal \wedge\ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
7. $\Lambda', \bullet \vdash tryCast(\tau_1, e_1') : \tau$ — 5, 6, Rule tryCast

Result is from 6, 7

Case $\dfrac{}{(C, tryCast(\tau', makeTypedVal(\tau', v))) \longrightarrow (C, in_{some}(v) : \tau'\ Option)}$ (tryCastVOk)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash tryCast(\tau', makeTypedVal(\tau', v)) : \tau$ — assumption
3. $\Lambda, \bullet \vdash makeTypedVal(\tau', v) : TypedVal$ — 2, Inversion Lemma
4. $\tau = \tau'\ Option$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash v : \tau'$ — 3, Inversion Lemma
6. $\Lambda, \bullet \vdash (in_{some}(v) : \tau'\ Option) : \tau$ — 4, 5, Def of types, Rule variant

Result is from 1, 6

Case $\dfrac{\tau_1 \neq \tau_2}{(C, tryCast(\tau_1, makeTypedVal(\tau_2, v))) \longrightarrow (C, in_{none}(unit) : \tau_1\ Option)}$ (tryCastVBad)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash tryCast(\tau_1, makeTypedVal(\tau_2, v)) : \tau$ — assumption
3. $\tau = \tau_1\ Option$ — 2, Inversion Lemma
4. $\Lambda, \bullet \vdash unit : Unit$ — Rule unitVal
5. $\Lambda, \bullet \vdash (in_{none}(unit) : \tau_1\ Option) : \tau$ — 3, 4, Def of types, Rule variant

Result is from 1, 5

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, \{\ e_1\ \}_{s(v)}) \longrightarrow (C', \{\ e_1'\ \}_{s(v)})}$ (endLabelE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash \{\ e_1\ \}_{s(v)} : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : \tau$ — 2, Inversion Lemma
5. $\exists \Lambda' : (\Lambda' \vdash C$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : \tau \wedge \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
6. $\Lambda', \bullet \vdash \{\ e_1'\ \}_{s(v)} : \tau$ — 5, Rule endLabel

Result is from 5, 6

Case $\dfrac{s \neq \text{“}monitor\text{”}}{(C, \{\ v_1\ \}_{s(v_2)}) \xrightarrow{end_{s(v_2):v_1}} (C, v_1)}$ (endLabelValue)

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 $\dfrac{s = \text{“}monitor\text{”}}{((M, R, inOb, rt, out, \tau_{out}), \{\ v_1\ \}_{s(v_2)}) \xrightarrow{end_{s(v_2):v_1}} ((M, R, false, rt, out, \tau_{out}), v_1)}$ (endLabelValueMonitor)

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. $\Lambda, \bullet \vdash rt : Res_{List}$ — 1, C-Inversion Lemma
7. $\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$ — 1, C-Inversion Lemma
8. $\Lambda, \bullet \vdash false : Bool$ — Rule boolVal
9. $\Lambda \vdash (M, R, false, rt, out, \tau_{out})$ **ok** — 4-8, Rule C-ok

Result is from 3, 9

Case $\dfrac{}{((\ldots, rt, out, \tau_{out}), getRT()) \longrightarrow ((\ldots, rt, out, \tau_{out}), rt)}$ (getRTVal)

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

Result is from 1, 4

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, makeCFG(e_1)) \longrightarrow (C', makeCFG(e_1'))}$ (makeCFGE)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash makeCFG(e_1) : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\tau = CFG$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash e_1 : Obligation$ — 2, Inversion Lemma
6. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\ \wedge \Lambda', \bullet \vdash e_1' : Obligation \wedge\ \Lambda \subseteq \Lambda')$ — 1, 3, 5, IH
7. $\Lambda', \bullet \vdash makeCFG(e_1') : \tau$ — 4, 6, Rule makeCFG

Result is from 6, 7

Case $\dfrac{g = makeCFG_\alpha(v)}{(C, makeCFG(v)) \xrightarrow{begin_{makeCFG(v)}} (C, \{g\}_{makeCFG(v)})}$ (makeCFGValue)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash makeCFG(v) : \tau$	assumption
3.	$g = makeCFG_\alpha(v)$	assumption
4.	$\Lambda, \bullet \vdash g : CFG$	3, Assumption 2
5.	$\tau = CFG$	2, Inversion Lemma
6.	$\Lambda, \bullet \vdash \{g\}_{makeCFG(v)} : \tau$	4, 5, Rule endLabel

Result is from 1, 6

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, setOutput(e_1)) \longrightarrow (C', setOutput(e_1'))}$ (setOutputE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash setOutput(e_1) : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : Event$	2, Inversion Lemma
5.	$\tau = Bool$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : Event \wedge \ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash setOutput(e_1') : \tau$	5, 6, Rule setOutput

Result is from 6, 7

Case $\dfrac{out = in_{some}(e) : \tau\ Event\ Option}{((\ldots, out, \tau_{out}), setOutput(v)) \longrightarrow ((\ldots, out, \tau_{out}), false)}$ (setOutputSet)

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

Result is from 1, 4

Case $\dfrac{}{\begin{array}{l}((M, R, inOb, rt, in_{none}(unit) : \tau_{out}\ Event\ Option, \tau_{out}), setOutput(in_{act}(v) : Event)) \longrightarrow \\ ((M, R, inOb, rt, in_{some}(in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option, \tau_{out}), true)\end{array}}$ (setOutputNotSetAct)

1.	$\Lambda \vdash (M, R, inOb, rt, in_{none}(unit) : \tau_{out}\ Event\ Option, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash setOutput(in_{act}(v) : Event) : \tau$	assumption
3.	$\Lambda, \bullet \vdash (in_{act}(v) : Event) : Event$	2, Inversion Lemma
4.	$\tau = Bool$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash v : Act$	3, Inversion Lemma
6.	$M : \Lambda$	1, C-Inversion Lemma
7.	$\Lambda \vdash R$ **ok**	1, C-Inversion Lemma
8.	$\Lambda, \bullet \vdash inOb : Bool$	1, C-Inversion Lemma
9.	$\Lambda, \bullet \vdash rt : Res_{List}$	1, C-Inversion Lemma
10.	$\Lambda, \bullet \vdash (in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event$	5, Rule variant, Def of types
11.	$\Lambda, \bullet \vdash (in_{some}(in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option) : \tau_{out}\ Event\ Option$	10, Rule variant, Def of types
12.	Let $out = in_{some}(in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option$	assumption
13.	$\Lambda \vdash (M, R, inOb, rt, out, \tau_{out})$ **ok**	6-9, 11, 12, Rule C-ok
14.	$\Lambda, \bullet \vdash true : \tau$	4, Rule boolVal

Result is from 13, 14

Case $\dfrac{}{((M, R, inOb, rt, in_{none}(unit) : \tau_{out}\ Event\ Option, \tau_{out}), setOutput(in_{res}(res(v_1, makeTypedVal(\tau_{out}, v_2))) : Event)) \longrightarrow ((M, R, inOb, rt, in_{some}(in_{res}(res(v_1, v_2)) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option, \tau_{out}), true)}$ (setOutputNotSetResGood)

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

Result is from 17, 18

Case $\dfrac{\tau' \neq \tau_{out} \quad out = in_{none}(unit) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), setOutput(in_{res}(res(v_1, makeTypedVal(\tau', v_2))) : Event)) \longrightarrow ((\ldots, out, \tau_{out}), false)}$ (setOutputNotSetResBad)

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

Result is from 1, 4

Case $\dfrac{out = in_{none}(unit) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), outputNotSet()) \longrightarrow ((\ldots, out, \tau_{out}), true)}$ (outputNotSetTrue)

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

Result is from 1, 4

Case $\dfrac{out = in_{some}(e_1) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}, outputNotSet()) \longrightarrow ((\ldots, out, \tau_{out}), false)}$ (outputNotSetFalse)

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

Result is from 1, 4

Case $\dfrac{out = in_{some}(in_{act}(v) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), getOutput()) \longrightarrow ((\ldots, out, \tau_{out}), in_{some}(in_{act}(v) : Event) : Event\ Option)}$ (getOutputSomeAct)

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

Result is from 1, 9

Case $\dfrac{out = in_{some}(in_{res}(res(v_1, v_2)) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), getOutput()) \longrightarrow ((\ldots, out, \tau_{out}), in_{some}(in_{res}(res(v_1, makeTypedVal(\tau_{out}, v_2))) : Event) : Event\ Option)}$ (getOutputSomeRes)

1.	$\Lambda \vdash (\ldots, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash getOutput() : \tau$	assumption
3.	$out = in_{some}(in_{res}(res(v_1, v_2)) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option$	assumption
4.	$\tau = Event\ Option$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash (in_{some}(in_{res}(res(v_1, v_2)) : \tau_{out}\ Event) : \tau_{out}\ Event\ Option) : \tau_{out}\ Event\ Option$	1, 3, C-Inversion Lemma
6.	$\Lambda, \bullet \vdash (in_{res}(res(v_1, v_2))) : \tau_{out}\ Event$	5, Inversion Lemma
7.	$\Lambda, \bullet \vdash res(v_1, v_2) : \tau_{out}\ Res$	6, Inversion Lemma
8.	$\Lambda, \bullet \vdash v_1 : Act$	7, Inversion Lemma
9.	$\Lambda, \bullet \vdash v_2 : \tau_{out}$	7, Inversion Lemma
10.	$\Lambda, \bullet \vdash makeTypedVal(\tau_{out}, v_2) : TypedVal$	9, Rule makeTypedVal
11.	$\Lambda, \bullet \vdash res(v_1, makeTypedVal(\tau_{out}, v_2)) : Res$	8, 10, Rule variant, Def of types
12.	$\Lambda, \bullet \vdash (in_{res}(res(v_1, makeTypedVal(\tau_{out}, v_2))) : Event) : Event$	11, Rule variant, Def of types
13.	$\Lambda, \bullet \vdash (in_{some}(in_{res}(res(v_1, makeTypedVal(\tau_{out}, v_2))) : Event) : Event\ Option) : \tau$	4, 12, Rule variant, Def of types

Result is from 1, 13

Case $\dfrac{out = in_{none}(unit) : \tau_{out}\ Event\ Option}{((\ldots, out, \tau_{out}), getOutput()) \longrightarrow ((\ldots, out, \tau_{out}), in_{none}(unit) : Event\ Option)}$ (getOutputNone)

