Composition of Atomic-Obligation Security Policies

Danielle Ferguson

University of South Florida
Composition of Atomic-Obligation Security Policies

by

Danielle Ferguson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Srinivas Katkoori, Ph.D.
Yao Liu, Ph.D.
Huseyin Arslan, Ph.D.
Brendan Nagle, Ph.D.

Date of Approval:
April 2, 2020

Keywords: software security, programming languages, policy composition, obligations,
software engineering

Copyright © 2020, Danielle Ferguson
Dedication

This dissertation is dedicated to the memory of my mother, Carolyn, who always inspired me to follow my dreams and to my husband, Ryan, who was there for me throughout its completion.
Acknowledgments

I would like to express my deepest gratitude to my major professor, Dr. Ligatti, for his guidance throughout the completion of this work as well as my committee members, Drs. Katkoori, Liu, Arslan, and Nagle for the time and energy spent reviewing this dissertation. I would additionally like to thank the many students that have worked on this project with me over the years: Yan Albright, Kevin Orr, Tyler Hanks, Cory Juhlin, Daniel Lomsak and Zach Carter.
Table of Contents

List of Tables ......................................................................................... iii
List of Figures ......................................................................................... iv
Abstract ....................................................................................................... v

Chapter 1: Introduction ................................................................................. 1

Chapter 2: Goals ......................................................................................... 6
  2.1 Obligation-Type Support ................................................................. 6
  2.2 Atomic Obligations .......................................................................... 7
  2.3 Obligations with Side Effects .......................................................... 7
  2.4 Complete Mediation of Obligations ............................................... 8
  2.5 Turing Completeness ...................................................................... 8
  2.6 Conflict Resolution .......................................................................... 8
  2.7 Custom Composition Operators ...................................................... 9

Chapter 3: Related Work .............................................................................. 11
  3.1 XACML ......................................................................................... 11
  3.2 Polymer ........................................................................................ 12
  3.3 Ponder .......................................................................................... 13
  3.4 SPL ............................................................................................ 13
  3.5 Heimdall ...................................................................................... 14
  3.6 Rei ............................................................................................. 15
  3.7 Aspect Oriented Programming ........................................................ 15

Chapter 4: The PoCo Monitor Architecture ................................................. 16
  4.1 Monitor Operation ........................................................................... 16
  4.2 Monitor Configuration ...................................................................... 18
  4.3 Obligations ..................................................................................... 19
  4.4 Complete Mediation ........................................................................ 20
  4.5 Non-termination of Policy Enforcement ........................................... 21

Chapter 5: PoCo Language ......................................................................... 22
  5.1 Syntax .......................................................................................... 22
  5.2 Static Semantics .............................................................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Dynamic Semantics</td>
<td>31</td>
</tr>
<tr>
<td>5.4</td>
<td>Type Safety</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>The onTrigger Policy Function</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>The vote Policy Function</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>The onObligation Policy Function</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>Parameterized Policies</td>
<td>47</td>
</tr>
<tr>
<td>6.5</td>
<td>Local Policy State</td>
<td>48</td>
</tr>
<tr>
<td>6.6</td>
<td>Complete Example Policy Specifications</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>Policies</td>
<td>54</td>
</tr>
<tr>
<td>7.2</td>
<td>Vote Combinator</td>
<td>54</td>
</tr>
<tr>
<td>7.3</td>
<td>Obligation Scheduler</td>
<td>56</td>
</tr>
<tr>
<td>7.4</td>
<td>Monitor Operation</td>
<td>57</td>
</tr>
<tr>
<td>8.1</td>
<td>Definition of ( \infty )-Languages</td>
<td>61</td>
</tr>
<tr>
<td>8.2</td>
<td>Atomicity of Obligations</td>
<td>62</td>
</tr>
<tr>
<td>8.3</td>
<td>Conflict Resolution</td>
<td>63</td>
</tr>
<tr>
<td>8.4</td>
<td>Obligation Reaction</td>
<td>63</td>
</tr>
<tr>
<td>8.5</td>
<td>Obligation Completeness</td>
<td>64</td>
</tr>
<tr>
<td>8.6</td>
<td>Policy Permutability</td>
<td>65</td>
</tr>
<tr>
<td>9.1</td>
<td>Conclusions</td>
<td>68</td>
</tr>
<tr>
<td>10</td>
<td>References</td>
<td>70</td>
</tr>
<tr>
<td>A</td>
<td>Appendix A: Proof of Type Safety</td>
<td>74</td>
</tr>
<tr>
<td>B</td>
<td>Appendix B: Proof of Obligation Properties</td>
<td>151</td>
</tr>
<tr>
<td>C</td>
<td>Appendix C: Copyright Permissions</td>
<td>167</td>
</tr>
</tbody>
</table>
List of Tables

Table 3.1 Summary of existing policy-specification languages........................... 12
List of Figures

Figure 4.1  Obligations are either pre-on-action or pre-on-result. ....................... 17
Figure 5.1  Formal syntax for PoCo ................................................................. 24
Figure 7.1  Policy scheduler flow ........................................................................ 57
Figure 7.2  The PoCo monitor algorithm............................................................. 58
Abstract

There has been significant work to date on policy-specification languages that allow specification of arbitrary obligations, but there continues to exist open challenges in the composition of these arbitrary obligations, especially when obligations can be complex (i.e. consist more than one action). There are currently no solutions that allow complete and automatic resolution of conflicts between policies and other policies’ obligations or that allow policies to react to the complex obligations of other policies. In particular, there is minimal work that considers the benefits and challenges of allowing complex obligations that operate in an atomic fashion, that is that execute in their entirety or not at all. This dissertation presents PoCo, a policy-specification language and enforcement system that allows for the principled composition of atomic-obligation policies. PoCo enables policies to interact meaningfully with other policies’ obligations and thus prevents the unexpected and insecure behaviors that can arise with partially executed obligations or obligations that violate other policies. Specifically, this dissertation presents the organization and operation of the PoCo security policy and enforcement system and an analysis of the PoCo language’s formal syntax and semantics as well as several specific and useful properties of this language.
Chapter 1: Introduction\(^1\)

Security-policy composition is a complex problem in software security with many previously proposed solutions. Due to the complexity of handling conflicts that arise when simultaneously enforced policies include requirements that compete with each other. The difficulty of handling these conflicts increases significantly when policies have the option of injecting supplemental logic, referred to as an obligation, into the execution of the application being monitored. To date, composition of obligation-based policies does not have a complete solution; many enforcement systems are domain specific and existing general-purpose systems may cause policies’ obligations to be composed in unexpected or undesirable ways. These general-purpose systems may allow obligations to execute even when they are in violation of other policies being enforced or they may allow partial execution of obligations, which may result in inconsistent or unexpected behavior.

As software increases in complexity, the frequency and severity of security vulnerabilities increases [22]. This increase in security vulnerabilities makes managing the policies that mitigate them more challenging; policy enforcement becomes a disorderly array of security mechanisms that each affect the overall behavior of the system in unexpected ways, or the enforced policy expands into a complex specification that combines many rules making it easy to affect more than the desired rules when making policy changes. Due to these non-optimal approaches, complex security policies increase the likelihood of errors in their enforcement.

\(^1\)Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.
As established by standard software-engineering discipline, maintaining modules of functionality that allow each security concern to be addressed individually is significantly more straightforward and less error prone. More complex policies can be built as compositions of these modules of functionality. Composition of classic safety properties [16] is trivial because the only decision enforced is whether to permit or deny a specific action; these decisions can be combined into a single decision using Boolean operators. Unfortunately, such simple policies are not expressive enough for practical policies because they do not give policies the ability to execute alternative or additional actions in order to meet their goals. These injected actions are commonly referred to as obligations [35], and they can significantly increase the complexity of policy composition.

The ability of security policies to include obligations is critical for the expressiveness of policy enforcement because it allows enforcement of many policies that are impossible when the mechanism only supports enforcement of simple policies that allow or deny specific actions. For example, a policy that grants or denies banking account fund transfers might also include an obligation to log such requests for later auditing, or a policy intended to prevent unintentional file deletion may have an obligation that prompts the user to confirm that the action was, in fact, intended before rendering a decision on whether to permit the requested delete action.

The complexity of identifying and resolving conflicts in policies that include obligations is well known (e.g., [3, 29, 7]). Failing to manage these conflicts often leads to unexpected system behavior or even security vulnerabilities due to unmonitored execution of actions that should have been restricted. Consider the policies $P_{\text{download}}$ and $P_{\text{popup}}$ that disallow file downloads and window pop-ups respectively. $P_{\text{download}}$ additionally defines an obligation to pop up a warning when the target application attempts to download a file. This warning message violates $P_{\text{popup}}$ by displaying a pop-up window. A policy $P$ that composes $P_{\text{download}}$ and $P_{\text{popup}}$ using conjunction (i.e., strictly enforcing both $P_{\text{download}}$ and
policy. If the composed policy $P$ does not validate that $P_{download}$’s obligation does not violate $P_{popup}$’s restrictions, $P$ would allow pop-ups and therefore not be strictly enforcing $P_{popup}$. It is, however, not necessarily the case that a strict interpretation of policy restrictions is automatically the best course of action. It is possible that the intent of $P_{popup}$ is to prevent only the pop-up actions initiated by the target application and would therefore allow the pop-up action to occur if it originates from another policy. However, these decisions should be made consciously by the policy architect and not by the enforcement system. This conscious decision to allow some instances of a restricted action furthers the policy architect’s understanding of how these policies will behave when they are composed.

In addition to these fairly obvious direct policy conflicts, policies may also need the ability to react to other policies’ obligations after they execute in order to properly account for security-relevant actions that occur during their execution. For example, a policy that attempts to limit the number of files that a particular application has open at any given time will need access to an accurate count of the currently open files—including those that have been opened or closed by other policies’ obligations. If this open-file-limiting policy is unable to observe and react to actions that have been performed by other policies’ obligations, it is not possible to accurately enforce it.

This dissertation presents a new security-policy specification language and enforcement system. This enforcement system, called PoCo (short for Policy Composition), is designed such that it:

- Allows composition of complex atomic obligations with provable guarantees regarding its operation
- Supports a wide range of obligations, including those known as pre-, post-, and ongoing obligations
• Allows policies’ obligations to be effectful and specified in a Turing-complete language

• Enables conflict resolution between policies and other policies’ obligations by using static analysis of obligation control flow

• Allows policies to monitor and react to other policies’ obligations

• Supports custom policy-composition operators that allow granular control over how policies interact with each other

As far as could be ascertained from a survey of existing related work, PoCo is the first system that provides support for conflict resolution among arbitrary atomic obligations and policies including the ability for policies to react to other policies’ obligations after they have executed. In addition to the architectural design of the PoCo enforcement system, this dissertation also presents and analyzes the PoCo language’s formal syntax and semantics as well as specific language properties. Additionally, it presents an overview of the implementation and case study that was completed to prove the effectiveness of the PoCo design.

This dissertation is based on a paper presented at ICSCA 2020 [9], which was a collaboration with Yan Albright, Daniel Lomsak, Tyler Hanks, Kevin Orr and Jay Ligatti. As part of this project, I worked extensively on the formal definition of the PoCo language and the design of the PoCo enforcement system’s architecture and operation. I also developed and proved five specific properties of the PoCo language and system showing that the designed system meets all of the project’s defined goals. This dissertation focuses on my specific contributions to the project but includes additional material developed by my teammates, in order to provide context for my contributions. This dissertation extends the conference research paper [9] by presenting:

• A detailed explanation of the formal language

• A complete listing of all static- and dynamic-semantics rules
• Full example policies written in the PoCo language

• The full algorithm for the PoCo monitor

• Proof that the PoCo language is type safe

• Formal definitions of $\infty$-languages and trace equivalence

• Proof of five useful properties relating to PoCo obligations

The remainder of this dissertation is organized as follows: Chapter 2 discusses goals for the enforcement system, Chapter 3 discusses previous works related to the composition of obligation-based policies, Chapter 4 covers the design of the system, Chapter 5 describes the formal syntax and semantics of the PoCo language, Chapter 6 covers the structure of policies, Chapter 7 discusses the process of composing policies, Chapter 8 presents useful properties of PoCo’s obligations, and Chapter 9 concludes the paper.
Chapter 2: Goals

In order 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. This chapter provides the definitions and details for the listed goals.

2.1 Obligation-Type Support

Based on their time of execution, obligations can be partitioned into three categories: pre-, post-, and ongoing- [24, 25, 4]. 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 enforcement mechanism. If pre-obligations are unsupported, any

\footnotetext{Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.}
policy that wants to log attempted actions would be unenforceable. If post-obligations are unsupported, it would be impossible for the system to know the success or failure of actions and would limit its ability to perform obligations based on this status. Ongoing-obligations are required in order to perform any asynchronous actions such as tracking the time a user spends in a specific state or real-time resource monitoring.

2.2 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.