1.	$\Lambda \vdash (\ldots, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash getOutput() : \tau$	assumption
3.	$\tau = Event\ Option$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash unit : Unit$	Rule unitVal
5.	$\Lambda, \bullet \vdash (in_{none}(unit) : Event\ Option) : \tau$	3, 4, Rule variant, Def of types

Result is from 1, 5

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, monitor(\tau_1, e_1)) \longrightarrow (C', monitor(\tau_1, e_1'))}$ (monitorE)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash monitor(\tau_1, e_1) : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 :$ $(evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$	2, Inversion Lemma
5.	$\tau = \tau_1\ Event$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge$ $\Lambda', \bullet \vdash e_1' : (evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$ $\wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash monitor(\tau_1, e_1') : \tau$	5, 6, Rule monitor

Result is from 6, 7

Case $\dfrac{}{((M, R, inOb, rt, out, \tau_{out}), monitor(\tau_1, v)) \xrightarrow{begin_{monitor(v)}} ((M, R, true, rt, out, \tau_{out}), \{e_{monitor}(\tau_1, v)\}_{monitor(v)})}$ (monitorV)

1.	$\Lambda \vdash (M, R, inOb, rt, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash monitor(\tau_1, v) : \tau$	assumption
3.	$\Lambda, \bullet \vdash v :$ $(evt : \tau_1\ Event \times pols : Pol_{List} \times os : OS \times vc : VC)$	2, Inversion Lemma
4.	$\tau = \tau_1\ Event$	2, Inversion Lemma
5.	$M : \Lambda$	1, C-Inversion Lemma
6.	$\Lambda \vdash R$ **ok**	1, C-Inversion Lemma
7.	$\Lambda, \bullet \vdash rt : Res_{List}$	1, C-Inversion Lemma
8.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, C-Inversion Lemma
9.	$\Lambda, \bullet \vdash true : Bool$	Rule boolVal
10.	$\Lambda \vdash (M, R, true, rt, out, \tau_{out})$ **ok**	5-9, Rule C-ok
11.	$\Lambda, \bullet \vdash e_{monitor}(\tau_1, v) : \tau_1\ Event$	3, Monitor Type Lemma
12.	$\Lambda, \bullet \vdash \{e_{monitor}(\tau_1, v)\}_{monitor(v)} : \tau_1\ Event$	11, Rule endLabel

Result is from 10, 12

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, invoke(e_1, e_2)) \longrightarrow (C', invok(e_1', e_2))}$ (invokeE1)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash invoke(e_1, e_2) : \tau$	assumption
3.	$(C, e_1) \longrightarrow (C', e_1')$	assumption
4.	$\Lambda, \bullet \vdash e_1 : String$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash e_2 : TypedVal$	2, Inversion Lemma
6.	$\tau = TypedVal\ Option$	2, Inversion Lemma
7.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_1' : String \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
8.	$\Lambda', \bullet \vdash e_2 : TypedVal$	5, 7, Λ-weakening Lemma
9.	$\Lambda', \bullet \vdash invoke(e_1', e_2) : \tau$	6-8, Rule Inovke

Result is from 7, 9

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, invoke(s_1, e_2)) \longrightarrow (C', invoke(s_1, e_2'))}$ (invokeE2)

1.	$\Lambda \vdash C$ **ok**	assumption
2.	$\Lambda, \bullet \vdash invoke(s_1, e_2) : \tau$	assumption
3.	$(C, e_2) \longrightarrow (C', e_2')$	assumption
4.	$\Lambda, \bullet \vdash e_2 : TypedVal$	2, Inversion Lemma
5.	$\tau = TypedVal\ Option$	2, Inversion Lemma
6.	$\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge\ \Lambda', \bullet \vdash e_2' : TypedVal \wedge\ \Lambda \subseteq \Lambda')$	1, 3, 4, IH
7.	$\Lambda', \bullet \vdash s_1 : String$	Rule Stringval
8.	$\Lambda', \bullet \vdash invoke(s_1, e_2') : \tau$	5-7, Rule invoke

Result is from 6, 8

Case $\dfrac{\forall (s', f) \in F.s_1 \neq s'}{((M, (F, \dots), \dots), invoke(s_1, v_2)) \longrightarrow ((M, (F, \dots), \dots), in_{none}(unit) : TypedVal\ Option)}$ (invokeValNotExists)

1.	$\Lambda \vdash (M, (F, \dots), \dots)$ **ok**	assumption
2.	$\Lambda, \bullet \vdash invoke(s_1, v_2) : \tau$	assumption
3.	$\tau = TypedVal\ Option$	2, Inversion Lemma
4.	$\Lambda, \bullet \vdash unit : Unit$	Rule unitVal
5.	$\Lambda, \bullet \vdash (in_{none}(unit) : TypedVal\ Option) : \tau$	3, 4, Def of types, Rule variant

Result is from 1, 5

Case $\dfrac{(s_1, fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F \quad v_2 = makeTypedVal(\tau_1, v_2')}{((M, (F, \dots), \dots), invoke(s_1, v_2)) \longrightarrow ((M, (F, \dots), \dots), in_{some}(makeTypedVal(\tau_2, call(fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1, v_2'))) : TypedVal\ Option)}$ (invokeValueExistsOk)

1.	$\Lambda \vdash (M, (F, \dots), \dots)$ **ok**	assumption
2.	$\Lambda, \bullet \vdash invoke(s_1, v_2) : \tau$	assumption
3.	$v_2 = makeTypedVal(\tau_1, v_2')$	assumption
4.	$\Lambda, \bullet \vdash v_2 : TypedVal$	2, Inversion Lemma
5.	$\tau = TypedVal\ Option$	2, Inversion Lemma
6.	$\Lambda, \bullet \vdash v_2' : \tau_1$	3, 4, Inversion Lemma
7.	$\Lambda \vdash (F, \dots)$ **ok**	1, C-Inversion Lemma
8.	$\Lambda \vdash F$ **ok**	7, R-Inversion Lemma
9.	$(s_1, fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F$	assumption
10.	8 is only derivable by Rule F-Ok	Inspection of $\Lambda \vdash F$ **ok** rules
11.	$\Lambda, \bullet \vdash (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1) : \tau_1' \rightarrow \tau_2'$	8, 10, Inversion of Rule F-ok
12.	11 is only derivable by Rule fun	Inspection of typing rules
13.	$\tau_1' = \tau_1$ and $\tau_2' = \tau_2$	11, 12, Inversion of Rule fun
14.	$\Lambda, \bullet \vdash call(fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1, v_2') : \tau_2$	6, 11, 13, Rule call
15.	Let $x = call(fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1, v_2')$	assumption
16.	$\Lambda, \bullet \vdash makeTypedVal(\tau_2, x) : TypedVal$	14, 15, Rule makeTypedVal
17.	$\Lambda, \bullet \vdash (in_{some}(makeTypedVal(\tau_2, x)) : TypedVal\ Option) : \tau$	5, 16, Rule variant, Def of types

Result is from 1, 17

Case $\dfrac{(s_1, fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F \quad v_2 = makeTypedVal(\tau_3, v_2') \quad \tau_1 \neq \tau_3}{((M, (F, \dots), \dots), invoke(s_1, v_2)) \longrightarrow ((M, (F, \dots), \dots), in_{none}(unit) : TypedVal\ Option)}$ (invokeValueExistsBad)

1. $\Lambda \vdash (M, (F, \dots), \dots)$ **ok** — assumption
2. $\Lambda, \bullet \vdash invoke(s_1, v_2) : \tau$ — assumption
3. $\tau = TypedVal\ Option$ — 2, Inversion Lemma
4. $\Lambda, \bullet \vdash unit : Unit$ — Rule unitVal
5. $\Lambda, \bullet \vdash (in_{none}(unit) : TypedVal\ Option) : \tau$ — 3, 4, Def of types, Rule variant

Result is from 1, 5

Case $\dfrac{(C, e_1) \longrightarrow (C', e_1')}{(C, call(e_1, e_2)) \longrightarrow (C', call(e_1' e_2))}$ (callE1)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash call(e_1, e_2) : \tau$ — assumption
3. $(C, e_1) \longrightarrow (C', e_1')$ — assumption
4. $\Lambda, \bullet \vdash e_1 : \tau_1 \to \tau_2$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash e_2 : \tau_1$ — 2, Inversion Lemma
6. $\tau = \tau_2$ — 2, Inversion Lemma
7. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_1' : \tau_1 \to \tau_2 \wedge\ \Lambda \subseteq \Lambda')$ — 1, 3, 4, IH
8. $\Lambda', \bullet \vdash e_2 : \tau_1$ — 5, 7, Λ-weakening Lemma
9. $\Lambda', \bullet \vdash call(e_1', e_2) : \tau$ — 6-8 Rule call

Result is from 7, 9

Case $\dfrac{(C, e_2) \longrightarrow (C', e_2')}{(C, call(f, e_2)) \longrightarrow (C', call(f, e_2'))}$ (callE2)

1. $\Lambda \vdash C$ **ok** — assumption
2. $\Lambda, \bullet \vdash call(f, e_2) : \tau$ — assumption
3. $(C, e_2) \longrightarrow (C', e_2')$ — assumption
4. $\Lambda, \bullet \vdash f : \tau_1 \to \tau_2$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash e_2 : \tau_1$ — 2, Inversion Lemma
6. $\tau = \tau_2$ — 2, Inversion Lemma
7. $\exists \Lambda' : (\Lambda' \vdash C'$ **ok** $\wedge \Lambda', \bullet \vdash e_2' : \tau_1 \wedge\ \Lambda \subseteq \Lambda')$ — 1, 3, 5, IH
8. $\Lambda', \bullet \vdash f : \tau_1 \to \tau_2$ — 4, 7, Λ-weakening Lemma
9. $\Lambda', \bullet \vdash call(f, e_2') : \tau$ — 6-8, Rule Call

Result is from 7, 9

Case $\dfrac{f \notin range(F) \quad \forall pol \in pols\ (f \neq pol.onTrigger \wedge f \neq pol.onObligation) \quad f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1)}{((M, (F, pols, \dots), \dots), call(f, v)) \xrightarrow{begin_{f(v)}} \left((M, (F, pols, \dots), \dots), \{[v/x_2, f/x_1]e_1\}_{f(v)}\right)}$ (callNonMonitoredFunction)