2.3 Obligations with Side Effects

Related work (e.g. [29]) requires obligations to be side-effect free, which makes some policies unenforceable. Any obligation that prompts the user for a decision or makes a call to a remote procedure causes side effects that cannot be undone; an enforcement mechanism that relies on rolling back obligations may be unable to manage such effectful obligations. This limitation makes such mechanisms undesirable for enforcing many policies.
2.4 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 [30]. In addition to ensuring policies have access to accurate state information, complete mediation allows policies of higher priority to ensure that they are not violated by lower priority policies.

2.5 Turing Completeness

Turing-complete policy-specification languages ensure expressiveness, at the cost of non-guaranteed enforcement termination (discussed in Section 4.5). Tools, like PoCo, that aim to provide a general-purpose policy-specification language, prioritize expressiveness and rely on the policy author to design their policies to terminate.

2.6 Conflict Resolution

The term conflict is used in numerous domains and, though the general idea remains the same (i.e., disagreements between policies), the understood meaning varies from domain to domain. In networking, overlapping rules on switches are conflicts (e.g. [11]); in databases, simultaneous updates to the same record are conflicts (e.g. [27]); and in distributed systems, choices made with incomplete or stale information are conflicts (e.g. [26]).

There are several types of conflicts that are possible when composing general-purpose, obligation-based policies. The policies may disagree on a permit/deny 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 these decisions enables resolution of this type of conflict.

When one policy’s obligation violates the rules of another policy, the resulting behavior of the composed policy that allows all obligations can be inconsistent with the behavior of each policy in isolation and may lead to unexpected behavior. While it may not be possible to comply with all policies completely, the decision as to which parts of each policy to enforce should not be random. There should be a well defined mechanism in place to allow a decision to be made on whether a given obligation should execute or not.

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.

2.7 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.
Chapter 3: Related Work

Composition of obligation policies is a long-standing research problem. This chapter describes the primary efforts in the area. Although there has been significant research on the composition of obligation policies, composition is a known challenge when considering obligations as aspects [32] and, to our knowledge, there has been no work that accomplishes all the goals outlined in Chapter 2. Table 3.1 provides an overview of the goals satisfied by existing policy-composition languages.

3.1 XACML

*Extensible Access Control Markup Language (XACML)* [5] 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.

The expressiveness of XACML policies is limited by a number of factors, including its stateless architecture [2], the inflexible manner in which obligations are combined, the inability to control when obligations are performed in relation to the access being granted or

---

1Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.
Table 3.1: Summary of existing policy-specification languages.

<table>
<thead>
<tr>
<th></th>
<th>Supports obligations that are</th>
<th>Supports resolving conflicts for obligations that are</th>
<th>Supports reacting to obligations that are</th>
<th>Supports custom composition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pre-</td>
<td>post-</td>
<td>effectful</td>
<td>Turing complete</td>
</tr>
<tr>
<td>XACML [5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>XACML Extensions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[1, 6, 18, 20, 17]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polymer [3]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Ponder [7, 33]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SPL [29, 28]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heimdall [12]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rei [14]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aspect Oriented [15]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

denied, and the lack of conflict detection. It is also difficult to determine its exact semantics due to the vague, modal verbs (i.e. “should”, “recommended”, “may”) used throughout the published specification. These terms are especially prevalent throughout the details of obligations making it difficult to determine exactly how they will be treated. In fact, the language implies that obligations may be ignored in some indeterminate circumstances [21].

Even with significant research extending XACML to overcome its limitations [1, 6, 18, 20, 17] (e.g., to add conflict resolution by requiring manual specification of which obligations conflict [20]), XACML is still lacking in some areas. Stateless policies are less expressive than stateful policies [10] and cannot express simple policies such as “disallow network-packet sends after file reads” [31]. Conflicts cannot be found and handled without upfront specification of exactly which obligations conflict with each other. The obligation-propagation mechanism and composition operators are still not customizable and there are many desirable compositions that can not be expressed.

3.2 Polymer

Polymer is an object-oriented policy specification language and run-time monitoring system [3] 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 as shown in Theorem 1.

3.3 Ponder

Ponder is a policy specification language that can be used to compose access-control
and general-purpose policies [7, 33]. 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.

3.4 SPL

Security Policy Language (SPL) is a policy specification language that enables users
to compose complex authorization policies by using policy combinators [29, 28] 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.

3.5 Heimdall

Heimdall uses compensatory actions in response to execution failures in obligations [12]. 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.
3.6 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 [14]. 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.

3.7 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 [15]. 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 [32].
Chapter 4: The PoCo Monitor Architecture\textsuperscript{12}

The enforcement mechanism used by the PoCo system is designed as a monitor located between a target (i.e. monitored) application and the underlying execution system that observes security-relevant actions (e.g., system or method calls) attempted by the target application and the security-relevant results of each of these actions, as shown in Figure 4.1. Hence, in the PoCo system, both actions and results can trigger responses from the monitor. The triggered response will depend on the specific logic of the enforced policies.

4.1 Monitor Operation

The PoCo system monitors all security-relevant events, both actions from the target application and results from the execution system, and broadcasts each event to every policy registered with the monitor for enforcement. Each policy receives the trigger event $e$ broadcast by the monitor and proposes an obligation to be executed before $e$ is processed (i.e executed or returned). This obligation may be empty, implement additional logic, or alter the triggering event to meet the policy’s goals. In the PoCo implementation, security-relevant events are automatically inferred from the implemented policy logic but can be further refined or expanded manually by the policy author. In practice, a defined list of security-relevant events is required because intercepting and broadcasting all events, even

\begin{footnotesize}
\begin{enumerate}
\item Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.
\item The PoCo language has been implemented as a compiler and enforcement system [9, 8]. Although this implementation is an important part of the overall PoCo project, this dissertation focuses on the parts of the PoCo project on which I worked.
\end{enumerate}
\end{footnotesize}
those not required for policy enforcement, could significantly limit the performance of the system.

The PoCo monitor is able to execute any number of obligations before ceding control to the target application by returning a result. Once the PoCo monitor has ceded control to the target application it cannot execute further obligations until a new security-relevant event is received from the application. Therefore, the PoCo monitor operates in a loop, with each iteration of the loop performing the following steps:

1. Receive security-relevant event $e$

2. Collect one obligation from each policy in response to $e$

3. While there are any obligations waiting to be processed

   (a) Select an obligation $o$

   (b) Collect and process policies’ votes on $o$

   (c) If $o$ is allowed, execute $o$

   (d) Collect any obligations triggered in response to $o$

4. If a new output event has been set, execute it or return it
5. Otherwise, execute or return the original input event

This series of steps is repeated until the resulting output event is a result which can then be used to cede control to the target application. With this method of operation, the monitor maintains control of until all approved obligations have executed, that is, the pool of pending obligations is exhausted.

4.2 Monitor Configuration

Before examining the PoCo language and enforcement system in detail, it is helpful to understand the high-level configuration options available to the policy author for controlling the composition of enforced policies. This section provides a quick overview of these options; a more detailed discussion of composition appears in Chapter 7. The policy author can supply up to three parameters to the monitor when specifying a composed policy.

The first parameter to the monitor is the list of base policies that will be used to construct the composed policy. This is the only required parameter when specifying a composed policy. There must be at least one policy to be enforced in this list.

The second parameter to the monitor is a *vote combinator* function that combines the votes from all policies on whether to allow an obligation to execute into a single permit/deny decision. The monitor uses this combined decision to determine which obligations to execute. PoCo supplies a default vote combinator and it is, therefore not required that the policy author provide one if the default behavior is acceptable to their use case. The default vote combinator is conjunction and, therefore, only allows obligations that are permitted by all enforced policies.

The final parameter is an *obligation scheduler* function which prioritizes obligations for execution. PoCo provides a default obligation scheduler and it is, therefore, not required that the policy author provide one if the default behavior is acceptable. The default obligation
scheduler arranges policies in the order in which they are defined. A custom obligation scheduler allows policy authors to prioritize obligations using other features, such as their complexity or based on the presence of certain actions in the obligation logic.

The PoCo monitor, therefore, can be thought of as a policy scheduler since it decides which policy obligations to execute and in what order to execute them. The monitor parameters discussed in this section allow this scheduling process to be customized to fit the needs of the composed policy being enforced.

4.3 Obligations

Throughout the existing literature on security policy specification and enforcement, there are many definitions of obligation [35, 3, 29, 7, 24, 25, 4, 13]. Generally, an obligation is one or more actions that the enforcement system executes under specified circumstances with specific timing in relation to events occurring in the target application.

When obligation-policy enforcement systems include dynamic conflict detection and resolution, the concept of an obligation being guaranteed to execute must become less strict. When a conflict involving an obligation occurs, the only options that are available to the system are to execute the offending obligation even though it conflicts (i.e., ignore the conflict), execute only the actions in the obligation that are not in conflict (i.e., non-atomic obligations), or skip execution of the obligation (i.e., obligation execution is not guaranteed).

Since one of this project's stated goals is to dynamically resolve conflicts between policies and atomic obligations, the only options that can be used when an obligation conflicts are to execute the obligation anyways or skip its execution entirely. Other works have referred to this definition of obligations as “suggestions” since they are not guaranteed to execute [3]. However, even XACML—which does not provide conflict resolution among its obligations—suffers from non-guaranteed obligation execution when an intermediate value in the policy/rule hierarchy doesn’t match the decision of the policy that generated the
obligation (even when the final result does) [23, Section 7.18]. In this work we have opted to use the term obligation over any alternative term (such as suggestion) with the understanding that the enforcement system is obligated to attempt execution of the obligation, but the end result may be that the obligation is not executed depending on the design of the composed policy.

4.4 Complete Mediation

Complete mediation—the ability for policies to monitor events executed by other policies’ obligations—is a desirable trait for the enforcement of obligation policies. By default, complete mediation is understood to be mean that each security-relevant event executed during an obligation can be responded to individually. This work refers to this design as event-by-event complete mediation. In event-by-event complete mediation, the monitor’s reaction to an event happens prior to the finished execution of the already executing obligation. At least one existing system has provided event-by-event complete mediation but without allowing for atomic obligations [3]. In fact, as Theorem 1 shows, it is not possible to have both event-by-event complete mediation and atomic obligations simultaneously.

Theorem 1 (Atomic Obligations and Complete Mediation). 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 by definition of event-by-event complete mediation, $m$ 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 $o$ before $o$ completes execution, then $o$ was not executed atomically.

Since event-by-event complete mediation is in direct opposition to one of the stated goals for this project (atomic obligations), PoCo enforces obligation-by-obligation complete
mediation, meaning that policies monitor and react to other policy’s atomic obligations in their entirety rather than each individual event within those obligations. Details on how this is accomplished can be found in Chapter 7.

4.5 Non-termination of Policy Enforcement

By including branching, looping and variables (Chapter 5), the PoCo language is designed as a Turing-complete language. This design choice introduces possible non-termination in the policy-enforcement code (e.g., policies may contain infinite loops). In addition to the standard non-termination introduced by a Turing-complete language, allowing policies to react to each other’s obligations introduces another path to non-termination; two policies may generate an infinite sequence of obligations in response to each other’s obligations. For example, if 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, an infinite sequence of file writes and database connections would be formed. Neither form of non-termination can be statically detected in general. This design prioritizes policy expressiveness over guaranteed enforcement termination.
Chapter 5: PoCo Language

This chapter defines the formal syntax and semantics for PoCo. These elements 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.

5.1 Syntax

Figure 5.1 lists all of the syntactic elements of the PoCo language. The PoCo language derives from the simply-typed lambda-calculus (STLC) with four base types: $\text{Int}$, $\text{Bool}$, $\text{String}$, and $\text{Unit}$; three composite types: arrow $\tau_1 \rightarrow \tau_2$, homogeneous list $\tau_{\text{List}}$, and reference type $\tau_{\text{Ref}}$; 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 of the language. $\text{Act}$ and $\text{Res}$ specify security-relevant actions and results. $\text{Event}$ is a sum type representing either an $\text{Act}$ or a $\text{Res}$. $\text{Obligation}$ is a sum type that holds a function-typed value taking either an $\text{Event}$ type to represent an $\text{onTrigger}$ or a $\text{Res}_{\text{List}}$ to represent an $\text{onObligation}$. The $\text{CFG}$ type is a 4-tuple consisting of nodes, edges,

\footnote{Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.}
the obligation represented by the CFG and the policy that proposed the obligation. 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 Chapter 6). Additionally, there are aliases for the two composition operators vote combinator (VC) and obligation scheduler (OS).

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 \( \tau \) Res wraps a security-relevant event of type \( \tau \). It is used in code that must unconditionally produce a term of type \( \tau \) (namely, in the dynamic semantics). The same distinction applies to \( \tau \) Act, Act, \( \tau \) Event and Event.

Introduction and elimination expressions are added for each type. It is worth mentioning that, although both \( \text{call}(e_1, e_2) \) and \( \text{invoke}(e_1, e_2) \) can be used to call functions, the two expressions are designed for different purposes. The expression \( \text{call}(e_1, e_2) \) 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, \( \text{invoke}(e_1, e_2) \) 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 \textit{String} that specifies a monitored function’s name.

Label elements are added to facilitate the obligation property proofs in Chapter 8. The two elements, \textit{begin} and \textit{end}, 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.
Types:

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

| Act | \equiv (\text{name} : \text{String} \times \text{arg} : \text{TypedVal}) |
| Res | \equiv (\text{act} : \text{Act} \times \text{result} : \tau) |
| Event | \equiv (\text{act} : \text{Act} + \text{res} : \tau \text{ Res}) |
| Event | \equiv (\text{act} : \text{Act} + \text{res} : \text{Res}) |

Obligation \equiv (\text{ot} : (\text{evt} : \text{Event} \times \text{onTrig} : \text{Event} \rightarrow \text{unit}) + \text{oo} : (\text{rt} : \text{ResList} \times \text{onOblig} : \text{ResList} \rightarrow \text{unit}))

CFG \equiv (\text{nodes} : \text{ActList} \times \text{edges} : (\text{start} : \text{Act} \times \text{end} : \text{ActList}) \rightarrow \text{obligation} : \text{Obligation})

Pol \equiv (\text{name} : \text{String} \times \text{onTrigger} : \text{Event} \rightarrow \text{unit} \times \text{onObligation} : \text{ResList} \rightarrow \text{unit} \times \text{vote} : \text{CFG} \rightarrow \text{Bool})

OS \equiv (\text{pol} : \text{Pol} \times \text{cfg} : \text{CFGList}) \rightarrow (\text{pol} : \text{Pol} \times \text{cfg} : \text{CFGList})

VC \equiv (\text{name} : \text{String} \times \text{vote} : \text{BoolList}) \rightarrow \text{Bool}

Option \equiv (\text{some} : \tau \rightarrow \text{none} : \text{Unit})

Values:

\[ b ::= \text{true} \mid \text{false} \]

\[ f ::= \text{fun} \ x_1(x_2 : \tau_1) : \tau_2 = e \]

\[ v ::= b \mid s \mid n \mid f \mid \text{unit} \mid v_1 : v_2 \mid [] : \tau\text{List} \mid \ell \]

\[ | \text{inv} \ v : \tau \mid (\ell_1 = v_1, \ldots, \ell_n = v_n) \]

\[ | \text{makeTypedVal}(\tau, v) \]

Expressions:

\[ e ::= v \mid x \mid e_1 : e_2 \mid (e_1 : e_2) \mid e_1 \lor e_2 \mid e_1 \land e_2 \mid e_1 \equiv e_2 \mid \neg e_1 \mid \text{ref} \ e \mid !e \mid e_1 := e_2 \mid e_1 + e_2 \mid \text{while}(e_1) \{ e_2 \} \mid \text{let} \ x = e_1 \text{ in } e_2 \text{ end} \mid \text{in}_\ell \ e : \tau \mid \text{head}(e) \mid \text{monitor}(\tau, e) \mid \text{if} \ e_1 \text{ then } e_2 \text{ else } e_3 \mid (\ell_1 = e_1, \ldots, \ell_n = e_n) \mid e.\ell_i \mid \text{tail}(e) \mid (\text{case } e \text{ of } \ell_1 \ x_1 \Rightarrow e_1 \mid \ldots \mid \ell_n \ x_n \Rightarrow e_n) \mid \text{setOutput}(e) \mid \text{getOutput}() \mid \text{outputNotSet}() \mid \text{getRT}() \mid \text{call}(e_1, e_2) \mid \text{invoke}(e_1, e_2) \mid \{ e \}_{(e)} \mid \text{makeCFG}(e) \mid \text{makeTypedVal}(\tau, e_1) \mid \text{tryCast}(\tau, e_1) \mid \text{empty}(e) \]

\[ \text{act}(e_1, e_2) \equiv (\text{name} = e_1, \text{arg} = e_2) \mid \text{res}(e_1, e_2) \equiv (\text{act} = e_1, \text{result} = e_2) \]

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

Monitors \( R ::= (F, \text{pols, os, vc}) \), where \( \text{pols, os, vc} \) are values

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

Configurations:

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

Labels \( \text{label} ::= \text{begin}_{s(v)} \mid \text{end}_{s(v)} : v_2 \)

\textbf{Figure 5.1:} Formal syntax for PoCo
5.2 Static Semantics

This section presents the full static semantics of the PoCo language. The static semantics are defined using the judgment form $\Lambda, \Gamma \vdash e : \tau$. In this judgement form, the context $\Gamma$ maps variables to their types while $\Lambda$ maps memory locations to their types. It can be said that “under the contexts $\Lambda$ and $\Gamma$, the expression $e$ is of type $\tau$” if and only if the judgment $\Lambda, \Gamma \vdash e : \tau$ is derivable by the rules presented in this section. The remainder of this section presents the the static semantic rules divided by their general function.

The following four rules define the values for the four base types included in the PoCo language. These base types consist of Integer, Boolean, String, and Unit.

$$
\Lambda, \Gamma \vdash e : \tau
$$

- $\Lambda, \Gamma \vdash n : \text{Int}$ (intVal)
- $\Lambda, \Gamma \vdash b : \text{Bool}$ (boolVal)
- $\Lambda, \Gamma \vdash s : \text{String}$ (stringVal)
- $\Lambda, \Gamma \vdash \text{unit} : \text{Unit}$ (unitVal)

The following six rules are used for managing memory access and variable declarations. The rules include creating and accessing references as well as declaring let environments.

$$
\Lambda, \Gamma \vdash e : \tau
$$

- $\Lambda, \Gamma \vdash \text{var} : \{x : \tau\} \vdash x : \tau$ (var)
- $\Lambda, \Gamma \vdash \text{ref} : \ell : \tau$ (location)
- $\Lambda, \Gamma \vdash e : \tau \text{Ref}$ (accessRef)
- $\Lambda, \Gamma \vdash \text{createRef} e : \tau$ (createRef)
- $\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2$ (let)
- $\Lambda, \Gamma \vdash \text{assignment} e_1 : = e_2 : \text{Unit}$ (assignment)
The following three rules are for the elimination forms of Boolean typed expressions. These elimination forms consist of conjunction, disjunction and negation.

\[ \Lambda, \Gamma \vdash e : \tau \]

\[ \frac{\Lambda, \Gamma \vdash e_1 : \text{Bool} \quad \Lambda, \Gamma \vdash e_2 : \text{Bool}}{\Lambda, \Gamma \vdash e_1 \land e_2 : \text{Bool}} \] (con) \quad \frac{\Lambda, \Gamma \vdash e_1 : \text{Bool} \quad \Lambda, \Gamma \vdash e_2 : \text{Bool}}{\Lambda, \Gamma \vdash e_1 \lor e_2 : \text{Bool}} \] (dis)

\[ \frac{\Lambda, \Gamma \vdash e : \text{Bool}}{\Lambda, \Gamma \vdash \neg e : \text{Bool}} \] (negation)

The equality rule defines the equality operator by limiting its scope to only Integers, Booleans, and Strings.

\[ \Lambda, \Gamma \vdash e : \tau \]

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

The add rule is used as the elimination form of Integer typed expressions. Obviously, there are other expressions that could have been implemented as elimination forms for Integers, but they were not needed for the examples included in this dissertation and did not add anything original to the overall solution.

\[ \Lambda, \Gamma \vdash e : \tau \]

\[ \frac{\Lambda, \Gamma \vdash e_1 : \text{Int} \quad \Lambda, \Gamma \vdash e_2 : \text{Int}}{\Lambda, \Gamma \vdash e_1 + e_2 : \text{Int}} \] (add)

The following six rules are used to work with lists, including retrieving the head or tail of the list, testing if the list is empty, the empty list value, and adding values to lists.

\[ \Lambda, \Gamma \vdash e : \tau \]

\[ \frac{\Lambda, \Gamma \vdash e : \tau_{\text{List}}}{\Lambda, \Gamma \vdash \text{head}(e) : \tau \text{ Option}} \] (head) \quad \frac{\Lambda, \Gamma \vdash e : \tau_{\text{List}}}{\Lambda, \Gamma \vdash \text{tail}(e) : \tau_{\text{List}}} \] (tail)
The following three rules define functions and how they are called. The rule call is used for standard function calls and specifies the first parameter as a function type. The rule invoke is used for PoCo to execute valid actions 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).

\[
\begin{array}{l}
\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash \text{call}(e_1, e_2) : \tau_2} \quad \text{(call)}
\end{array}
\]

\[
\begin{array}{l}
\frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_2}{\Lambda, \Gamma \vdash \text{invoke}(e_1, e_2) : \text{TypedVal Option}} \quad \text{(invoke)}
\end{array}
\]

The rules variant and case define the labelled sum algebraic type and the case statement that is its elimination form.

\[
\begin{array}{l}
\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash e_1 : \tau_1 \quad i \in \{1, \ldots, n\}}
\end{array}
\]

\[
\frac{\Lambda, \Gamma \vdash \text{in}_i(e_1) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)}{\Lambda, \Gamma \vdash e(e_1) : \tau} \quad \text{(variant)}
\]

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

The record and projection rules define the labelled product type and its elimination form, projection.

\[
\begin{array}{l}
\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash e : \tau}
\end{array}
\]
The following three rules define the control flow operations that are available in the 
Poco language (while, sequence, if).

\[
\begin{align*}
\Lambda, \Gamma \vdash e : \tau \\
\Lambda, \Gamma \vdash e_1 : \text{Bool} & \quad \Lambda, \Gamma \vdash e_2 : \tau \\
\Lambda, \Gamma \vdash \text{while}(e_1) \{e_2\} : \text{Boo}l & \quad \Lambda, \Gamma \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \vdash e_2 : \tau_2 \\
\Lambda, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau &
\end{align*}
\]

The following seven rules are used to handle functionality that is very specific to Poco.
The rule \textit{makeCFG} specifies the type of the function used to create CFGs from Obligation 
specifications. The rule \textit{getRT} is used to retrieve the result trace from the last obligation 
executed. The rule \textit{endLabel} is an internal rule used to specify the completion of a function 
call that should be added to the current trace. Additionally, there are three rules around 
working with the output event and a rule used to specify the internal “monitor” function 
that is called on each trigger event.

\[
\begin{align*}
\Lambda, \Gamma \vdash e : \tau \\
\Lambda, \Gamma \vdash e : \text{Obligation} & \quad \Lambda, \Gamma \vdash \text{makeCFG}(e) : \text{CFG} \\
\Lambda, \Gamma \vdash \text{getRT}() : \text{ResList} & \quad \Lambda, \Gamma \vdash \text{setOutput}(e) : \text{Boo}l \\
\Lambda, \Gamma \vdash \{ e \}_{s(v)} : \tau & \quad \Lambda, \Gamma \vdash \text{getOutput}() : \text{Event Option} \\
\Lambda, \Gamma \vdash \text{outputNotSet}() : \text{Boo}l
\end{align*}
\]
\[ \Lambda, \Gamma \vdash e : (\text{evt} : \tau \text{ Event} \times \text{pols} : \text{Pol List} \times \text{os} : \text{OS} \times \text{vc} : \text{VC}) \quad (\text{monitor}) \]

The final two rules are used for handling dynamically typed values (of TypedVal type). These values are used when handling obligation execution where the return type cannot be statically determined.

\[ \Lambda, \Gamma \vdash e : \tau \]

\[ \Lambda, \Gamma \vdash \text{makeTypedVal}(\tau, e) : \text{TypedVal} \quad (\text{makeTypedVal}) \]

\[ \Lambda, \Gamma \vdash \text{tryCast}(\tau, e) : \tau \text{ Option} \quad (\text{tryCast}) \]

An additional judgement form \( \Lambda, \Gamma \vdash e \text{ ok} \) is defined to capture the idea of user-level static semantics. In other words, it excludes rules that are meant to only be used by internal enforcement system functions. The rules for \( \Lambda, \Gamma \vdash e \text{ ok} \) are identical to the rules for \( \Lambda, \Gamma \vdash e : \tau \) (that is, \( \Lambda, \Gamma \vdash e \text{ ok} \iff \exists \tau. \Lambda, \Gamma \vdash e : \tau \)), except there are no equivalent rules for \text{label} and \text{monitor}. 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 built-in monitor function (\( \text{monitor}(\tau, e) \)). The dynamic semantics will allow an ok program to step to a non-ok (but still well-typed) program. For example, it may be the case that \( (C, \text{invoke}("exit", v)) \rightarrow (C, \{\cdots\}_{\text{exit}(v)}) \).

A list of monitored functions \( F \) is \text{ok} if and only if for each pair \((s_i, f_i)\), \( s_i \) is of type \text{String} and \( f_i \) is of type \( \tau_1 \rightarrow \tau_2 \), for some \( \tau_1 \) and \( \tau_2 \). It is also assume that each monitored function name (i.e., each \( s_i \)) is unique in \( F \). Additionally, a monitor configuration \( R \) is \text{ok} if and only if, \( F \) is \text{ok} and the values \( \text{pols}, \text{os} \) and \( \text{vc} \) are of types \( \text{Pol List}, \text{OS}, \) and \( \text{VC} \), respectively.

**Assumption 1.** \( F = (s_n, f_n), F' \Rightarrow s_n \notin \text{dom}(F') \)
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 \( \tau_{\text{out}} \), 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.
5.3 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 is the result of the current step.