1. $\Lambda \vdash (M, (F, \dots), \dots)$ **ok** — assumption
2. $\Lambda, \bullet \vdash call(f, v) : \tau$ — assumption
3. $f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1)$ — assumption
4. $\Lambda, \bullet \vdash f : \tau_1' \to \tau_2'$ — 2, Inversion Lemma
5. $\Lambda, \bullet \vdash v : \tau_1'$ — 2, Inversion Lemma
6. $\tau = \tau_2'$ — 2, Inversion Lemma
7. $\Lambda, \{x_1 : \tau_1' \to \tau_2', x_2 : \tau_1'\} \vdash e_1 : \tau_2'$ — 3, 4, Inversion Lemma
8. $\Lambda, \bullet \vdash [v/x_2, f/x_1]e_1 : \tau$ — 3-7, Substitution Lemma
9. $\Lambda, \bullet \vdash \{[v/x_2, f/x_1]e_1\}_{f(v)} : \tau$ — 8, Rule endLabel

Result is from 1, 9

$$\frac{(name = s, onTrigger = f_1, onObligation = f_2, vote = f_3) \in pols \quad f_1 = (fun\ x_1(x_2 : Event) : Unit = e_1)}{((M, (F, pols, os, vc), inOb, rt, out, \tau_{out}), call(f_1, v)) \xrightarrow{begin_{f_1(v)}} \left((M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out}), \{[f_1/x_1, v/x_2]e_1\}_{f_1(v)}\right)}\ \text{(callOnTrigger)}$$

Case

1.	$\Lambda \vdash (M, (F, pols, os, vc), inOb, rt, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash call(f_1, v) : \tau$	assumption
3.	$f_1 = (fun\ x_1(x_2 : Event) : Unit = e_1)$	assumption
4.	$\Lambda, \bullet \vdash f_1 : \tau_1 \rightarrow \tau_2$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash v : \tau_1$	2, Inversion Lemma
6.	$\tau = \tau_2$	2, Inversion Lemma
7.	$\Lambda, \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e_1 : \tau_2$	3, 4, Inversion Lemma
8.	$\Lambda, \bullet \vdash [f_1/x_1, v/x_2]e_1 : \tau$	3-7, Substitution Lemma
9.	$M : \Lambda$	1, C-Inversion Lemma
10.	$\Lambda \vdash (F, pols, os, vc)$ **ok**	1, C-Inversion Lemma
11.	$\Lambda, \bullet \vdash inOb : Bool$	1, C-Inversion Lemma
12.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, C-Inversion Lemma
13.	$\Lambda, \bullet \vdash ([] : Res_{List}) : Res_{List}$	Rule listEmptyVal
14.	$\Lambda \vdash (M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out})$ **ok**	9-13, Rule C-ok
15.	$\Lambda, \bullet \vdash \{[f_1/x_1, v/x_2]e_1\}_{f_1(v)} : \tau$	8, Rule endLabel

Result is from 14, 15

$$\frac{(name = s, onTrigger = f_1, onObligation = f_2, vote = f_3) \in pols \quad f_2 = (fun\ x_1(x_2 : Res_{List}) : Unit = e_1)}{((M, (F, pols, os, vc), inOb, rt, out, \tau_{out}), call(f_2, v)) \xrightarrow{begin_{f_2(v)}} \left((M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out}), \{[f_2/x_1, v/x_2]e_1\}_{f_2(v)}\right)}\ \text{(callOnObligation)}$$

Case

1.	$\Lambda \vdash (M, (F, pols, os, vc), inOb, rt, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash call(f_2, v) : \tau$	assumption
3.	$f_2 = (fun\ x_1(x_2 : Event) : Unit = e_1)$	assumption
4.	$\Lambda, \bullet \vdash f_2 : \tau_1 \rightarrow \tau_2$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash v : \tau_1$	2, Inversion Lemma
6.	$\tau = \tau_2$	2, Inversion Lemma
7.	$\Lambda, \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e_1 : \tau_2$	3, 4, Inversion Lemma
8.	$\Lambda, \bullet \vdash [f_2/x_1, v/x_2]e_1 : \tau$	3-7, Substitution Lemma
9.	$M : \Lambda$	1, C-Inversion Lemma
10.	$\Lambda \vdash (F, pols, os, vc)$ **ok**	1, C-Inversion Lemma
11.	$\Lambda, \bullet \vdash inOb : Bool$	1, C-Inversion Lemma
12.	$\Lambda, \bullet \vdash out : \tau_{out}\ Event\ Option$	1, C-Inversion Lemma
13.	$\Lambda, \bullet \vdash ([] : Res_{List}) : Res_{List}$	Rule listEmptyVal
14.	$\Lambda \vdash (M, (F, pols, os, vc), inOb, [] : Res_{List}, out, \tau_{out})$ **ok**	9-13, Rule C-ok
15.	$\Lambda, \bullet \vdash \{[f_2/x_1, v/x_2]e_1\}_{f_2(v)} : \tau$	8, Rule endLabel

Result is from 14, 15

Case
$$\frac{(s, fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F \quad f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1)}{((M, (F, pols, os, vc), true, rt, out, \tau_{out}), call(f, v)) \xrightarrow{begin_{x_1(v)},\ begin_{appendRes()}} \left((M, (F, pols, os, vc), true, rt\ @\ res(act(s, makeTypedVal(\tau_1, v)), makeTypedVal(\tau_2, [f/x_1, v/x_2]e_1)) :: [] : Res_{List}, out, \tau_{out}), \{[f/x_1, v/x_2]e_1\}_{x_1(v)}\right)}\ \text{(callFromObligation)}$$

1.	$\Lambda \vdash (M, (F, pols, os, vc), true, rt, out, \tau_{out})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash call(f, v) : \tau$	assumption
3.	$f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1)$	assumption
4.	$\Lambda, \bullet \vdash f : \tau_1' \to \tau_2'$	2, Inversion Lemma
5.	$\Lambda, \bullet \vdash v : \tau_1'$	2, Inversion Lemma
6.	$\tau = tau_2'$	2, Inversion Lemma
7.	4 is only derivable by Rule fun	Inspection of typing rules
8.	$\tau_1 = \tau_1' \wedge \tau_2 = \tau_2'$	3, 4, 7, Inversion of Rule fun
9.	$\Lambda, \{x_1 : \tau_1 \to \tau_2, x_2 : \tau_1\} \vdash e_1 : \tau_2$	3, 4, 6, Inversion Lemma
10.	$\Lambda, \bullet \vdash [f/x_1, v/x_2]e_1 : \tau_2$	4, 5, 9, Substitution Lemma
11.	$\Lambda, \bullet \vdash makeTypedVal(\tau_2, [f/x_1, v/x_2]e_1) : TypedVal$	10, Rule makeTypedVal
12.	$\Lambda, \bullet \vdash makeTypedVal(\tau_1, v) : TypedVal$	5, Rule makeTypedVal
13.	$\Lambda, \bullet \vdash s : String$	Rule stringVal
14.	$\Lambda, \bullet \vdash act(s, makeTypedVal(\tau_1, v)) : Act$	12, 13, Rule product, Def of types
15.	$\Lambda, \bullet \vdash res(act(s, makeTypedVal(\tau_1, v)), makeTypedVal(\tau_2, [f/x_1, v/x_2]e_1)) : Res$	11, 14, Rule product, Def of types
16.	Let $r = res(act(s, makeTypedVal(\tau_1, v)), makeTypedVal(\tau_2, [f/x_1, v/x_2]e_1))$	assumption
17.	$\Lambda, \bullet \vdash ([] : Res_{List}) : Res_{List}$	Rule listEmptyVal
18.	$\Lambda, \bullet \vdash (r :: [] : Res_{List}) : Res_{List}$	15-17, Rule listCons
19.	$\Lambda, \bullet \vdash rt : Res_{List}$	1, C-Inversion Lemma
20.	$\Lambda, \bullet \vdash rt\ @\ (r :: [] : Res_{List}) : Res_{List}$	18, 19, Rule listAppend
21.	$M : \Lambda$	1, C-Inversion Lemma
22.	$\Lambda \vdash (F, pols, os, vc)$ **ok**	1, C-Inversion Lemma
23.	$\Lambda, \bullet \vdash true : Bool$	Rule boolVal
24.	$\Lambda, \bullet out : \tau_{out}\ Event\ Option$	1, C-Inversion Lemma
25.	$\Lambda \vdash (M, (F, pols, os, vc), true, rt\ @\ (r :: [] : Res_{List}), out, \tau_{out})$ **ok**	20-24, Rule C-ok
26.	$\Lambda, \bullet \vdash \{[f/x_1, v/x_2]e_1\}_{x_1(v)} : \tau$	6, 8, 10, Rule endLabel
	Result is from 25, 26	

Case

$$\frac{(s,f) \in F \quad f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1)}{((M,(F,pols,os,vc),false,rt,out,\tau_{old}),call(f,v)) \xrightarrow{begin_{s(v)}} ((M,(F,pols,os,vc),false,rt,in_{none}(unit) : \tau_2\ Event\ Option,\tau_2),e_{procEvt})}\ \text{(callFromApplication)}$$

Where $e_{procEvt} =$

$\{let\ aux = (fun\ aux(event : \tau_2\ Event) : \tau_2\ Res =$
$case\ event\ of$
$act\ a \Rightarrow$
$case\ invoke(a.name, a.arg)\ of$
$some\ r_1 \Rightarrow$
$case\ tryCast(\tau_2, r_1)\ of$
$some\ v_1 \Rightarrow$
$let\ action\_output = in_{res}\ res(a, v_1) : \tau_2\ Event\ in$
$let\ mon\_output = monitor(\tau_2, (evt = action\_output, pols = pols, os = os, vc = vc))\ in$
$call(aux, mon\_output)$
end
end
$| none\ u_1 \Rightarrow call(aux, event)$
$| none\ u_2 \Rightarrow call(aux, event)$
$| res\ r_2 \Rightarrow r_2)$
in
$call(aux, in_{act}\ act(s, v) : \tau_2\ Event).result$
$end\}_{s(v)}$