The semantics assume the existence of a procedure $\text{makeCFG}_\alpha$ which, given an arbitrary value $v$ of type $\text{Obligation}$, computes a value $g$ of type $\text{CFG}$ such that $g$ represents the control-flow graph of $v$. Formally:

**Assumption 2.** $\forall v \forall \Lambda \; \Lambda, \bullet \vdash v : \text{Obligation} \Rightarrow \Lambda, \bullet \vdash \text{makeCFG}_\alpha(v) : \text{CFG}$

The first nine rules handle the Boolean operations of conjunction, disjunction and negation. Each operation has rules to allow further reduction of the expressions as well as rules for handling the values of true and false.

\[
\begin{align*}
(C, e) \xrightarrow{\text{label}_1, \ldots, \text{label}_n} & (C', e') \\
(C, e_1) & \rightarrow (C', e'_1) \quad \text{(andE)} \\
(C, e_1 \land e_2) & \rightarrow (C', e'_1 \land e_2) \\
(C, \text{false} \land e_2) & \rightarrow (C, \text{false}) \quad \text{(andFalse)} \\
(C, \text{true} \lor e_2) & \rightarrow (C, \text{true}) \quad \text{(orTrue)} \\
(C, e) & \rightarrow (C', e') \quad \text{(notE)} \\
(C, \neg e) & \rightarrow (C', \neg e') \\
(C, \neg \text{true}) & \rightarrow (C, \text{false}) \quad \text{(notTrue)} \\
(C, \text{true}) & \rightarrow (C, e_2) \\
(C, e_1 \land e_2) & \rightarrow (C', e'_1 \land e_2) \\
(C, \text{true} \land e_2) & \rightarrow (C, \text{false}) \\
(C, \text{false} \lor e_2) & \rightarrow (C, e_2) \\
(C, \neg \text{false}) & \rightarrow (C, \text{true}) \quad \text{(notFalse)} \\
\end{align*}
\]
The following three rules are used to handle the integer add operation. As discussed in Section 5.2, this is the only integer operation implemented in this formalization of the PoCo language.

\[
(C, e) \xrightarrow{label_1, \ldots, label_n} (C', e')
\]

\[
(C, e_1) \rightarrow (C', e_1') \quad \text{(addE1)}
\]

\[
(C, e_2) \rightarrow (C', e_2') \quad \text{(addE2)}
\]

\[
(C, n_1 + e_2) \rightarrow (C', n_1 + e_2') \quad \text{(addValue)}
\]

The following eight rules handle the equality operator. Note that separate rules are needed for each type that equality is allowed to be applied to. PoCo implements equality for the types Integer, Boolean, and String.

\[
(C, e) \xrightarrow{label_1, \ldots, label_n} (C', e')
\]

\[
(C, e_1) \rightarrow (C', e_1') \quad \text{(eqE1)}
\]

\[
(C, e_2) \rightarrow (C', e_2') \quad \text{(eqE2)}
\]

\[
(C, v_1 == e_2) \rightarrow (C', v_1 == e_2') \quad \text{(eqE1)}
\]

\[
(C, n_1 == n_2) \rightarrow (C, true) \quad \text{(eqIntTrue)}
\]

\[
(C, n_1 == n_2) \rightarrow (C, false) \quad \text{(eqIntFalse)}
\]

\[
(C, true == b_2) \rightarrow (C, b_2) \quad \text{(eqBoolTrue)}
\]

\[
(C, false == b_2) \rightarrow (C, \neg b_2) \quad \text{(eqBoolFalse)}
\]

\[
(C, s_1 == s_2) \rightarrow (C, true) \quad \text{(eqStrTrue)}
\]

\[
(C, s_1 == s_2) \rightarrow (C, false) \quad \text{(eqStrFalse)}
\]

The following six rules handle program control flow by implementing standard if/then/else statements, statement sequencing and while loops. The if expression has a rule for further reducing the condition expression as well as rules to handle when the condition is true and when the condition is false. Sequencing has a rule to further reduce the first expression as well as a rule for when the first expression is already a a value. The while expression has only a single rule that expands the while recursively into an if expression and a while expression.
The following nine rules are for the management of variables and memory references. They include creating, updating and accessing memory references as well as let environments.

\[
(C, e) \xrightarrow{label_1 \ldots label_n} (C', e')
\]

\[
(C, if \ true \ then \ e_2 \ else \ e_3) \rightarrow (C, e_2)
\]

\[
(C, if \ false \ then \ e_2 \ else \ e_3) \rightarrow (C, e_3)
\]

\[
(C, e_1) \rightarrow (C', e'_1)
\]

\[
(C, if \ e_1 \ then \ e_2 \ else \ e_3) \rightarrow (C', if \ e'_1 \ then \ e_2 \ else \ e_3)
\]

\[
(C, e_1) \rightarrow (C', e'_1)
\]

\[
(C, e_1; e_2) \rightarrow (C', e'_1; e_2)
\]

\[
(C, v_1; e_2) \rightarrow (C, e_2)
\]

\[
(C, ref \ e) \rightarrow (C', ref \ e')
\]

\[
(C, !e) \rightarrow (C', !e')
\]

\[
\ell \notin dom(M)
(M, \ldots, ref \ v) \rightarrow ((M \cup \{(\ell, v), \ldots, \ell\})
\]

\[
(M \cup \{(\ell, v), \ldots, \ell\}) \rightarrow ((M \cup \{(\ell, v), \ldots, \ell\})
\]

\[
(C, e_1 := e_2) \rightarrow (C', e'_1 := e_2)
\]

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

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

\[
((M \cup \{(\ell, v), \ldots, \ell := v\}) \rightarrow ((M \cup \{(\ell, v'), \ldots, unit\})
\]

\[
(C, let \ x = e_1 \ in \ e_2 \ end) \rightarrow (C', let \ x = e'_1 \ in \ e_2 \ end)
\]

\[
(C, let \ x = v \ in \ e_2 \ end) \rightarrow (C, [v/x]e_2)
\]
The following rules are used to handle lists. They include the head and tail operations, the empty test and the operations used to add values to lists. These rules include rules that handle further reduction of expressions as well as operations on values for each operation.

\[
(C, e) \xrightarrow{label_1, \ldots, label_n} (C', e')
\]

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

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

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

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

\[
(C, \text{head}([] : \tau_{List})) \rightarrow (C, \text{in}_{\text{none}}(\text{unit}) : \tau \text{ Option}) \quad \text{(listHeadNil)}
\]

\[
(C, \text{tail}([] : \tau_{List})) \rightarrow (C, []) : \tau_{List} \quad \text{(listTailNil)}
\]

\[
(C, \text{tail}(v_1 :: v_2)) \rightarrow (C, v_2) \quad \text{(listTailCons)}
\]

\[
(C, \text{head}(v_1 :: \cdots :: [] : \tau_{List})) \rightarrow (C, \text{in}_{\text{some}}(v_1) : \tau \text{ Option}) \quad \text{(listHeadCons)}
\]

\[
(C, \text{empty}([] : \tau_{List})) \rightarrow (C, \text{true}) \quad \text{(listEmptyNil)}
\]

\[
(C, \text{empty}(v_1 :: v_2)) \rightarrow (C, \text{false}) \quad \text{(listEmptyCons)}
\]

\[
(C, e_2) \rightarrow (C', e'_2) \quad \text{(listAppendE2)}
\]

\[
(C, v_1 @ e_2) \rightarrow (C', v_1 @ e'_2) \quad \text{(listAppend)}
\]

\[
(C, v_1 :: v_2) @ v_3 \rightarrow (C, v_1 :: (v_2 @ v_3)) \quad \text{(appendCons)}
\]

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

\[
(C, e_2) \rightarrow (C', e'_2) \quad \text{(listPrependE2)}
\]

The following three rules are for creating and accessing values from records. There is a rule for reducing sub-expressions in a record’s declaration as well as a rule to handle record evaluation and a rule for accessing values from a fully reduced record.

\[
(C, e) \xrightarrow{label_1, \ldots, label_n} (C', e')
\]

\[
\forall j(1 \leq j \leq i), e_j = v_j \quad (C, e_i) \rightarrow (C', e'_i) \quad i \in \{1, \ldots, n\} \quad \text{(recordE)}
\]

\[
(C, (l_1 = e_1, \ldots, l_n = e_n)) \rightarrow (C', (l_1 = e'_1, \ldots, l_n = e_n)) \quad \text{(recordE)}
\]
\[
\begin{align*}
(C, e) & \rightarrow (C', e') \\
(C, e.\ell_i) & \rightarrow (C', e'.\ell_i) \quad \text{(projectionE)} \\
i \in \{1, \ldots, n\} & \\
(C, (\ell_1 = v_1, \ldots, \ell_n = v_n).\ell_i) & \rightarrow (C, v_i) \quad \text{(projectionV)}
\end{align*}
\]

The following three rules are for creating and accessing values from variants. These rules are very similar to the rules for records, but the elimination form for variants is the case expression rather than projection.

\[
(C, e) \xrightarrow{\text{label}_1, \ldots, \text{label}_n} (C', e')
\]

\[
(C, e_i) \rightarrow (C', e_i') \quad \text{(variantE)}
\]

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

The following rules are used for handling function calls. There are four rules for invoke (calls function based on function signature). Additionally, there are seven rules to handle direct function calls. Specifically, there are five rules that handle specific cases of function call. Each of these rules add begin and end labels to the execution trace and executes the function call based on the situation. Specifically, 1) for a security-irrelevant function, PoCo directly executes the function. 2) for a security-relevant function call originating from the target, PoCo will invoke the monitor to resolve the event; 3) for onTrigger and onObligation calls, PoCo resets the current result trace before evaluation; 4) for a security-relevant function call originating from an obligation, PoCo will append the result of the call to the result trace.

\[
(C, e) \xrightarrow{\text{label}_1, \ldots, \text{label}_n} (C', e')
\]

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

\[
(C, \text{invoke}(e_1, e_2)) \rightarrow (C', \text{invoke}(e_1', e_2)) \quad \text{(invokeE1)}
\]
\[(C, e_2) \rightarrow (C', e'_2)\] (invokeE2)

\[(C, invoke(s_1, e_2)) \rightarrow (C', invite(s_1, e'_2))\] (invokeE2)

\[(s_1, fun x_1(x_2 : \tau_1) : \tau_2 = e) \in F \rightarrow v_2 = makeTypedVal(\tau_1, v'_2)\] (invokeValueExistsOk)

\[((M, (F, \ldots), \ldots), invoke(s_1, v_2)) \rightarrow ((M, (F, \ldots), \ldots), in\_some(makeTypedVal(\tau_2, call(fun x_1(x_2 : \tau_1) : \tau_2\{e\}, v'_2))) : TypedVal Option)\]

\[(s_1, fun x_1(x_2 : \tau_1) : \tau_2 = e) \in F \rightarrow v_2 = makeTypedVal(\tau_3, v'_2) \quad \tau_1 \neq \tau_3\] (invokeValueExistsBad)

\[((M, (F, \ldots), \ldots), invoke(s_1, v_2)) \rightarrow ((M, (F, \ldots), \ldots), in\_none(unit) : TypedVal Option)\]

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

\[((M, (F, \ldots), \ldots), invoke(s_1, v_2)) \rightarrow ((M, (F, \ldots), \ldots), in\_none(unit) : TypedVal Option)\]

\[(C, e_1) \rightarrow (C', e'_1)\] (callE1)

\[(C, call(e_1, e_2)) \rightarrow (C', call(e'_1, e'_2))\] (callE2)

\[(s, fun x_1(x_2 : \tau_1) : \tau_2 = e) \in F \rightarrow f = (fun x_1(x_2 : \tau_1) : \tau_2 = e)\] (callFromObligation)

\[((M, (F, \ldots), \ldots), true, rt, \ldots), call(f, v)) \begin{align*}
\text{begin}_{e_1(v)}: \text{begin}_{append\_Rest}(e_2(v))
\end{align*}\]

\[((M, (F, \ldots), \ldots), true, rt \@ res(onTrigger(s, makeTypedVal(\tau_1, v)), makeTypedVal(\tau_2, [f/x_1, v/x_2]e)) :: [] : Res\_List, \ldots),
\{[f/x_1, v/x_2]e\}_{e_1(v)}\]

\[f \notin \text{range}(F)\]

\[\forall \text{pol} \in \text{pols} \quad (f \neq \text{pol.onTrigger} \land f \neq \text{pol.onObligation})\]

\[f = (fun x_1(x_2 : \tau_1) : \tau_2 = e)\] (callNonMonitoredFunction)

\[f = \begin{align*}
(M, (F, \text{pols}, \ldots), \ldots), call(f, v)) & \quad \begin{align*}
\text{begin}_{f(v)}
\end{align*}
\end{align*}\]

\[\begin{align*}
(M, (F, \text{pols}, \ldots), \ldots), \text{call}(f, v) & \quad \begin{align*}
\text{begin}_{f(v)}
\end{align*}
\end{align*}\]

\[\begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\end{align*}
\end{align*}\]

\[\text{f}_1 = \begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\end{align*}
\end{align*}\]

\[\begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\text{name} = s, \text{onTrigger} = f_1, \text{onObligation} = f_2, \text{vote} = f_3) \in \text{pols} & \quad \begin{align*}
\end{align*}
\end{align*}\]
\[(s, f) \in F \quad f = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e) \]

\[
\begin{align*}
(M, (F, \text{pols, os, vc}), \text{false, rt, out, } \tau_{old}), \text{call}(f, v) & \xrightarrow{\text{begin}_{\text{eval}}} \\
(M, (F, \text{pols, os, vc}), \text{false, rt, in}_{\text{none(unit)}} : \tau_2 \text{ Event Option, } \tau_2), e_{\text{procEvt}}
\end{align*}
\]

where \( e_{\text{procEvt}} = \)

\[
\{ \text{let aux} = (\text{fun aux(event : } \tau_2 \text{ Event) : } \tau_2 \text{ Res} = \text{case event of} \text{ act a} \Rightarrow \text{ case invoke(a.name, a.arg) of} \text{ some r} \Rightarrow \text{ case tryCast(} \tau_2, r_1) \text{ of} \text{ some v_1} \Rightarrow \text{ let action}_\text{out} = \text{in_res(res(a, v_1) : } \tau_2 \text{ Event in} \text{ let mon}_\text{out} = \text{monitor(} \tau_2, \text{ (} \text{ev} = \text{action}_\text{out, pols} = \text{pols, os} = \text{os, vc} = \text{vc}) \text{ in} \text{ call(aux, mon}_\text{out}) \text{ end} \text{ end} \text{ | none u}_1 \Rightarrow \text{call(aux, event) \text{ | none u}_2 \Rightarrow \text{call(aux, event) \text{ | res r}_2 \Rightarrow r_2) \text{ in call(aux, in}_{\text{act (s, v) : } \tau_2 \text{ Event).result end})}_s(v) \}
\]

The \( e_{\text{procEvt}} \) expression used in the \text{callFromApplication} rule processes a single security-relevant event from the target application. This expression recursively calls the monitor function until a result is returned as the output event. When a result is returned, it is returned to the target application.

The following rules implement functionality specific to the PoCo monitor including handling of the output event, retrieval of the current result trace, making CFGs from obligations, handling execution trace labels and execution of the monitor expression (including setting the flag \text{inOb} to true before evaluation to indicate the current executing context).

\[
\begin{align*}
(C, e) & \xrightarrow{\text{label}_1, \ldots, \text{label}_n} (C', e') \\
(C, e) & \xrightarrow{\text{setOutput(E)}} (C', \text{setOutput}(e')) \\
(out = \text{in}_{\text{some(e) : } \tau} \text{ Event Option}) & \xrightarrow{\text{setOutputSet}} ((\ldots, \text{out, } \tau_{out}), \text{setOutput}(v)) \xrightarrow{\text{setOutputSet}} ((\ldots, \text{out, } \tau_{out}, \text{false})
\end{align*}
\]
\[
\begin{align*}
\text{(setOutputNotSetAct)} & : ((..., \text{in\_not}(\text{unit}) : \tau_{out} \text{ Event Option}, \tau_{out}), \text{outputNotSet}(\text{in\_act}(v) : \text{Event})) \rightarrow (..., \text{in\_not}(\text{unit}) : \tau_{out} \text{ Event Option}, \tau_{out}, \text{true}) \\
\text{(setOutputNotResGood)} & : ((..., \text{in\_not}(\text{unit}) : \tau_{out} \text{ Event Option}, \tau_{out}), \text{setOutput}(\text{in\_res}(\text{res}(v_{1, \text{make\_Typed\_Val}((\tau_{out}, v_{2}))) : \text{Event})) \rightarrow ((..., \text{in\_not}(\text{unit}) : \tau_{out} \text{ Event Option}, \tau_{out}, \text{true}) \\
\text{(setOutputNotResBad)} & : (\tau' \neq \tau_{out}) \rightarrow (..., \text{in\_not}(\text{unit}) : \tau_{out} \text{ Event Option}) \\
\text{(outputNotTrue)} & : ((..., \text{out}, \tau_{out}), \text{outputNotSet}()) \rightarrow ((..., \text{out}, \tau_{out}), \text{true}) \\
\text{(outputNotFalse)} & : ((..., \text{out}, \tau_{out}), \text{outputNotSet}()) \rightarrow ((..., \text{out}, \tau_{out}), \text{false}) \\
\text{(outputSomeAct)} & : ((..., \text{out}, \tau_{out}), \text{getOutput}()) \rightarrow ((..., \text{out}, \tau_{out}), \text{true}) \\
\text{(outputSomeRes)} & : ((..., \text{out}, \tau_{out}), \text{getOutput}()) \rightarrow ((..., \text{out}, \tau_{out}), \text{true}) \\
\text{(outputNone)} & : ((..., \text{out}, \tau_{out}), \text{getOutput}()) \rightarrow ((..., \text{out}, \tau_{out}), \text{true}) \\
\text{(getRTVal)} & : ((..., \text{rt}, \text{out}, \tau_{out}), \text{get\_RT}()) \rightarrow ((..., \text{rt}, \text{out}, \tau_{out}), \text{rt}) \\
\text{(makeCFGValue)} & : (C, \text{make\_CFG}(e)) \rightarrow (C', \text{make\_CFG}(e')) \\
\text{(make\_CFG)} & : g = \text{make\_CFG}_{\alpha}(v) \\
\text{(end\_Label\_E)} & : (C, e) \rightarrow (C', e') \\
\text{(end\_Label\_Value)} & : s \neq \text{"monitor"} \\
\text{((M, R, false, ...), \{ v_{1} \}_{s_{(v_{1})}}) \rightarrow \text{end}_{\tau_{(v_{1})} : v_{1}} (\text{((M, R, false, ...), v_{1})}}
\end{align*}
\]
The following four rules handle the creation and use of dynamically typed valued (TypedVal type) by implementing the makeTypedVal and tryCast functionality.

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

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

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

\[
(C, \text{tryCast}(\tau_1, \text{makeTypedVal}(\tau_2, v))) \rightarrow (C, \text{in} \text{none}(\text{unit}) : \tau_1 \text{ Option}) \quad \text{(tryCastVBad)}
\]

### 5.4 Type Safety

The PoCo language is proven type safe via the standard Preservation and Progress Lemmas [34]. Type safety of the PoCo language guarantees that well-typed 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 A.

**Theorem 2** (Type-safety).

\[
(C, e) : \tau \land (C, e) \rightarrow^* (C', e') \Rightarrow

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

(\exists v : e' = v \lor \exists C'', e'' : (C', e') \rightarrow (C'', e'')).
\]
In addition to guaranteeing that a language cannot get stuck, type safety is important to languages used for security-policy enforcement because of the additional guarantees it imposes. Type safe languages prevent operations from being applied to the incorrect types and can therefore prevent some logic errors that may be introduced by the policy author. Type safety also guarantees memory safety which protects against both accidental and willful use of values in memory as the incorrect type.
Chapter 6: PoCo Policies

Policies written in the PoCo language are designed as modules of logic that propose specific obligations based on events that they consider security-relevant. Each security-relevant action the target application attempts and any corresponding security-relevant results that the executing system returns are broadcast to all policies that have been registered with the PoCo monitor. This allows each policy to see all events that any policy considers security relevant rather than just the ones it specifically cares about. This approach allows all policies to participate whenever the monitor is activated.

In this chapter, five policies are used to highlight the unique design of PoCo policies. These five policies are defined as:

- $P_{file}$ prevents the target application from opening the secret.txt file.
- $P_{postlog}$ requires that each file-open action be logged after the action completes.
- $P_{prelog}$ requires that each file-open action be logged before the action completes.
- $P_{confirm}$ requires that each file-open action attempted by the target be confirmed by the user through a pop-up window.
- $P_{time}$ disallows popups unless at least 100 seconds have passed since the target application last showed a popup.

Together, these example policies can be used to illustrate the core features of PoCo’s policy enforcement. Policies in PoCo are specified with a tuple consisting of three specific functions.

\(^1\)Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.
The remainder of this chapter will discuss this policy structure as well as the core features of PoCo policies.

6.1 The onTrigger Policy Function

The first element in the specification of a PoCo policy is called the onTrigger function. The onTrigger function is used by the policy to specify an obligation that executes prior to processing a security-relevant input event. Policies that need to react to actions attempted by the application or results returned from the system will specify a non-empty onTrigger function. Since onTrigger obligations can be used to respond to both actions and results, they can implement pre-, post-, and ongoing- obligations (see Theorem 7). These obligations are atomic; every onTrigger function implicitly obtains a mutex lock that is released upon completion of the obligation’s execution.

The onTrigger function accepts a trigger event e as an input parameter and may specify arbitrary logic to run before e is either executed or returned. onTrigger obligations may also optionally set an output event. An output event is the PoCo monitor’s final response to a trigger event. Eventually, the PoCo monitor must have a way to relinquish control and enable the application or system to continue executing. If none of the executed obligations specify an output event, the PoCo monitor will cede control with the default of allowing the trigger event to be output (and consequently executed if it is an action or returned if it is a result).

For example, P_file’s onTrigger function examines the input event e and if e is the action `fopen(secret.txt)` then it sets the exit action as the output event. This indicates that the monitor should cede control to the underlying system to execute the exit action. When e is not the action `fopen(secret.txt)`, P_file does nothing, thus allowing events that are not relevant to P_file to continue executing normally. This particular obligation does not
inject additional events prior to the monitor ceding control. Hence, $P_{file}$’s \texttt{onTrigger} is
defined as follows.

\begin{verbatim}
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 )
\end{verbatim}

Output events are treated specially because the monitor must reach agreement regarding how the control is transferred to the target application or underlying system. Therefore, a primary objectives of any system that enforces composed run-time policies must be to determine the singular output event for each trigger event. When an output event is an action, the monitor cedes control to the underlying system. When the output event is a result the monitor cedes control to the application. Prior work has defined monitors that operate in this way, interposing between applications and executing systems and responding to trigger events with output events [19].

As seen in $P_{file}$’s \texttt{onTrigger} function, the \texttt{setOutput} function is used to trigger a change in the output event. This function commits the monitor to using that event as the output event after all obligations have been executed. Once \texttt{setOutput} has been called for a specific trigger event, further calls to \texttt{setOutput} by any policy for the same trigger event return \texttt{false} indicating that the output event cannot be overwritten. To manage this condition, a policy may first call the \texttt{getOutput} or \texttt{outputNotSet} functions to verify that an output event has not yet been set; policy logic may then determine what needs to be done if the output event can not be set to the desired value.

$P_{confirm}$’s \texttt{onTrigger} function checks if the input event, $e$, is a file-open action and if it is and an output event has not been set, then the \texttt{onTrigger} function specifies an obligation to confirm the action $e$. Based on the result of the confirmation dialog, the \texttt{onTrigger}
either sets the output event to $e$ (indicating that the original action must be executed) or $\text{unit}$ (indicating that an empty result should be returned to the application instead of opening the file). If $e$ is not a file-open action or an output event has already been set for the current trigger action, $P_{\text{confirm}}$’s $\text{onTrigger}$ function does nothing. Hence, $P_{\text{confirm}}$’s $\text{onTrigger}$ is defined as follows.

```haskell
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(res(a,makeTypedVal(Unit,unit))):Event); unit
    else unit
    | res r => unit),
```

This ability for a policy to permanently set the output event is required in order for $P_{\text{confirm}}$ to operate as expected. If it were possible for the designated event not to execute due to other policies restrictions, then the user experience the unexpected behavior where they opt to allow the file open but file is not opened or they opt to disallow the file open and it gets executed anyway. This granular control also enables well-written policies to self-manage in when they conflict with other policies.