1.	$\Lambda \vdash (M,(F,pols,os,vc),false,rt,out,\tau_{old})$ **ok**	assumption
2.	$\Lambda, \bullet \vdash call(f,v) : \tau$	assumption
3.	$f = (fun\ x_1(x_2 : \tau_1) : \tau_2 = e_1)$	assumption
4.	$M : \Lambda$	1, C-Inversion Lemma
5.	$\Lambda \vdash (F,pols,os,vc)$ **ok**	1, C-Inversion Lemma
6.	$\Lambda, \bullet \vdash rt : Res_{List}$	1, C-Inversion Lemma
7.	$\Lambda, \bullet \vdash false : Bool$	Rule boolVal
8.	$\Lambda, \bullet \vdash unit : Unit$	Rule unitVal
9.	$\Lambda, \bullet \vdash (in_{none}(unit) : \tau_2\ Event\ Option) : \tau_2\ Event\ Option$	8, Rule variant, Def of types
10.	$\Lambda \vdash (M,(F,pols,os,vc),false,rt,in_{none}(unit) : \tau_2\ Event\ Option,\tau_2)$ **ok**	4-7, 9, Rule C-ok
11.	Let $\Gamma_0 = \{aux : \tau_2\ Event \to \tau_2\ Res, event : \tau_2\ Event, a : Act, r_2 : \tau_2\ Res,$ $r_1 : TypedVal, u_2 : Unit, v_1 : \tau_2, u_1 : Unit, action\_output : \tau_2\ Event, mon\_output : \tau_2\ Event\}$	assumption
12.	$\Lambda, \Gamma_0 \vdash aux : \tau_2\ Event \to \tau_2\ Res$	11, Rule var
13.	$\Lambda, \Gamma_0 \vdash mon\_output : \tau_2\ Event$	11, Rule var
14.	$\Lambda, \Gamma_0 \vdash call(aux, mon\_output) : \tau_2\ Res$	12, 13, Rule call
15.	Let $\Gamma_1 = \{aux : \tau_2\ Event \to \tau_2\ Res, event : \tau_2\ Event, a : Act, r_2 : \tau_2\ Res,$ $r_1 : TypedVal, u_2 : Unit, v_1 : \tau_2, u_1 : Unit, action\_output : \tau_2\ Event\}$	assumption
16.	$\Lambda, \bullet \vdash pols : Pol_{List}$	5, R-Inversion Lemma
17.	$\Lambda, \bullet \vdash os : OS$	5, R-Inversion Lemma
18.	$\Lambda, \bullet \vdash vc : VC$	5, R-Inversion Lemma
19.	$\Lambda, \Gamma_1 \vdash pols : Pol_{List}$	15, 16, Weakening Lemma
20.	$\Lambda, \Gamma_1 \vdash os : OS$	15, 17, Weakening Lemma
21.	$\Lambda, \Gamma_1 \vdash vc : VC$	15, 18, Weakening Lemma
22.	Let $c = ((evt = action\_output, pols = pols, os = os, vc = vc)$	assumption
23.	$\Lambda, \Gamma_1 \vdash c : (evt : \tau_2\ Event \times pol : Pol_{List} \times os : OS \times vc : VC)$	19-22, Rule variant
24.	$\Lambda, \Gamma_1 \vdash monitor(\tau_2, c) : \tau_2\ Event$	23, Rule monitor
25.	$\Lambda, \Gamma_1 \vdash let\ mon\_output = monitor(\tau_2, c)\ in\ call(aux, mon\_output)\ end : \tau_2\ Res$	14, 24, Rule let
26.	Let $\Gamma_2 = \{aux : \tau_2\ Event \to \tau_2\ Res, event : \tau_2\ Event, a : Act, r_2 : \tau_2\ Res,$ $r_1 : TypedVal, u_2 : Unit, v_1 : \tau_2, u_1 : Unit\}$	assumption
27.	$\Lambda, \Gamma_2 \vdash a : Act$	26, Rule var
28.	$\Lambda, \Gamma_2 \vdash v_1 : \tau_2$	26, Rule var
29.	$\Lambda, \Gamma_2 \vdash res(a, v_1) : \tau_2\ Res$	27, 28, Rule variant, Def of types
30.	$\Lambda, \Gamma_2 \vdash (in_{res}\ res(a, v_1) : \tau_2\ Event) : \tau_2\ Event$	29, Rule variant, Def of types
31.	Let $e_2 = (let\ mon\_output = monitor(\tau_2, c)\ in\ call(aux, mon\_output)\ end)$	assumption
32.	$\Lambda, \Gamma_2 \vdash let\ action\_output = in_{res}\ res(a, v_1) : \tau_2\ Event\ in\ e_2\ end : \tau_2\ Res$	25, 30, 31, Rule let

33.	$\Lambda, \Gamma_2 \vdash event : \tau_2\ Event$	26, Rule var
34.	$\Lambda, \Gamma_2 \vdash aux : \tau_2\ Event \to \tau_2\ Res$	26, Rule var
35.	$\Lambda, \Gamma_2 \vdash call(aux, event) : \tau_2\ Res$	33, 34, Rule call
36.	Let $\Gamma_3 = \{aux : \tau_2\ Event \to \tau_2\ Res, event : \tau_2\ Event, a : Act, r_2 : \tau_2\ Res, r_1 : TypedVal, u_2 : Unit\}$	assumption
37.	$\Lambda, \Gamma_3 \vdash r_1 : TypedVal$	35, Rule var
38.	$\Lambda, \Gamma_3 \vdash tryCast(\tau_2, r_1) : TypedVal\ Option$	36, Rule tryCast
39.	Let $e_3 = (let\ action\_output = in_{res}\ res(a, v_1 : \tau_2\ Event\ in\ e_2\ end)$	assumption
40.	$\Lambda, \Gamma_3 \vdash case\ tryCast(\tau_2, r_1)\ of\ some\ r_1 \Rightarrow e_3 \mid none\ u_1 \Rightarrow call(aux, event) : \tau_2\ Res$	32, 34, 37, 38, Rule case
41.	Let $e_4 = (case\ tryCast(\tau_2, r_1)\ of\ some\ r_1 \Rightarrow e_3 \mid none\ u_1 \Rightarrow call(aux, event))$	assumption
42.	$\Lambda, \Gamma_3 \vdash aux : \tau_2\ Event \to \tau_2\ Res$	36, Rule var
43.	$\Lambda, \Gamma_3 \vdash event : \tau_2\ Event$	36, Rule var
44.	$\Lambda, \Gamma_3 \vdash call(aux, event) : \tau_2\ Res$	42, 43, Rule call
45.	Let $\Gamma_4 = \{aux : \tau_2\ Event \to \tau_2\ Res, event : \tau_2\ Event, a : Act, r_2 : \tau_2\ Res\}$	assumption
46.	$\Lambda, \Gamma_4 \vdash a : Act$	45, Rule var
47.	$\Lambda, \Gamma_4 \vdash a.name : String$	46, Rule projection
48.	$\Lambda, \Gamma_4 \vdash a.arg : TypedVal$	46, Rule projection
49.	$\Lambda, \Gamma_4 \vdash invoke(a.name, a.arg) : TypedVal\ Option$	47, 48, Rule invoke
50.	$\Lambda, \Gamma_4 \vdash (case\ invoke(a.name, a.arg)\ of\ some\ r_1 \Rightarrow e_4 \mid none\ u_2 \Rightarrow call(aux, event)) : \tau_2\ Res$	40, 41, 44, 49, Rule case
51.	Let $e_5 = (case\ invoke(a.name, a.arg)\ of\ some\ r_1 \Rightarrow e_4 \mid none\ u_2 \Rightarrow call(aux, event))$	assumption
52.	$\Lambda, \Gamma_4 \vdash r_2 : \tau_2\ Res$	45, Rule var
53.	Let $\Gamma_5 = \{aux : \tau_2\ Event \to \tau_2\ Res, event : \tau_2\ Event\}$	assumption
54.	$\Lambda, \Gamma_5 \vdash event : \tau_2\ Event$	53, Rule var
55.	$\Lambda, \Gamma_5 \vdash (case\ event\ of\ act\ a \Rightarrow e_5 \mid res\ r_2 \Rightarrow r_2) : \tau_2\ Res$	50, 51, 52, 54, Rule case
56.	Let $e_6 = (case\ event\ of\ act\ a \Rightarrow e_5 \mid res\ r_2 \Rightarrow r_2)$	assumption
57.	Let $\Gamma_6 = \{aux : \tau_2\ Event \to \tau_2\ Res\}$	assumption
58.	$\Lambda, \Gamma_6 \vdash aux : \tau_2\ Event \to \tau_2\ Res$	57, Rule var
59.	$\Lambda, \Gamma_6 \vdash s : String$	Rule stringVal
60.	$\Lambda, \bullet \vdash v : TypedVal$	2, Inversion Lemma
61.	$\Lambda, \Gamma_6 \vdash v : TypedVal$	60, Weakening Lemma
62.	$\Lambda, \Gamma_6 \vdash act(s, v) : Act$	59, 61, Rule record, Def of types
63.	$\Lambda, \Gamma_6 \vdash (in_{act}\ act(s, v) : \tau_2\ Event) : \tau_2\ Event$	62, Rule variant, Def of types
64.	$\Lambda, \Gamma_6 \vdash call(aux, in_{act}\ act(s, v) : \tau_2\ Event) : \tau_2\ Res$	58, 63, Rule call
65.	$\Lambda, \Gamma_6 \vdash call(aux, in_{act}\ act(s, v) : \tau_2\ Event).result : \tau_2$	64, Rule productSelect, Def of types
66.	$\Lambda, \bullet \vdash (fun\ aux(event : \tau_2\ Event) : \tau_2\ Res = e_6) : \tau_2\ Event \to \tau_2\ Res$	55, 56, Rule fun
67.	$\Lambda, \bullet \vdash (let\ aux = (fun\ aux(event : \tau_2\ Event) : \tau_2\ Res = e_6)$ $in\ call(aux, in_{act}\ act(s, v) : \tau_2\ Event).result\ end) : \tau_2$	65, 66, Rule let
68.	$\Lambda, \bullet \vdash \{let\ aux = (fun\ aux(event : \tau_2\ Event) : \tau_2\ Res = e_6)$ $in\ call(aux, in_{act}\ act(s, v) : \tau_2\ Event).result\ end\}_{s(v)} : \tau_2$	67, Rule endLabel
69.	$\tau = \tau_2$	2, Inversion Lemma
70.	$e_{procEvt} : \tau$	68, 69
	Result is from 10, 70	

APPENDIX F DEFINITION OF ∞-LANGUAGES

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

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

(1) If a language L is regular, then L is an ∞-language
(2) If an ω-language L is ω-regular, then L is an ∞-language
(3) If L_1, L_2 are ∞-languages, then L_1L_2 is an ∞-language, and $L_1L_2 = \{xy \text{ if } x \text{ is finite, else } x \mid (x, y) \in L_1 \times L_2\}$

As Rules 1 and 2 show, the set of ∞-languages is the union of regular languages and ω-regular languages, while Rule 3 defines concatenation such that the set of ∞-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 ω-regular languages).