### 6.2 The vote Policy Function

The second element in the PoCo policy structure is the $\text{vote}$ function. The $\text{vote}$ function is responsible for supplying the policy’s vote on whether a given obligation should be allowed to execute or not. To enable the analysis that is necessary to allow the obligations to be understood prior to their execution, PoCo represents each obligation as a statically generated Control Flow Graph (CFG). The $\text{vote}$ function accepts such a statically generated CFG of an obligation and returns a Boolean vote indicating approval or disapproval of it. For
example, the goal of $P_{file}$ is to prevent the secret.txt file from being opened, even by other policies’ obligations. Therefore, when it examines an obligation, $P_{file}$’s vote function looks for instances of `fopen(secret.txt)` in the obligation’s CFG. If that specific action is found, the vote function returns false in order to express disapproval of the obligation. If that action is not found $P_{file}$’s vote function returns true to indicate approval of the obligation. 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)))
```

These statically generated CFGs are conservative approximations of the obligations they represent since computing the exact CFG for an arbitrary program is undecidable. Because it is not always possible to determine the exact arguments that will be used for each action in an obligation statically, it is necessary to allow these CFGs to include unresolved arguments, which are used to indicate parameters to a security-relevant action that could not be determined prior to execution. Indicating these unresolved arguments in the CFG allows each policy to specify how they should be handled. For example, $P_{file}$ could vote against obligations known to open the secret.txt file as well as obligations containing file opens with unresolved arguments. $P_{file}$’s vote is therefore:

```
6.3 The onObligation Policy Function

The third element of the PoCo policy structure is the onObligation function. The onObligation function is an obligation that may be executed following the execution of other obligations to inject supplemental logic after the triggering obligation. This is necessary to achieve the stated goal of allowing policies to react to other policies’ obligations. The onObligation function reacts to other policies’ obligations by analyzing the results of all security-relevant actions completed during an obligation’s execution. This series of results is known as a result trace. For example, \( P_{\text{postlog}} \) uses the onObligation function to log every file open action that occurs in another obligation. This obligation is specified in \( P_{\text{postlog}} \)’s onObligation function as follows:

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

It is not possible for PoCo to insert any additional obligations prior to the execution of a triggering obligation. Doing so may create inconsistency in the execution. Before executing any obligation \( o_1 \), the monitor decides whether \( o_1 \) should be executed. If, after the decision was made to execute \( o_1 \) but before it was actually executed, another another obligation \( o_2 \) was executed, it is possible that policies may vote differently than they did when originally deciding to permit \( o_1 \). If PoCo were designed to re-query policies after \( o_2 \) was executed, and the new decision was to not skip execution of \( o_1 \), then \( o_2 \) should never have been proposed since it was triggered by the pending execution of \( o_1 \). In order to have an enforcement
mechanism that behaves reliably, the voting on and execution of a given obligation must be treated as an atomic unit. For this reason, it is not possible in $P_{\text{prelog}}$ to log events occurring in obligations prior to their execution even though it is possible to do so for trigger events in the \texttt{onTrigger} function.

### 6.4 Parameterized Policies

To allow policy writers to reuse common code patterns in their policies, PoCo is designed to enable abstraction over policies. This is accomplished by declaring a function to instantiate policies based on the provided argument. For example, there are many types of policies that might want to disallow a specific action. This set of policies can be abstracted over with the following function:

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

This functionality might be used to specify any data that is relevant to a specific instance of a policy. For example, directory paths, port numbers, or any other value that may change based on the situation that the policy is being used in.
6.5 Local Policy State

The ability to keep local state information is essential for the expressiveness of an enforcement system. Without it any policy that must “remember” details about previous events cannot be enforced. It has been noted that, in general, restricting a monitor’s access to state information can drastically limit the policies that are enforceable [10]. PoCo policies are able to use let environments and memory references to manage this local state information.

\( P_{\text{time}} \), shown below, tracks the last time a popup window was displayed by the monitored application. If the “popup” action occurred less than 100 seconds ago and the application attempts to open another one, \( P_{\text{time}} \)’s \texttt{onTrigger} function attempts to exit the application. The time variable, \( t \), is used to record the time of the last occurrence of the “popup” action. The \texttt{onTrigger} function compares this value to the current time during future attempts to execute the “popup” action.

\[
\begin{aligned}
\text{let } t \text{ = ref } 0 \text{ in (}
\begin{aligned}
\text{name } &= \text{ pol\_time,}
\text{onTrigger } &= \text{( fun } \text{o}(\text{e: Event }): \text{Unit } =
\begin{aligned}
\text{case } \text{e of } \text{act a =>}
\text{if } \text{a.name=="popup" then}
\text{if } \text{currTime<!(t+100 } \land \text{ outputNotSet() then}
\text{setOutput( act("exit", makeTypedVal( Unit, unit )))}
\text{else } t := \text{currTime}
\text{else } \text{unit}
\end{aligned}
\end{aligned}
\text{end})
\end{aligned}
\end{aligned}
\]

Local state information can be used to track any value that should not be lost between triggering events. Another potential use case for this functionality is to track if (or how many times) an event has occurred including any specific parameters that were used in such events.
6.6 Complete Example Policy Specifications

This section contains the complete PoCo specifications for all example policies presented in this chapter. Taken together, the policies shown in this chapter exhibit all of the core features of the PoCo language and enforcement system. In addition to specifying the three components discussed in this chapter, each full policy specifies a name that can be used during composition when it is necessary to distinguish between policies.

The $P_{file}$ policy disallows the target application and other obligations from opening the secret.txt file. The policy actively tries to exit the monitored application if it is the source of the attempted action. The policy votes to disallow any obligation that would open the forbidden file.

```
1 ( 
2    name = polfile, 
3    onTrigger = (fun ot(e:Event):Unit = 
4      case e of act a =>
5          if a.name == "fopen" ∧ tryCast(String,a.arg) == "secret.txt"
6              then setOutput(event(act("exit", makeTypedVal(Unit,unit))))
7              else unit 
8          | res r ⇒ unit),
9    onObligation = (fun oo(rt: ResList):Unit = unit),
10   vote = (fun vt(cfg: CFG):Bool =
11          ¬call(containseAct, cfg = cfg, name = "fopen",
12              arg=inarg makeTypedVal(String, "secret.txt")(arg:TypedVal + none: unit, count=1)) ∧
13          ¬call(containseAct, cfg = cfg, name = "fopen",
14              arg=inone unit:(arg:TypedVal + none: unit, count=1)) )
15 )
```

The function disallow specifies a generic family of policies that, similar to $P_{file}$, wish to disallow a specific action. This function instantiates the needed policy based on the parameter provided to the function.
The $P_{\text{postlog}}$ policy logs all file-open actions after they occur. This logging includes open-file actions that occur in both the target application and other policies’ obligations. It does not attempt to prevent any obligations from executing.
The $P_{\text{prelog}}$ policy logs all target application file-open actions before they are executed and all obligation file-open actions after they occur. Note that, based on the discussion in Section 6.3, it is not possible to log obligation actions prior to their execution.

The $P_{\text{confirm}}$ policy requires all file-open actions attempted by the target application to be confirmed by the user through a pop-up window prior to allowing the action. It does not enforce the same restrictions on other policies' obligations.
The \( P_{time} \) policy disallows pop-ups unless at least 100 seconds have passed since the last pop-up was initiated by the monitored application.

\[
\begin{align*}
\text{let } t = \text{ref 0 in (}
&
\text{name = pol}_{time}, \\
&
\text{onTrigger} = (\text{fun } o\text{t}(e:\text{Event}):\text{Unit} =
\begin{cases}
\text{case } e \text{ of}
&
\text{act } a \rightarrow \\
&\text{if } a.\text{name}=="\text{popup}" \land \text{outputNotSet()} \text{ then}
&\text{if } \text{call}(\text{popupConfirm},e)
&\text{then } \text{setOutput}(e)
&\text{else } \text{setOutput}(\text{inۋres}(a,\text{makeTypedVal(Unit,unit))}:\text{Event})); \text{unit}
&\text{else } \text{unit}
&| \text{res } r \rightarrow \text{unit}), \\
&\text{onObligation} = (\text{fun } o\text{o}(rt: \text{Res}_L\text{ist}):\text{Unit} = \text{unit}, \\
&\text{vote} = (\text{fun } v\text{t}(\text{cfg}: \text{CFG}):\text{Bool} =
&\neg \text{call}(\text{containsActAnyArg}, (\text{cfg}=\text{cfg}, \text{name}="\text{popup}" , \text{count}=2)) \land \\
&\neg (\text{call}(\text{containsActAnyArg}, (\text{cfg}=\text{cfg}, \text{name}="\text{popup}" , \text{count}=1)) \land \\
&\text{currTime} < (!t+100))) \text{ end}
\end{cases}
\end{align*}
\]
Chapter 7: Policy Composition

The PoCo monitor, briefly discussed in Section 4.2, handles composition of policies by scheduling policies’ obligations, dispatching the agreed-upon output event, and handing control back to the application or system as appropriate.

This composition requires that the PoCo monitor be able to successfully identify and handle conflicts between policies. Conflicts produced when composing PoCo policies fall into two high-level 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 \texttt{vote} method returning a deny response to \( o \)’s request to execute. 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. This vote combinator can be customized to meet the needs of the composed policy being implemented.

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 in ordering. Like the

\[^1\text{Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.}\]
vote combinator, the obligation scheduler can be completely customized to meet the needs of the composed policy being implemented.

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

7.1 Policies

The policy list is first parameter used to initialize the PoCo monitor. The policy list includes the specification of all individual policies to be enforced by the monitor. Each policy is registered with the monitor 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.

7.2 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 $(\text{name} : \text{String} \times \text{vote} : \text{Bool})_{\text{List}} \rightarrow \text{Bool}$, which will be referred to as simply VC. Some possible VCs include conjunction, disjunction, and majority.

A VC can implement any arbitrary logic that the policy architect desires. For example, one could write a VC that executes an obligation as long as a specified policy, say P1, does...
not veto it. This vote combinator essentially ignores the votes of all policies except for P1. This VC would look like:

```ocaml
define VCvote \(\text{name}: \text{String} \times \text{vote}: \text{Bool}\) \text{List} \rightarrow \text{Bool} =
  let output = ref true in
  let rvotes = ref votes in
  while (not empty (!rvotes)) {
    case head (!rvotes) of
      some v =>
        if v.name == "P1" then
          output := v.vote
        else unit
      none unit => unit;
    rvotes := tail (!rvotes)
  }
  !output
end
```

The PoCo implementation includes several built-in vote combinators that can be used independently or as a building block to create more complex 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:

```ocaml
define VCoverride \(\text{name}: \text{String} \times \text{vote}: \text{Bool}\) \text{List} \rightarrow \text{Bool} =
  call (VCdisjunction, call (VCconjunction, 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.
7.3 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:

```haskell
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. This scheduling strategy closely models the standard shortest job first algorithm when considering possible security relevant events as the job length.

```haskell
fun OS_complexity (obs : CFG_List) : CFG_List =
    call (sort, (list = obs,
        comparator = (fun c((o_1,o_2):(CFG\times CFG)):Int =
            call (length,o_1.nodes) - call (length,o_2.nodes))
    ))
```
Figure 7.1: Policy scheduler flow. Obligations are generated by policies, 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.

7.4 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 a single obligation o, the monitor collects a vote on o from each registered policy and sends these votes to the VC to make the final decision on whether the monitor should execute o. Once an 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. This process is depicted in Figure 7.1.
\[ \text{monitor}(\tau, c) := \]

\begin{verbatim}
let pols = ref c.pols in
let obQueue = ref []:(pol:Pol × cfg:CFG)List in
let obStack = ref []:(pol:Pol × cfg:CFG)ListList in
while (¬empty(pols)) {
  obQueue := ! obQueue @ (pol = head(! pols), cfg =
  makeCFG(inState(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 []:BoolList 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) :: []:BoolList end;
  if call(c.vc, ! votes) then
    case ob.obligation of ot o₁ ⇒ call(o₁.onTrig, o₁ evt)
    | oo o₂ ⇒ call(o₂.onOblig, o₂ 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(inState(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
\end{verbatim}

Figure 7.2: The PoCo monitor algorithm
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.

Figure 7.2 presents the PoCo monitor algorithm. The monitor is defined as an expression parameterized by $\tau$ and $\text{config}$. An instance $e_{\text{monitor}}(\tau,(\text{evt}=e,\text{pols}=\text{pols},\text{os}=\text{os},\text{vc}=\text{vc}))$ monitors the security-relevant event $e$—whose return type is $\tau$—using policies $\text{pols}$, obligation scheduler $\text{os}$, and vote combinator $\text{vc}$. Lines 5 through 9 of the algorithm collect the CFG representations of the $\text{onTrigger}$ obligation for each registered policy for the input event and places them in a queue. On line 11 this queue is pushed onto a stack that will later allow any $\text{onObligation}$ obligations that are generated to be processed closer to their time of generation. The loop from line 12 to line 38 is responsible for the processing of each individual obligation. The next obligation to process is selected in lines 13 and 14, is voted on in lines 15-23 and if the $\text{VC}$ returns true on line 24 is executed on lines 25 and 26. Once an obligation has been executed, line 28 of the algorithm checks to see if there were any any security-relevant events executed; if there were the algorithm collects the CFG representations of the $\text{onObligation}$ obligation of each registered policy and places them in a queue. This queue is then added to the top of the stack of obligations waiting to be executed on line 36. Adding to the top of this stack ensures that these newly generated obligations are processed next.
Chapter 8: PoCo Language Properties

Using the semantics shown in Chapter 5 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 B. 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 (x) and $end_{f(x);v}$ indicates the end of a step with applicable parameters (x) and result (v). To simplify theorems’ presentation, the syntax also defines the following values:

\[
\begin{align*}
N & ::= \text{# of policies in } Pol_{List} \\
\text{beginOb}(e) & ::= begin_{onTrigger(e)}|begin_{onObligation(e)} \\
\text{endOb}(e) & ::= end_{onTrigger(e)}|end_{onObligation(e)} \\
\text{ob}(e) & ::= \text{beginOb}(e) — \text{endOb}(e)
\end{align*}
\]
8.1 Definition of $\infty$-Languages

To express the program traces that are used in theorems in this chapter, this section defines the notion of an $\infty$-language. An $\infty$-language can generate possibly infinite length strings (i.e., belonging to the union of a regular and an $\omega$ language).

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

1. If a language $L$ is regular, then $L$ is an $\infty$-language

2. If an $\omega$-language $L$ is $\omega$-regular, then $L$ is an $\infty$-language

3. If $L_1, L_2$ are $\infty$-languages, then $L_1 L_2$ is an $\infty$-language, and $L_1 L_2 = \{xy \mid x \text{ is finite, else } x \in L_1 \times L_2\}$

As Rules 1 and 2 show, the set of $\infty$-languages is the union of regular languages and $\omega$-regular languages, while Rule 3 defines concatenation such that the set of $\infty$-languages is closed under this operation. Defining concatenation this way allows for an infinitely-long string to be concatenated with another string. Note that concatenating a finite string with an infinite string is already well-defined in $\omega$-regular languages, but concatenating an infinite string with a finite string is not defined. The addition of this concatenation option differentiates $\infty$-languages from $\omega$-regular languages. The inclusion of possibly infinite strings differentiates $\infty$-languages from regular languages.

With this third rule, a left-hand concatenation operand $L^\infty$ is either a finite repetition of $L$ followed by the rest of the expression, or an infinite repetition of $L$. This 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. With this concatenation rule, a possibly divergent program trace can be expressed concisely. For instance, consider the $\infty$-expression $ax^\infty by^\infty c$. As shown by in
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”.

\[ \text{ax}^\omega b \text{y}^\omega c \]

\[ = a(x^\omega|x^*)b(y^\omega|y^*)c \quad \text{[definition of } L^\omega \text{]} \]

\[ = (ax^\omega | ax^*)b(y^\omega | y^*)c \quad \text{[distribution]} \]

\[ = (ax^\omega b|ax^*b)(y^\omega|y^*)c \quad \text{[distribution]} \]

\[ = (ax^\omega | ax^*b)(y^\omega | y^*)c \quad \text{[concatenation]} \]

\[ = (ax^\omega y^\omega | ax^*y^* | ax^*by^* | ax^*by^*)c \quad \text{[distribution]} \]

\[ = (ax^\omega | ax^*by^\omega | ax^*by^*)c \quad \text{[concatenation]} \]

\[ = (ax^\omega | ax^*by^\omega | ax^*by^*)c \quad \text{[idempotent]} \]

\[ = ax^\omega c|ax^*by^\omega | ax^*by^*c \quad \text{[distribution]} \]

\[ = ax^\omega | ax^*by^\omega | ax^*by^*c \quad \text{[concatenation]} \]

8.2 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. Atomicity 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 * p' and t matches the \( \infty \)-expression \((. \infty) \) beginOb\( (e_n) \) t' beginOb\( (e_m) (.\infty) \) then t' matches the \( \infty \)-expression \((. \infty) \) endOb\( (e_n) (.\infty) \)

Essentially, Theorem 3 states that another obligation will never start if the previous obligation has not completed execution.
8.3 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 $p \xrightarrow{\tau}^* p'$, $t$ matches the $\infty$-expression $(-beginOb(e))^{\infty}(v_{true}(e_n)beginOb(e_n)(-beginOb(e))^{\infty})^{\infty}$ where:

$v_{true}(e) ::= (begin_{vote}(e) (-beginOb(e))^{\infty} end_{vote(e)} : v_n) \begin{array}{c} N \end{array} begin_{vc}(v_1:::\cdots::v_N) (-beginOb(e))^{\infty} end_{vc}(v_1:::\cdots::v_N) : true$

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.

8.4 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 $p \xrightarrow{\tau}^* p'$, $t$ matches the $\infty$-expression

$((-begin_{appendRes()}^{\infty}(begin_{appendRes()}^{\infty} endOb(e)(.^{\infty} end_{makeCFG(onObligation, v)} : g)^{N})^{?})^{\infty}$

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 does nothing. This theorem, by itself, is insufficient to prove
that these obligations are executed once they are retrieved; Theorem 6 is needed in order to complete the proof that proposed obligations are then voted on and executed, if approved.

**Theorem 6 (Obligation Reaction Part 2).** For all well-typed programs \( p \) where \( p \xrightarrow{\cdot}^* p' \) and \( p \)'s monitor is the tuple \((M, \text{fun}_{\text{mon}}, p_1 :: \cdots :: p_n, e_{\text{os}}, e_{\text{vc}})\) with functions \( e_{\text{vc}}, p_1.\text{onTrigger}, \ldots, p_n.\text{onTrigger}, \ p_1.\text{onObligation}, \ldots, p_n.\text{onObligation} \) that terminate, for each trace event \( \text{end}_{\text{makeCFG}(v_1, v_2)} : g \) in \( t \) there must exist a \( v_{\text{true}}(g) \) or \( v_{\text{false}}(g) \) in \( t \) where:

\[
v_{\text{true}}(e) := (\text{begin}_{\text{vote}}(e) \ (\neg \text{beginOb}(e))^{\infty} \ \text{end}_{\text{vote}}(e) : v_n)^N \ \text{begin}_{\text{vc}}(v_1 :: \cdots :: v_N) (\neg \text{beginOb}(e))^{\infty} \ \text{end}_{\text{vc}}(v_1 :: \cdots :: v_N) : \text{true}
\]

\[
v_{\text{false}}(e) := (\text{begin}_{\text{vote}}(e) \ (\neg \text{beginOb}(e))^{\infty} \ \text{end}_{\text{vote}}(e) : v_n)^N \ \text{begin}_{\text{vc}}(v_1 :: \cdots :: v_N) (\neg \text{beginOb}(e))^{\infty} \ \text{end}_{\text{vc}}(v_1 :: \cdots :: v_N) : \text{false}
\]

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.

### 8.5 Obligation Completeness

Although the categories *pre-*-, *post-*-, and *ongoing-* [24, 25, 4] 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 4.1. 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 under-
lying 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 3.1 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 \( \infty \)-expressions \( e_{\text{pre}}, e_{\text{post}}, e_{\text{ongoing}}, \) respectively, where

\[
\begin{align*}
    e_{\text{pre}} &::= (\infty) \begin{align*}
        \text{begin} &\ f(x) \ (\infty) \begin{align*}
            \text{begin} &\ \text{monitor}(\text{act}(f,x)) \ (\infty) \begin{align*}
                \text{end} &\ f(x);v \ (\infty)
            \end{align*}
        \end{align*}
    \end{align*}
\end{align*}
\]

\[
\begin{align*}
    e_{\text{post}} &::= (\infty) \begin{align*}
        \text{end} &\ f(x);v \ (\infty) \begin{align*}
            \text{begin} &\ \text{monitor}(\text{res}(\text{act}(f,x),rt)) \ (\infty)
        \end{align*}
    \end{align*}
\end{align*}
\]

\[
\begin{align*}
    e_{\text{ongoing}} &::= (e_{\text{pre}} | e_{\text{post}}) (\infty) (e_{\text{pre}} | e_{\text{post}}).
\end{align*}
\]

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.

### 8.6 Policy Permutability

It has also been 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 that the design of such a monitor is possible, we must first define what it means for the outcome to be unaffected. In general
terms, this means that identical obligations should be executed in the same order, and the same output event should be selected by the monitor regardless of the order in which the policies are input to the monitor. To formalize this idea, \textit{trace event equivalence} is defined as follows:

\[
n_1 \approx n_2
\]

\[
\frac{n_1 = n_2}{n_1 \approx n_2}
\]

\[
\begin{align*}
\{p_1 :: \cdots :: p_n\} &\equiv \{p'_1 :: \cdots :: p'_n\} \\
n_1 = \text{begin}_{\text{os}}(p_1 :: \cdots :: p_n) &\quad n_2 = \text{begin}_{\text{os}}(p'_1 :: \cdots :: p'_n) \quad n_1 \approx n_2
\end{align*}
\]

\[
\frac{n_1 = \text{end}_{\text{os}}(p_1 :: \cdots :: p_n):v}{n_1 \approx n_2}
\]

\[
\begin{align*}
\{p_1 :: \cdots :: p_n\} &\equiv \{p'_1 :: \cdots :: p'_n\} \\
n_1 = \text{begin}_{\text{monitor}}(p_1 :: \cdots :: p_n) &\quad n_2 = \text{begin}_{\text{monitor}}(p'_1 :: \cdots :: p'_n) \quad n_1 \approx n_2
\end{align*}
\]

\[
\frac{n_1 = \text{end}_{\text{monitor}}(p_1 :: \cdots :: p_n):v}{n_1 \approx n_2}
\]

\[
\begin{align*}
\{p_1 :: \cdots :: p_n\} &\equiv \{p'_1 :: \cdots :: p'_n\} \\
n_1 = \text{begin}_{\text{makeCFG}}(o):v &\quad n_2 = \text{begin}_{\text{makeCFG}}(o'):v \quad n_1 \approx n_2
\end{align*}
\]

\[
\frac{n_1 = \text{end}_{\text{makeCFG}}(o):v'}{n_1 \approx n_2}
\]

In other words, two trace events are equivalent if they are identical events or if they are both calls to specific functions (\textit{os}, \textit{vc}, \textit{monitor}, \textit{makeCFG}) with varied parameters. A program trace is considered \textit{equivalent} if all of its trace events are equivalent. Trace equivalence guarantees that all effectful code is executed in the same order in each equivalent trace by guaranteeing that all obligations are executed in the same order.
Theorem 8 (Policy Permutability). There exists well-typed programs $p_1$ with monitor $(F, p_1 :: \cdots :: p_n, e_{os}, e_{vc})$ and $p_2$ with monitor $(F, p'_1 :: \cdots :: p'_n, e_{os}, e_{vc})$ 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 it is possible to create an OS and VC pair where inputting the same policies with different ordering would result in identical order of execution for all effectful code. In other words, the user of the application would not notice a difference. Not all configured monitors will need to make use of this property, but when true modularity is needed, designing the monitor to exhibit the policy permutability property will allow policies to be more freely added and removed. A proof for this theorem can be found in Appendix B.
Chapter 9: Conclusions

This dissertation presented PoCo, a policy-specification language and enforcement system designed to enable the composition of atomic-obligation policies in a manner that is reliable, consistent and versatile. The language used to encode PoCo policies is Turing complete and allows for the enforcement of obligations that may include observable side effects. The PoCo system enables obligations that are pre-, post- or ongoing. The design of the PoCo enforcement system relies on static analysis to allow policies to validate the obligations of other policies prior to execution and allows policies to react to these obligations after execution has completed. In addition to the expressive nature of individual PoCo policies, the PoCo enforcement system allows versatile composition of these policies by allowing custom operators for prioritizing and evaluating obligations.

A full syntax and semantics for the PoCo language is presented as well as a proof of this language is type safe (Theorem 2). Additionally, a number of useful properties regarding PoCo obligations are proven based on these semantics:

- event-by-event obligations and atomic obligations are mutually exclusive (Theorem 1)
- all obligations are atomic (Theorem 3)
- obligations always allow for conflict resolution (Theorem 4)
- policies can always react to other policies’ obligations (Theorems 5 and 6)
- pre-obligations can be used to implement post- and ongoing obligations (Theorem 7)

¹Significant portions of this chapter were published in the International Conference on Software and Computer Applications, 2020. Permission is included in Appendix C.
• it is possible to design a PoCo monitor such that the order in which the policies are declared does not affect the outcome (Theorem 8)

In addition to the formal definition and analysis of the PoCo language and enforcement system that was presented in this dissertation, PoCo has been implemented and evaluated by enforcing a case-study composition of eleven policies for securing an email client. This work showed that the PoCo language was able to express a wide variety of policies and that policy enforcement run time was dominated by policy logic and had very little overhead related directly to the enforcement code.
References


Appendix A: Proof of Type Safety\textsuperscript{1}

A proof of PoCo’s type-safety is presented below. The proof consists of seven main lemmas: \(\Lambda\)-Weakening (on page 75), Weakening (on page 82), Substitution (on page 89), Typing Rule Inversion (on page 97), Canonical Forms (on page 102), Progress (on page 107), and Preservation (on page 122). Throughout these proofs, the shorthand “IH” refers to the inductive hypothesis.