With this additional rule, a left-hand concatenation operand L^∞ 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

cedes control. Thus, with this concatenation rule, a possibly divergent program trace can be expressed concisely. For instance, consider the ∞-expression $ax^{\infty}by^{\infty}c$. As shown by this expansion below, the interpretation should be "a, followed by a finite or infinite repetition of x; if finite, then b, followed by a finite or infinite repetition of y; if finite, then c".

$$
\begin{aligned}
& ax^{\infty}b\ y^{\infty}c \\
&= a(x^{\omega}|x^{*})b(y^{\omega}|y^{*})c && \text{[definition of } L^{\infty}\text{]} \\
&= (ax^{\omega}|ax^{*})b(y^{\omega}|y^{*})c && \text{[distribution]} \\
&= (ax^{\omega}b|ax^{*}b)(y^{\omega}|y^{*})c && \text{[distribution]} \\
&= (ax^{\omega}|ax^{*}b)(y^{\omega}|y^{*})c && \text{[concatenation]} \\
&= (ax^{\omega}y^{\omega}|ax^{\omega}y^{*}|ax^{*}by^{\omega}|ax^{*}by^{*})c && \text{[distribution]} \\
&= (ax^{\omega}|ax^{\omega}|ax^{*}by^{\omega}|ax^{*}by^{*})c && \text{[concatenation]} \\
&= (ax^{\omega}|ax^{*}by^{\omega}|ax^{*}by^{*})c && \text{[idempotent]} \\
&= ax^{\omega}c|ax^{*}by^{\omega}c|ax^{*}by^{*}c && \text{[distribution]} \\
&= ax^{\omega}|ax^{*}by^{\omega}|ax^{*}by^{*}c && \text{[concatenation]}
\end{aligned}
$$

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 $e_1 \rightarrow^{*} v1$ 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.	e2 cannot be reduced until $e_1 \rightarrow^{*} v_1$	Rule sequenceE2
2.	after $e1 \rightarrow^{*} v1$ producing trace t_1, $t = t_1$	1, sequenceE1
3.	values are always added to end of trace	Definition of $\xrightarrow{label:\ v} \beta$
4.	after $e_2 \rightarrow^{*} v_2$ producing trace t_2, $t = [t_1, t_2]$	2, 3, Rule sequence E2

□

Lemma 13 (Case-statement Traces). Given a well-typed expression e where $e = (case\ e_x\ of\ \ell_1\ x_1\ \Rightarrow\ e_1 \mid \cdots \mid \ell_n\ x_n\ \Rightarrow\ e_n)$: if $e_x \rightarrow^{*} in_{\ell_i} v_i$ 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.	$e = (case\ e_x\ of\ \ell_1\ x_1\ \Rightarrow\ e_1 \mid \cdots \mid \ell_n\ x_n\ \Rightarrow\ e_n)$	assumption
2.	e_x must be reduced before $e_1, \ldots,$ or e_n are reduced	Rule CaseE, Rule caseV
3.	$e_x \rightarrow^{*} in_{\ell_i} v_i$ while producing traces t_x	assumption
4.	after $e_x \rightarrow^{*} in_{\ell_i} v_i$ producing trace t_x, $t = t_x$	2, 3, Rule CaseE
5.	after $e_x \rightarrow^{*} in_{\ell_i} v_i$, $e \rightarrow^{*} [v_i/x_i]e_i$	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 $\xrightarrow{label:\ v} \beta$
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 $e = while\ (e_1)\{e_2\}$, if $e_1 \rightarrow^{*} v1$ 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 ∞-language expression $(t_1, t_2)^{\infty}t_1$.

Proof.

1.	$e = while\ (e_1)\{e_2\}$	assumption
2.	$e \rightarrow if\ e_1\ then\ (e_2; while\ (e_1)\{e_2\})\ else\ unit$	1, Rule whileE
3.	$e_1 \rightarrow^* v_1$ producing trace t_1	assumption
4.	assume $v_1 = false$ while producing trace t_1 in the first iteration	assumption
5.	$e \rightarrow^* unit$ while producing trace t_1	2, 3, 4, Rule ifFalse
6.	assume $v_1 = true$ while producing trace t_1 in the first iteration	assumption
7.	$e \rightarrow^* (e_2;\ while\ (e_1)\{e_2\})$ while producing trace t_1	2, 3, 6, Rule ifTrue
8.	$e_2 \rightarrow^* v_2$ producing trace t_2	assumption
9.	after $e_2 \rightarrow^* v_2$, the produced trace $t = [t_1, t_2]$	3, 7, 8, sequence
10.	the produced trace will be either t_1 or $[t_1, t_2]$ after the first iteration	5, 9
11.	both t_1 and $[t_1, t_2]$ matches ∞-regular expression $(t_1, t_2)^\infty t_1$	Rule of ∞-expression
12.	the trace produced after first iteration matches ∞-regular expression $(t_1, t_2)^\infty t_1$	10, 11
13.	assume that e has run $n(n > 1)$ iterations and the trace t_n produced after the n^{th} iteration matches ∞-regular expression $(t_1, t_2)^\infty t_1$	assumption
14.	values are always added to end of trace	Definition of $\xrightarrow{label:\ v} \beta$
15.	the trace t_{n+1} produced after $(n+1)^{th}$ iteration will either $[t_n, t_1]$ or $[t_n, t_1, t_2]$	10, 13, 14
16.	both $[t_n, t_1]$ and $[t_n, t_1, t_2]$ matches ∞-regular expression $(t_1, t_2)^\infty t_1$	11, 13, Rule of ∞-expression
17.	the trace produced after $(n+1)^{th}$ iteration matches expression $(t_1, t_2)^\infty t_1$	15, 16
	Result from 12, 17	

□

Lemma 15 (Let-statement Traces). Given a well-typed expression e where $e = (let\ x = e_1\ in\ e_2\ end)$: if $e_1 \rightarrow^* v_1$ while producing a trace t_1 and $[v_1/x]e_2 \rightarrow^* v$ while producing a trace t_2, then $e \rightarrow^* v$ 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 $e_1 \rightarrow^* v_1$ producing trace t_1, $t = t_1$	1, Rule letE
3.	values are always added to end of trace	Definition of $\xrightarrow{label:\ v} \beta$
4.	after $[v_1/x]e_2 \rightarrow^* v$, $t = [t_1, t_2]$	2, 3

□

Lemma 16 (If-statement Traces). Given a well-typed expression e where $e = if\ e_3\ then\ e_1\ else\ e_2$: if $e_1 \rightarrow^* v1$, $e_2 \rightarrow^* v_2$, and $e_3 \rightarrow^* v_3$ while producing traces t_1, t_2 and t_3 respectively, then either $e \rightarrow^* v_1$ while producing trace $[t_3, t_1]$ when $v_3 = true$ or $e \rightarrow^* v_2$ while producing trace $[t_3, t_2]$ when $v_3 = false$.

Proof.

1.	e_3 must be reduced before e_1 or e_2 are reduced	Rule ifE, Rule ifTrue, Rule ifFalse
2.	after $e_3 \rightarrow^* v_3$ producing trace t_3, $t = t_3$	1, Rule ifE
3.	if $v_3 = false$, $if\ v_3\ then\ e_1\ else\ e_2 \rightarrow^* e_2$	Rule ifFalse
4.	values are always added to end of trace	Definition of $\xrightarrow{label:\ v} \beta$
5.	if $v_3 = false$, after $if\ v_3\ then\ e_1\ else\ e_2 \rightarrow^* v_2$, $t = [t_3, t_2]$	2, 3, 4
6.	if $v_3 = true$, $if\ v_3\ then\ e_1\ else\ e_2 \rightarrow^* e_1$	Rule ifTrue
7.	if $v_3 = true$, after $if\ v_3\ then\ e_1\ else\ e_2 \rightarrow^* v_1$, $t = [t_3, t_1]$	2, 6, 4
	Result from 5, 7	

□

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

Proof.

1.	$e_2 \rightarrow^* v_2$ while producing a trace t_2	assumption
2.	e_2 does not add or remove values from the list value stored at ℓ	assumption
3.	the trace produced as the expression $\neg empty(!\ell)$ reduces to v is []	Rule derefE, Rule derefValue, Rule inEqE1, Rule inEqE2, Rule ineqFale, Rule ineqT
4.	$e \rightarrow^* v$ while producing trace t that matches $([]\ t_2)^\infty []$	1, Lemma 3
5.	t matches t_2^∞	4, rules of ∞-expression
6.	$\lvert tail(!\ell)\rvert = \lvert\ell\rvert - 1$	Rule listTailValue, Rule listTialEmptyValue
7.	$\ell ::= tail(!\ell)$ reduce $!\ell$ by 1	6, Rule AssignE2, Rule assignValue

8.	$head(\ell)$ does not modify $\|!\ell\|$	Rule listHeadValue, Rule listHeadE
9.	each iteration of while loop reduces $\|!\ell\|$ by 1	2, 6, 7
10.	when $\|!\ell\| > 0$, $\neg empty(!\ell) \rightarrow^* true$	Rule ineqT
11.	when $\|!\ell\| == 0$, $\neg empty(!\ell) \rightarrow^* false$	Rule eqT, ineqF
12.	$\|!\ell\| > 0 \rightarrow^* true$ N times	8, 9, 10
13.	$e \rightarrow^* v$ while producing trace t that matches ${t_2}^N$	4, 12

□

Lemma 18 (CallOnTrigger-statement Traces). Given an expression $e_1 = call(onTrigger,\ e)$: if $\Lambda, \bullet \vdash e_1$ **ok**, and $e_1 \rightarrow^* v_1$ while producing a trace t, then t matches the ∞-expression $(beginOb(e)\ (\neg ob(e))^\infty\ endOb(e))$.

Proof.

1.	$beginOb(e)$ can not occur directly from $e_{\text{onTrigger}}$	Definition of $\cdots \vdash e_1$ **ok**
2.	$endOb(e)$ can not occur directly from $e_{\text{onTrigger}}$	Definition of $\cdots \vdash e_1$ **ok**
3.	$call(onTrigger, e) \rightarrow^* \{[v/x]e_{onTrigger}\}_{onTrigger(e)}$ while producing trace $begin_{onTrigger(e)}$	3, Rule callNonMonitoredFunction
4.	the trace t_1 produced while $[v/x]e_{onTrigger} \rightarrow^* v'$ matched $(\neg ob(e))^\infty$	1, 2, definition of ob(e)
5.	values are always added to end of trace	Definition of $\xrightarrow{label:\ v} \beta$
6.	after $\{[v/x]e_{onTrigger}\}_{onTrigger(e)} \rightarrow^* \{v'\}_{onTrigger(e)}$, $t = [begin_{onTrigger(e)}, t_1]$	4, 5, 6, rule endLabelE
7.	$\{v'\}_{onTrigger(e)} \rightarrow^* v_1$ while producing trace $end_{onTrigger(e)}$	7, Rule endLabelValue
8.	after $\{v'\}_{onTrigger(e)} \rightarrow^* v_1$, the produced trace $t = [begin_{onTrigger(e)}, t_1, end_{onTrigger(e)}]$	5, 6, 7
9.	the trace t produced while $e_1 \rightarrow^* v_1$ matches the ∞-expression $(beginOb(e)\ (\neg ob(e))^\infty\ endOb(e))$	4, 8, Definition of beginOb(e), Definition of endOb(e), Rules of ∞-expression

□

Lemma 19 (CallOnObligation-statement Traces). Given an expression $e_1 = call(onObligation,\ e)$: if $\Lambda, \bullet \vdash e_1$ **ok**, and $e_1 \rightarrow^* v_1$ while producing a trace t, then t matches the ∞-expression $beginOb(e)\ (\neg ob(e))^\infty\ endOb(e)$.

Proof.

1.	$beginOb(e)$ can not occur directly from $e_{\text{onObligation}}$	Definition of $\cdots \vdash e_1$ **ok**
2.	$endOb(e)$ can not occur directly from $e_{\text{onObligation}}$	Definition of $\cdots \vdash e_1$ **ok**
3.	$call(onObligation,\ e) \rightarrow^* \{[v/x]e_{onObligation}\}_{onObligation(e)}$ while producing trace $begin_{onObligation(e)}$	3, Rule callNonMonitoredFunction
4.	the trace t_1 produced while $[v/x]e_{onObligation} \rightarrow^* v'$ matched $(\neg ob(e))^\infty$	1, 2, definition of ob(e)
5.	values are always added to end of trace	Definition of $\xrightarrow{label:\ v} \beta$
6.	after $\{[v/x]e_{onObligation}\}_{onObligation(e)} \rightarrow^* \{v'\}_{onObligation(e)}, t = [begin_{onObligation(e)}, t_1]$	3, 4, 5, rule endLabelE
7.	$\{v'\}_{onObligation(e)} \rightarrow^* v_1$ while producing trace $end_{onObligation(e)}$	7, Rule endLabelValue
8.	After $\{v'\}_{onObligation(rt)} \rightarrow^* v_1$, the produced trace $t = [begin_{onObligation(e)}, t_1,\ end_{onObligation(e)}]$	5, 6, 7
9.	the trace t produced while $e_1 \rightarrow^* v_1$ matches the ∞-expression $(beginOb(e)\ (\neg ob(e))^\infty\ endOb(e))$	4, 8, Definition of beginOb(e) and endOb(e), Rules of ∞-expression

□

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

Proof.

1.	$beginOb(e)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ **ok**
2.	$call(monitor, v)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ **ok**
3.	$call(monitor, v)$ happens only when a monitored function is called	Rule callFromApplication
4.	beginOb(e) cannot exist in t	1, 2, 3
5.	t matched $(\neg beginOb(e))^\infty$	4

□

Lemma 21 (App-no-append-result Traces). Given a program p which is an untrusted application e_{app} with enforced policies, if $\Lambda, \bullet \vdash p$ **ok**, and e_{app} does not contain any monitored functions, and $p \rightarrow^* p'$ while producing a trace t, then t matches the ∞-expression $(\neg begin_{appendRes(r)})^\infty$.

Proof.

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

□

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

Proof.

1.	Rule callFromObligation is the only rule that adds $begin_{appendRes(r)}$	Inspection of the dynamic semantics
2.	no function calls from e_{app} can trigger rule callFromObligation	Rule callFromApplication
3.	no non-function calls outside of e_{app} can trigger rule callFromObligation	Rule callFromApplication
4.	$begin_{appendRes(r)}$ cannot exist in t	1, 2, 3
5.	t matched $(\neg begin_{appendRes(r)})^{\infty}$	4

□

Lemma 23 (No-makeCFG Traces). Given a program p which is an untrusted application e_{app} with enforced policies, if $\Lambda, \bullet \vdash p$ **ok**, and e_{app} does not contain any monitored functions, and $p \rightarrow^* p'$ while producing a trace t, then t matches the ∞-expression $(\neg end_{makeCFG(v_1, v_2):g})^{\infty}$.

Proof.

1.	$end_{makeCFG(v_1, v_2):g}$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ **ok**
2.	$call(monitor, v)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ **ok**
3.	$call(monitor, v)$ happens only when a monitored function is called	Rule callFromApplication
4.	$end_{makeCFG(v_1, v_2):g}$ cannot exist in t	1, 2, 3
5.	t matches $(\neg makeCFG(v_1, v_2))^{\infty}$	4

□

Theorem 9 (Atomic Obligation). For all p, t, and t', if $\Lambda, \bullet \vdash p$ **ok**, and $p \xrightarrow{t}{}^* p'$, and t matches the ∞-expression $((.^{\infty})\ beginOb(e_n)\ t'\ beginOb(e_m)\ .(.^{\infty}))$ then t' matches the ∞-expression $((.^{\infty})\ endOb(e_n)\ (.^{\infty}))$

Proof.

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

1.	t matches $(\neg\ beginOb(e))^{\infty}$	Lemma 9
2.	The theorem holds vacuously in this case because t does not match $((.\infty)\ beginOb(e_n)\ t'\ beginOb(e_m)\ (.\infty))$	1, Rules of ∞-expression

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

1.	$beginOb(e)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ ok
2.	$call(monitor, v)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ 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 $call(o_1.onTrigg, o_1.evt)$ on line 24 reducing to v_{ot} is $[beginOb(e),\ (\neg ob(e))^{\infty},\ endOb(e)]$	Lemma 7, Definition of monitor
6.	the trace produced while $call(o_2.onOblig, o_2.rt)$ on line 25 reducing to v_{oo} is $[beginOb(e),\ (\neg ob(e))^{\infty},\ endOb(e)]$	Lemma 8, 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	Lemma 2, Definition of monitor
8.	the trace t_1 produced while *case* expression that spans from line 24 to 25 reducing to v_1 is $[beginOb(e),\ (\neg ob(e))^{\infty},\ endOb(e)]$	5, 6, 7, Lemma 2, Rules of ∞-expression
9.	taken the *case* expression that spans from line 24 to 25 as its else branch, the $if\text{-}then\text{-}else$ expression spans from line 23 to 26 does not introduce	Lemma 5, Definition of monitor
10.	the trace t_2 produced while the $if\text{-}then\text{-}else$ expression spans from line 23 to 26 reducing to v_2 is either $[(\neg beginOb(e))^{\infty}]$ or $[(\neg beginOb(e))^{\infty}, t_1]$	8, 9, Lemma 5, Rule of ∞-expression

11.	containing the if-$else$-$then$ expression that spans from line 23 to 26, the let expression that spans from line 15 to 37 does not introduce additional $beginOb(e)$ in the overall trace	Lemma 4, Definition of monitor
12.	the trace produced while the let expression spans from line 15 to 37 reducing to v_3 matches is either $[(\neg beginOb(e))^\infty,\ t_1\ , (\neg beginOb(e))^\infty]$ or $[(\neg beginOb(e))^\infty]$	10, 11, Lemma 4, Rules of ∞-language expression
13.	containing the let expression that spans from line 15 to 37, the $while$ expression that spans from line 5 to 37 does not introduce additional $beginOb(e)$ in the overall trace	Lemma 6, Definition of monitor
14.	the trace produced while the $while$ expression spans from line 5 to 37 reducing to v_4 is either $[(\neg beginOb(e))^\infty]$ or $[((\neg beginOb(e))^\infty,\ t_1\ , (\neg beginOb(e))^\infty)^\infty,\ (\neg beginOb(e))^\infty]$	12, 13, Lemma 3 Rules of ∞-expression
15.	containing the $while$ expression that spans from line 5 to 37, the monitor function doesn't introduce additional $beginOb(e)$ in the overall trace	Lemma 9, Definition of monitor
16.	the trace t produced when the overall monitor function reducing to v is either $[((\neg beginOb(e))^\infty,\ t_1\ , (\neg beginOb(e))^\infty)^\infty,\ (\neg beginOb(e))^\infty]$ or $[(\neg beginOb(e))^\infty]$	14, 15, Lemma 1, 3, 4 Rules of ∞-expression
17.	When t is $[(\neg beginOb(e))^\infty]$, the theorem holds vacuously because t does not match $((.^\infty)\ beginOb(en)\ t'\ beginOb(em)\ (.^\infty))$	16, Rules of ∞-expression
18.	When t is $[((\neg beginOb(e))^\infty,\ t_1\ , (\neg beginOb(e))^\infty)^\infty,\ (\neg beginOb(e))^\infty]$ and $(beginOb(e)\ \neg ob(e)^\infty\ endOb(e))$ exists more than once in t, t' must always include endOb(e) Results from 17 and 18	16, Rules of ∞-expression

□

Theorem 10 (Conflict Resolution). For all programs p such that $\Lambda, \bullet \vdash p$ **ok** and $p \xrightarrow{t}{}^{*} p'$, t matches the ∞-expression $(\neg beginOb(e))^\infty\ (v_{true}(e_n)\ beginOb(e_n)\ (\neg beginOb(e))^\infty)^\infty$ where: $v_{true}(e) ::= (begin_{vote(e)}\ (\neg beginOb(e))^\infty\ end_{vote(e)\,:\,v_n})^N\ begin_{vc(v_1\,::\,\cdots\,::\,v_N)}\ (\neg beginOb(e))^\infty\ end_{vc(v_1::\cdots::v_N)\,:\,true}$.

Proof.

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

1.	t matches $(\neg\ beginOb(e))^\infty$	Lemma 9
2.	t matches $((\neg beginOb(e))^\infty\ v_{true}(e_N)\ beginOb(e_N))^\infty\ (\neg beginOb(e))^\infty$	1, Rules of ∞-expression

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

1.	$beginOb(e)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ ok
2.	$call(monitor, v)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ 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 t_{ot} produced while $call(o_1.onTrig, o_1.evt)$ on line 24 reducing to v_{ot} is $[beginOb(e),\ (\neg ob(e))^\infty,\ endOb(e)]$	Lemma 7, Definition of monitor
6.	the trace t_{oo} produced while $call(o_2.onOblig, o_2.rt)$ on line 25 reducing to v_{oo} matches $[beginOb(e),\ (\neg ob(e))^\infty,\ endOb(e)]$	Lemma 8, 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	Lemma 2, Definition of monitor
8.	the trace t_1 produced while the $case$ expression reducing to v_1 is $[beginOb(e),\ (\neg ob(e))^\infty,\ endOb(e)]$	7, 8, Lemma 2 5, 6, 9, Rules of ∞-expression
9.	taken 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 $beginOb(e)$ in the overall trace	Lemma 5, Definition of monitor
10.	the trace t_2 produced while the condition expression $call(c.vc, !votes)$ on line 23 reducing to v_2 is $[begin_{vc(v_1::\cdots::v_N)},\ end_{vc(v_1::\cdots::v_N)}]$	Rule callNonMonitoredFunction, Rule endLabelValue, Definition of monitor
11.	the trace t_3 produced while the if-$then$-$else$ expression spanned from line 23 to 26 reducing to v_3 is either $[begin_{vc(v_1::\cdots::v_n)},\ end_{vc(v_1::\cdots::v_n):false}]$ or $[begin_{vc(v_1::\cdots::v_n)},\ end_{vc(v_1::\cdots::v_n):true},\ t_1]$	9, 10, 11, Lemma 5, Rules of ∞-expression
12.	the trace t_4 produced while the expression $call(!pol.policy.vote, ob)$ one line 21 reducing to v_4 is $[begin_{vote(e)},\ end_{vote(e):v_N}]$	Rule listHeadValue, Rule listTailValue, Rule listTailEmptyValue

13.	the trace t_5 produced while the *while* expression spans from line 30 to 34 reducing to v_5 is $[begin_{vote(e)},\ end_{vote(e):v_N}]^N$	13, Lemma 1, Lemma 6, Definition of monitor
14.	the trace t_6 produced while the expressions spans from line 30 to 26 reducing to v_6 is $[begin_{vc(v_1:: \cdots ::v_n)},\ end_{vc(v_1:: \cdots ::v_n):false}]$ or $[t_5,\ begin_{vc(v_1:: \cdots ::v_n)},\ end_{vc(v_1:: \cdots ::v_n):true},\ t_1]$	12, 13, Lemma 1, Definition of monitor
15.	the trace t_7 produced while the *while* expression spans from line 5 to 37 reducing to v_7 is $[{t_6}^\infty]$	14, Lemma 3, Definition of monitor
16.	the trace t produced when the overall monitor function reducing to v $[(\neg beginOb(e))^\infty,\ t_7,\ (\neg beginOb(e))^\infty]$	15, Lemma 1, Lemma 3, Lemma 4, Definition of monitor, Rules of ∞-expression
17.	t can be simplified to $[(\neg beginOb(e))^\infty,\ ((t_5,\ begin_{vc(v_1:: \cdots ::v_n)},\ end_{vc(v_1:: \cdots ::v_n):true},\ beginOb(e_n))?)^\infty]$	16, Rules of ∞-expression
18.	all possible traces produced by p match the ∞-expression $(((\neg beginOb(e))^\infty,\ v_{true}(e_n),\ beginOb(e_n))^\infty\ (\neg beginOb(e))^\infty)$	17, Rules of ∞-expression Definition of $v_{true}(e_n)$

□

Theorem 11 (Obligation Reaction Part 1). For all programs p such that $\Lambda, \bullet \vdash p$ **ok** and $p \xrightarrow{t}{}^{*} p'$, t matches the ∞-expression $((\neg begin_{appendRes()})^\infty\ (begin_{appendRes()}\ .^\infty\ endOb(e)\ (.^\infty\ end_{makeCFG(onObligation,\ v)\ :\ g})^N)?)^\infty$.

Proof.

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

1.	t matches $(\neg\ begin_{appendRes()})^\infty$	Lemma 10
2.	t matches re	1, Rules of ∞-expression

Case 2: no obligations contain any monitored functions calls:

1.	t matches $(\neg\ begin_{appendRes()})^\infty$	Lemma 11
2.	t matches re	1, Rules of ∞-expression

Case 3: e_{app} contains at least one monitored functions call:

1.	$begin_{appendRes}$ only can occur with the context of executions of obligations	Definition of $\cdots \vdash p$ ok
2.	only lines 24 and 25 of monitor can result in a trace including $begin_{appendRes}$	1, Definition of monitor
3.	the trace produced while $call(o_1.onTrig, o_1.evt)$ on line 24 reducing to v_{ot} is $[(\neg begin_{appendRes()})^\infty,\ begin_{appendRes()},\ .\infty,\ endOb(e)]$	Rule calFromObligation, Definition of monitor
4.	the trace produced while $call(o_2.onOblig, o_2.rt)$ on line 25 reducing to v_{ot} is $[(\neg begin_{appendRes()})^\infty,\ begin_{appendRes()},\ .\infty,\ endOb(e)]$	Rule calFromObligation, Definition of monitor
5.	containing of the two call expressions, the trace produced while the *case* expression spans from line 24 to 25 reducing to v_2 is $[(\neg begin_{appendRes()})^\infty,\ begin_{appendRes()},\ .\infty,\ endOb(e)]$	3, 4, Lemma 2, Definition of monitor Rules of ∞-expression
6.	the trace produced while the *then* branch spans from line 28 to 35 reducing to v_3 is $[.\infty,\ end_{makeCFG(onObligation,v):g}{}^N]$	Lemma 8, Definition of monitor
7.	the trace produced while the *if-then-else* expression spans from line 27 to 36 evaluates to v_4 is $[.\infty,\ end_{makeCFG(onObligation,v):g}{}^N]$?	6, Definition of monitor, Lemma 5
8.	the trace t_6 produced while the expression spans from line 23 to 36 reducing to v_6 is $[(\neg begin_{appendRes()})^\infty,\ (begin_{appendRes()})^\infty,\ (.\infty),\ endOb(e),\ (.\infty\ end_{makeCFG(onObligation,v):g})^N)?]$	7, Lemma 1, Lemma 2, Lemma 4, Definition of monitor
9.	the trace t_7 produced while the *while* expression spans from line 5 to 37 reducing to v_7 is $[t_6]^\infty$	8, Lemma 3, Lemma 4,
10.	the trace t_8 produced when the overall monitor function is reduced to v_8 is $[(\neg begin_{appendRes()})^\infty,\ t_7,\ (\neg begin_{appendRes()})^\infty]$	2, 9, Lemma 1, Definition of monitor
11.	p produces the trace $t = [(\neg begin_{appendRes()})^\infty,\ t_8^\infty,\ (\neg begin_{appendRes()})^\infty]$	2, 10
12.	t can be simplified to $[(\neg begin_{appendRes()})^\infty,\ ((\neg begin_{appendRes()})^\infty\ (begin_{appendRes()}\ (.\infty)\ endOb(e)\ (.\infty\ end_{makeCFG(onObligation,v):g})^N)?)^\infty]$	11, Rules of ∞-regular expression
13.	t matches ∞-expression $((\neg begin_{appendRes()})^\infty\ (begin_{appendRes()}\ (.\infty)\ endOb(e)\ (.\infty\ end_{makeCFG(onObligation,v):g})^N)?)^\infty$	12, Rules of ∞-regular expression

□

Theorem 12 (Obligation Reaction Part 2). For all programs p s.t. $\Lambda, \bullet \vdash p$ **ok**, and $p \xrightarrow{t}{}^{*} p'$, and p's monitor is $(M,\ fun_{mon},\ p_1 :: \cdots :: p_n,\ e_{os},\ e_{vc})$ where the functions $e_{vc},\ p_1.onTrigger, \ldots, p_n.onTrigger,\ p_1.onObligation, \ldots, p_n.onObligation$ terminate, for each event $end_{makeCFG(v1,\ v2)\ :\ g}$ in t there must exist a $v_{true(g)}$ or $v_{false(g)}$ in t where:
$v_{true}(e)$ ::= $(begin_{vote(e)}\ (\neg beginOb(e))^{\infty}\ end_{vote(e)\ :\ v_n})^N\ begin_{vc(v_1\ ::\ \cdots\ ::\ v_N)}\ (\neg beginOb(e))^{\infty}\ end_{vc(v_1 :: \cdots ::v_N)\ :\ true}$ and
$v_{false}(e)$ ::= $(begin_{vote(e)}\ (\neg beginOb(e))^{\infty}\ end_{vote(e)\ :\ v_n})^N begin_{vc(v_1\ ::\ \cdots\ ::\ v_N)}\ (\neg beginOb(e))^{\infty}\ end_{vc(v_1 :: \cdots ::v_N)\ :\ false}$.
Proof.

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

1.	t matches $(\neg end_{makeCFG(v_1,v_2):g})^{\infty}$	Lemma 10
2.	The theorem holds vacuously in this case because t does not match either $v_{true}(e)$ or $v_{false}(e)$	1, Rules of ∞-expression

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

1.	$end_{makeCFG(v_1,v_2):g}$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ ok
2.	$call(monitor, v)$ cannot occur directly from e_{app}	Definition of $\cdots \vdash p$ ok
3.	only the two expressions *makeCFG* on lines 7 and 33 of monitor code can result in a trace including $end_{makeCFG(v_1,v_2):g}$	1, Definition of monitor
4.	the result of the expression *makeCFG* on line 7 is immediately appended to a list stored in $\ell_{obQueue}$	Rules listAppendE1, listAppendE2, Rule listAppendValue, assignValue, Rule derefValue, Definition of monitor
5.	the list stored in $\ell_{obQueue}$ is prepended to a list stored in $\ell_{obStack}$ on the line 10 of the monitor code resulting in all results of *makeCFG* in line 7 being included in $\ell_{obStack}$	4, Rules listPrependE1, listPrependE2, Rule listPrependValue, assignValue, Rule derefValue, Definition of monitor
6.	the result of the expression *makeCFG* on line 33 immediately appended to a list stored in $\ell_{obQueue}$	Rules listAppendE1, listAppendE2, Rule listAppendValue, assignValue, Rule derefValue, Definition of monitor
7.	the list stored in $\ell_{obQueue}$ is prepended to a list stored in $\ell_{obStack}$ on line 35 of the monitor code resulting in all results of *makeCFG* in line 33 being included in $\ell_{obStack}$	6, Rules listPrependE1, listPrependE2, Rule listPrependValue, assignValue, Rule derefValue, Definition of monitor
8.	results from all possible calls to *makeCFG* are stored in $\ell_{obStack}$	1, 2, 3, 5, 7
9.	the trace produced while code from lines 30 to 26 is reduced to a value is either $[begin_{vc(v_1:: \cdots ::v_N)},\ end_{vc(v_1:: \cdots ::v_N):false}]$ or $[(begin_{vote(e)},\ end_{vote(e):v_N})^N,\ begin_{vc(v_1:: \cdots ::v_N)},\ end_{vc(v_1:: \cdots ::v_N):true}, beginOb(e),\ (\neg ob(e))^{\infty},\ endOb(e)]$	Theorem 3(14)
10.	the expression $obQueue := head(!obStack)$ on line 12 of the monitor code takes $\ell_{obStack}$'s first list of obligations and stores it in $\ell_{obQueue}$	Rules derefValue, Rule headValue, Rule assignValue, Definition of monitor
11.	the expression $let\ ob = head(!obQueue)$ on line 13 of the monitor code takes the first obligation stored in $\ell_{obQueue}$	Rules derefValue, Rule headValue, Rule assignValue, Definition of monitor
12.	if $!\ell_{obQueue} = [ob]$, then the expression $\neg empty(tail(!obQueue))$ on line 16 of of the monitor code will evaluate to $false$	Rules derefValue, Rule headValue, Rule assignValue, Definition of monitor
13.	If $\neg empty(tail(!obQueue))$ evaluates to $false$, then the first element of $!\ell_{obStack}$ will be removed after executing the expression $obstack := tail(!obStack)$ on line 17 of the monitor code	12, Rules tailValue, Rule derefValue Rule assignValue, Definition of monitor
14.	if $!\ell_{obQueue} = [ob, \ldots]$, then $\neg empty(tail(!obQueue))$ on line 16 of of the monitor code will evaluate to $true$	Rules derefValue, Rule headValue, Rule assignValue, Definition of monitor
15.	If $\neg empty(tail(!obQueue))$ evaluates to $true$, the first element of $!\ell_{obStack}$ will be replaced with $tail(!\ell_{obQueue})$ after executing the expression on line 16 of the monitor code	14, Rules tailValue, Rule derefValue, Rule listPrependValue, Rule assignValue, Definition of monitor
16.	for each iteration, one obligation will be removed from $\ell_{obStack}$ and stored in ℓob	10, 11, 13, 15
17.	no locations besides lines 10, 16, 17, and 35 in the monitor code assign to $\ell_{obStack}$	Definition of monitor, Rule assignValue
18.	ob is only assigned to on line 13 of the monitor code where it is assigned with $\ell_{obStack}$'s the first list of obligations	11, Definition of monitor

19.	*ob* is the parameter that passed into *vote* function of the monitor code	Definition of monitor
20.	an obligation that is ever added to $\ell_{obStack}$ generates a subtrace matches either $(begin_{vc(v_1:: \cdots ::v_N)}\ end_{vc(v_1:: \cdots ::v_N):false})$ or $((begin_{vote(e)}\ end_{vote(e):v_N})^N\ begin_{vc(v_1:: \cdots ::v_N)}\ end_{vc(v_1:: \cdots ::v_N):true}\ beginOb(e)\ (\neg ob(e))^{\infty}\ endOb(e))$	9, 16, 17, 18, 19
21.	for each $end_{makeCFG(v_1,v_2):g}$ in t there must exist a matching $v_{true(g)}$ or $v_{false(g)}$	8, 20

□

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 ∞-expressions e_{pre} for pre-obligations, e_{post} for post-obligations and $e_{ongoing}$ for ongoing obligations where:
$e_{pre} = ((.\infty)\ begin_{f(x)}\ (.\infty) begin_{monitor(act(f,x))}\ (.\infty)\ end_{f(x):v}\ .(\infty))$,
$e_{post} = ((.\infty)\ end_{f(x):v}\ (.\infty)\ begin_{monitor(res(act(f,x),rt))}\ (.\infty))$, and
$e_{ongoing} = (e_{pre} \mid e_{post})\ (.\infty)\ (e_{pre} \mid e_{post})$.

Proof.

Case 1: pre-obligation:

1.	assume p_1 has a policy Pol_1 with the onTrigger $e_{onTrigger}(e) = if\ matches(e, act(\text{“}print\text{”}, *))\ then\ call(log, e)\ else\ unit$	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 $call(print, v)$, $call(monitor, act(\text{“}print\text{”}, v))$ will be triggered	2, Rule callFromApplication
5.	after $call(monitor, act(\text{“}print\text{”}, v))$ is executed, $begin_{monitor(act(\text{“}print\text{”},v))}$ gets added to the trace resulting in a trace of $[(.\infty),\ begin_{print(v)},\ begin_{monitor(act(\text{“}print\text{”},v))}]$	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 $[(.\infty),\ begin_{print(v)},\ begin_{monitor(act(\text{“}print\text{”},v))},\ (.\infty),\ end_{print(x):v},\ (.\infty)]$	5, 6, Rule endLabelValue
8.	execution of p_1 results in a trace of $[(.\infty),\ begin_{print(v)},\ begin_{monitor(act(\text{“}print\text{”},v))},\ (.\infty),\ end_{print(x):v},\ (.\infty)]$	2, 7
9.	execution of p1 results in a trace of that matches re_{preOb}	8, Rules of ∞-expression

Case 2: post-obligation:

1.	assume p_2 has a policy Pol_2 with the onTrigger $e_{onTrigger}(e) = if\ matches(e, res(\text{“}print\text{”}, *))\ then\ call(log, e)\ else\ unit$	Assumption
2.	$print \in F$	1, Definition of F
3.	when e_{app} attempts to execute the expression $call(print, v)$, the $call(monitor, res(\text{“}print\text{”}, v))$ 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 $[.\infty,\ end_{log(v):r}]$	5, Rule callFromObligation, Rule endLabelValue
7.	a successful execution of $call(print, v)$ triggers $call(monitor, res(act(\text{“}print\text{”}, v), r))$	Rule callFromApplication
8.	after a successful execution of $call(monitor, res(act(\text{“}print\text{”}, v), r))$, the trace is $[(.\infty),\ end_{log(v):r},\ (.\infty), begin_{monitor(res(act(\text{“}print\text{”},v),r))}]$	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 re_{postOb}	9, Rules of ∞-expression

Case 3: ongoing-obligation:

1.	assume p_3 has policies as specified in Case 1 and Case 2	Assumption
2.	execution of p_3 matches the ∞-expression re_{preOb} and re_{postOb}	Case 1, Case 2
3.	execution of p_3 matches the ∞-expression $re_{ongoingOb}$	2, Rules of ∞-expression

□

Theorem 14 (Policy Permutability).
If p is a well-typed program with monitor $(fun_{mon},\ p_1 :: \cdots :: p_n,\ e_{os},\ e_{vc})$ and $p \rightarrow^* p'$ while producing trace t, then there exists e_{os} and e_{vc} such that if q is a well-typed program with monitor $(fun_{mon},\ p'_1 :: \cdots :: p'_n,\ e_{os},\ e_{vc})$ where $p'_1 :: \cdots :: p'_n$ is a permutation of $p_1 :: \cdots :: p_n$ and $q \rightarrow^* q'$ while producing trace t' then $t \approx t'$ (all $e_{\text{onTrigger}}$, $e_{\text{onObligation}}$, e_{os}, and e_{vc} must be pure functions).
Proof.

1.	only $call(e_1, e_2)$ and $makeCFG(e)$ expressions can result in values being added to traces	Rules callFromApplication, Rule makeCFGValue, Rule callFromObligation, Rule endLabelValue, Rule callNonMonitoredFunction
2.	the parts of t and t' generated outside of calls to the monitor are identical	Definitions of p and n, Definition of monitor, Definition of function scope of e_{app}
3.	calls to monitor from p and q will differ only in the order of policies in parameter	Definitions of p and n, Rule callFromApplication
4.	calls to monitor from p and q result in traces that are equivalent if the current traces are equivalent	Definition of trace equivalence, 3
5.	only lines 6, 10, 23, 25, 24, 25 and 41 of monitor result in values being added to the trace	1, Definition of monitor
6.	calls to the expression $makeCFG(e)$ on line 6 of the monitor code in p and q will differ only in parameters	Definition of Monitor, Rule callNonMonitoredFunction
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	Definition of trace equivalence, 6
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	Rule callNonMonitoredFunction, Definition of Monitor
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	Rule callNonMonitoredFunction, Rule while, Definition of Monitor
10.	$votingPols$ in p and q are identical if $votingPolsList$ in p and q are identical	Definition of Monitor, Rule assignValue
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	Rule assignValue, Rule listTailValue, Definition of monitor
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	Definition of monitor, Rule assignValue, Rule callNonMonitoredFunction
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 $e_{os}(obs) = call(orderByPolicyName, obs)$	Assumption
14.	the traces result from the execution of the expression line 25 in p and q will be identical if $e_{os}(obs) = call(orderByPolicyName, obs)$	8, 9, 10, 11, 12, 13
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	Definition of monitor, Rule callNonMonitoredFunction
16.	$votes$ in p and q will be identical if traces produced on line 25 of p and q are identical	Definition of monitor, Rule assignValue, Rule derefValue, Rule listAppendValue
17.	the traces result from each iteration of line 25 will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$	13, 14, Definition of monitor
18.	the traces result from the execution of the expression line 23 in p and q will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$	15, 16, 17, Definition of monitor

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 in p and q are identical	Definition of Monitor
20.	the traces result from the execution of the $call(vc, !votes)$ expression in p and q will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$	15, 16, 17, 18, Definition of Monitor
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	Definition of monitor, Rule while, Rule listTailValue, listHeadValue, listAppendValue, listPrependValue
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 $e_{os}(obs) = call(orderByPolicyName, obs)$	19, 20, 21, Definition of monitor
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	Definition of monitor, Rule while, Rule listHeadValue, Rule listTailValue, Rule makeCFGValue
24.	the traces result from the execution of the expressions on line 9 for each iteration in p and q will be identical when $e_{os}(obs) = call(orderByPolicyName, obs)$	12, 13, 23
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	Definition of Monitor, Rule makeCFGValue, Rule while
26.	$votingPols$ in p and q will be identical when $e_{os}(obs)$ is defined as $call(orderByPolicyName, obs)$	10, 11, 12 13
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 $call(orderByPolicyName, obs)$	25, 26
28.	when $e_{os}(obs) = call(orderByPolicyName, obs)$, $t = t'$	2, 4, 5, 7, 14, 18, 22, 24, 27

□