**Lemma 1 (C-Inversion).** \(\Lambda \vdash (M, R, inOb, rt, out, \tau_{out}) \ ok \Rightarrow M : \Lambda \land \Lambda \vdash R \ ok \land \Lambda , \bullet \vdash inOb : \text{Bool} \land Abullet \vdash rt : \text{Res}_{List} \land \Lambda , \bullet \vdash out : \tau_{out} \ Event\ Option\)

**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. \(\Lambda, \bullet \vdash inOb : \text{Bool}\) 2, Inversion of Rule C-ok
6. \(\Lambda, \bullet \vdash rt : \text{Res}_{List}\) 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).** \(\Lambda \vdash (F, pols, os, vc) \ ok \Rightarrow \Lambda \vdash F \ ok \land \Lambda , \bullet \vdash pols : \text{Pol}_{List} \land \Lambda , \bullet \vdash os : \text{OS} \land \Lambda , \bullet \vdash vc : \text{VC}\)

**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. \(\Lambda, \bullet \vdash pols : \text{Pol}_{List}\) 2, Inversion of Rule R-ok
5. \(\Lambda, \bullet \vdash os : \text{OS}\) 2, Inversion of Rule R-ok
6. \(\Lambda, \bullet \vdash vc : \text{VC}\) 2, Inversion of Rule R-ok

Result is from 3-6

\textsuperscript{1}This proof was created in collaboration with Tyler Hanks and Kevin Orr.
Lemma 3 (C-Weakening). \( \Lambda \vdash (M, (F, \text{pols}, \text{os}, \text{vc}), \text{inOb}, \text{rt}, \text{out}, \tau_{\text{out}}) \text{ ok} \land M' : \Lambda' \land \Lambda \subseteq \Lambda' \Rightarrow \Lambda' \vdash (M', (F, \text{pols}, \text{os}, \text{vc}), \text{inOb}, \text{rt}, \text{out}, \tau_{\text{out}}) \text{ ok} \)

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

\[ \square \]

Lemma 4 (\( \Lambda \)-Weakening). \( (\Lambda_1, \Gamma \vdash e : \tau \land \Lambda_1 \subseteq \Lambda_2) \Rightarrow \Lambda_2, \Gamma \vdash e : \tau \)

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

Case \( \frac{\Lambda_1, \Gamma \vdash n : \text{Int}}{(\text{intVal})} \)
1. \( \Lambda_2, \Gamma \vdash n : \text{Int} \) Rule intVal

Case \( \frac{\Lambda_1, \Gamma \vdash b : \text{Bool}}{(\text{boolVal})} \)
1. \( \Lambda_2, \Gamma \vdash b : \text{Bool} \) Rule boolVal

Case \( \frac{\Lambda_1, \Gamma \vdash s : \text{String}}{(\text{stringVal})} \)
1. \( \Lambda_2, \Gamma \vdash s : \text{String} \) Rule stringVal
Case $\Lambda_1, \Gamma \vdash e : \text{Int}$ (unitVal)

1. $A_2, \Gamma \vdash \text{unit} : \text{Unit}$ Rule unitVal

   Case $\Lambda_1, \Gamma \cup \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e : \tau_2$ (fun)
   1. $\Lambda_1, \Gamma \cup \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e : \tau_2$ assumption
   2. $A_1 \subseteq \Lambda_2$ assumption
   3. $A_2, \Gamma \cup \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e : \tau_2$ 1, 2, IH
   4. $A_2, \Gamma \vdash (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e) : \tau_1 \rightarrow \tau_2$ 3, Rule fun

Case $(\Lambda'_1 \cup \{\ell : \tau\}), \Gamma \vdash \ell : \tau$ (ref)

1. $A_1 = \Lambda'_1 \cup \{\ell : \tau\}$ assumption
2. $A_1 \subseteq \Lambda_2$ assumption
3. $\ell : \tau \in \Lambda_2$ 1, 2, Definition of $\subseteq$
4. $A_2 = \Lambda'_2 \cup \{\ell : \tau\}$ 3
5. $A_2, \Gamma \vdash \ell : \tau$ Ref 4, Rule location

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

1. $A_2, \Gamma' \cup \{x : \tau\} \vdash x : \tau$ Rule var

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

1. $\Lambda_1, \Gamma \vdash e_1 : \text{Bool}$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \text{Bool}$ assumption
3. $A_1 \subseteq \Lambda_2$ assumption
4. $A_2, \Gamma \vdash e_1 : \text{Bool}$ 1, 3, IH
5. $A_2, \Gamma \vdash e_2 : \text{Bool}$ 2, 3, IH
6. $A_2, \Gamma \vdash e_1 \land e_2 : \text{Bool}$ 4, 5, Rule con

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

1. $\Lambda_1, \Gamma \vdash e_1 : \text{Bool}$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \text{Bool}$ assumption
3. $A_1 \subseteq \Lambda_2$ assumption
4. $A_2, \Gamma \vdash e_1 : \text{Bool}$ 1, 3, IH
5. $A_2, \Gamma \vdash e_2 : \text{Bool}$ 2, 3, IH
6. $A_2, \Gamma \vdash e_1 \lor e_2 : \text{Bool}$ 4, 5, Rule dis

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

1. $\Lambda_1, \Gamma \vdash e : \text{Bool}$ assumption
2. $A_1 \subseteq \Lambda_2$ assumption
3. $A_2, \Gamma \vdash e : \text{Bool}$ 1, 2, IH
4. $A_2, \Gamma \vdash \neg e : \text{Bool}$ 3, Rule negation
\[
\text{Case } \begin{align*}
\Lambda_1, \Gamma &\vdash e_1 : \tau \\
\Lambda_1, \Gamma &\vdash e_2 : \tau
\end{align*}
\quad \tau \in \{\text{Int, Bool, String}\} \quad (\text{equality})
\]

1. \(\Lambda_1, \Gamma \vdash e_1 : \tau\) assumption
2. \(\Lambda_1, \Gamma \vdash e_2 : \tau\) assumption
3. \(\tau \in \{\text{Int, Bool, String}\}\) 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. \(\Lambda_2, \Gamma \vdash e_1 == e_2 : \text{Bool}\) 3, 5, 6, Rule equality

\[
\text{Case } \begin{align*}
\Lambda_1, \Gamma &\vdash e_1 : \text{Int} \\
\Lambda_1, \Gamma &\vdash e_2 : \text{Int}
\end{align*} \quad (\text{add})
\]

1. \(\Lambda_1 \subseteq \Lambda_2\) assumption
2. \(\Lambda_1, \Gamma \vdash e_1 : \text{Int}\) assumption
3. \(\Lambda_1, \Gamma \vdash e_2 : \text{Int}\) assumption
4. \(\Lambda_2, \Gamma \vdash e_1 : \text{Int}\) 1, 2, IH
5. \(\Lambda_2, \Gamma \vdash e_2 : \text{Int}\) 1, 3, IH
6. \(\Lambda_2, \Gamma \vdash e_1 + e_2 : \text{Int}\) 4, 5, Rule add

\[
\text{Case } \begin{align*}
\Lambda_1, \Gamma &\vdash e_1 : \tau_1 \\
\Lambda_1, \Gamma &\vdash e_2 : \tau_2
\end{align*} \quad (\text{sequence})
\]

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. \(\Lambda_2, \Gamma \vdash e_1 : \tau_2\) 4, 5, Rule sequence

\[
\text{Case } \begin{align*}
\Lambda_1, \Gamma &\vdash e_1 : \text{Bool} \\
\Lambda_1, \Gamma &\vdash e_2 : \tau \\
\Lambda_1, \Gamma &\vdash e_3 : \tau
\end{align*} \quad (\text{if})
\]

1. \(\Lambda_1, \Gamma \vdash e_1 : \text{Bool}\) 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. \(\Lambda_2, \Gamma \vdash e_1 : \text{Bool}\) 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. \(\Lambda_2, \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau\) 5-7, Rule if

\[
\text{Case } \begin{align*}
\Lambda_1, \Gamma &\vdash e_1 : \text{Bool} \\
\Lambda_1, \Gamma &\vdash e_2 : \tau
\end{align*} \quad (\text{while})
\]

1. \(\Lambda_1, \Gamma \vdash e_1 : \text{Bool}\) assumption
2. \(\Lambda_1, \Gamma \vdash e_2 : \tau\) assumption
3. \(\Lambda_1 \subseteq \Lambda_2\) assumption
4. \(\Lambda_2, \Gamma \vdash e_1 : \text{Bool}\) 1, 3, IH
5. \(\Lambda_2, \Gamma \vdash e_2 : \tau\) 2, 3, IH
6. \(\Lambda_2, \Gamma \vdash \text{while}(e_1) \{e_2\} : \text{Bool}\) 4, 5, Rule while
\[
\text{Case } \frac{\Lambda_1, \Gamma \vdash e_1 : \tau_1 \quad \Lambda_1, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{\Lambda_1, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2} \quad \text{(let)}
\]

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. \(\Lambda_2, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2\) 4, 5, Rule let

\[
\text{Case } \frac{\Lambda_1, \Gamma \vdash e : \tau}{\Lambda_1, \Gamma \vdash \text{ref } e : \tau \text{ Ref}} \quad \text{(createRef)}
\]

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. \(\Lambda_2, \Gamma \vdash \text{ref } e : \tau \text{ Ref}\) 3, Rule createRef

\[
\text{Case } \frac{\Lambda_1, \Gamma \vdash e : \tau \text{ Ref}}{\Lambda_1, \Gamma \vdash !e : \tau} \quad \text{(accessRef)}
\]

1. \(\Lambda_1, \Gamma \vdash e : \tau \text{ Ref}\) assumption
2. \(\Lambda_1 \subseteq \Lambda_2\) assumption
3. \(\Lambda_2, \Gamma \vdash e : \tau \text{ Ref}\) 1, 2, IH
4. \(\Lambda_2, \Gamma \vdash !e : \tau\) 3, Rule accessRef

\[
\text{Case } \frac{\Lambda_1, \Gamma \vdash e_1 : \tau \text{ Ref} \quad \Lambda_1, \Gamma \vdash e_2 : \tau}{\Lambda_1, \Gamma \vdash e_1 :: e_2 : \text{Unit} \quad \text{ assignment}}
\]

1. \(\Lambda_1, \Gamma \vdash e_1 : \tau \text{ Ref}\) assumption
2. \(\Lambda_1, \Gamma \vdash e_2 : \tau\) assumption
3. \(\Lambda_1 \subseteq \Lambda_2\) assumption
4. \(\Lambda_2, \Gamma \vdash e_1 : \tau\) 1, 3, IH
5. \(\Lambda_2, \Gamma \vdash e_2 : \tau\) 2, 3, IH
6. \(\Lambda_2, \Gamma \vdash e_1 :: e_2 : \text{Unit}\) 4, 5, Rule assignment

\[
\text{Case } \frac{\Lambda_1, \Gamma \vdash ([] : \text{List}) : \text{List}}{\text{listEmptyVal}}
\]

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

\[
\text{Case } \frac{\Lambda_1, \Gamma \vdash e_1 : \tau \quad \Lambda_1, \Gamma \vdash e_2 : \text{List}}{\Lambda_1, \Gamma \vdash e_1 :: e_2 : \text{List} \quad \text{listCons}}
\]

1. \(\Lambda_1, \Gamma \vdash e_1 : \tau\) assumption
2. \(\Lambda_1, \Gamma \vdash e_2 : \text{List}\) assumption
3. \(\Lambda_1 \subseteq \Lambda_2\) assumption
4. \(\Lambda_2, \Gamma \vdash e_1 : \tau\) 1, 3, IH
5. \(\Lambda_2, \Gamma \vdash e_2 : \text{List}\) 2, 3, IH
6. \(\Lambda_2, \Gamma \vdash e_1 :: e_2 : \text{List}\) 4, 5, Rule listCons
Case $\begin{array}{l}
\Lambda_1, \Gamma \vdash e_1 : \tau_{\text{List}} \\
\Lambda_1, \Gamma \vdash e_2 : \tau_{\text{List}} \\
\end{array}$ (listAppend)
1. $\Lambda_1, \Gamma \vdash e_1 : \tau_{\text{List}}$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau_{\text{List}}$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau_{\text{List}}$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_2 : \tau_{\text{List}}$ 2, 3, IH
6. $\Lambda_2, \Gamma \vdash e_1 @ e_2 : \tau_{\text{List}}$ 4, 5, Rule listAppend

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

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

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

Case $\begin{array}{l}
\Lambda_1, \Gamma \vdash e_1 : \tau_1 \cdots \Lambda_1, \Gamma \vdash e_n : \tau_n \\
\Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \\
\end{array}$ (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. $\Lambda_2, \Gamma \vdash (\ell_1 = e_1, \ldots, \ell_n = e_n)$ 4, Rule record
   : $(\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$

Case $\begin{array}{l}
\Lambda_1, \Gamma \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \ i \in \{1, \ldots, n\} \\
\Gamma \vdash e.\ell_i : \tau_i \\
\end{array}$ (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 $\Lambda_1, \Gamma \vdash e : \tau_i \ i \in \{1, \ldots, n\}$ (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. $\Lambda_2, \Gamma \vdash (in_{\ell_i}, e_i : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ 2, 4, Rule variant

$$\Lambda_1, \Gamma \vdash e : \tau_i$$

1. $\Lambda_1, \Gamma \vdash e : \tau_i$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : \tau_i$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash \text{tryCast}(\tau, e_i) : \tau \text{ Option}$ 3, Rule tryCast

Case $\Lambda_1, \Gamma \vdash e : \text{TypedVal}$ (tryCast)

1. $\Lambda_1, \Gamma \vdash e : \text{TypedVal}$ assumption
2. $\Lambda_2 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e : \text{TypedVal}$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash \text{tryCast}(\tau, e_i) : \tau \text{ Option}$ 3, Rule tryCast

Case $\Lambda_1, \Gamma \vdash e : \tau$ (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. $\Lambda_2, \Gamma \vdash \{e\}_{\text{s(v)}} : \tau$ 3, Rule endLabel

Case $\Lambda_1, \Gamma \vdash \text{getRT}() : \text{Res}_\text{List}$ (getRT)

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

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

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

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

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

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

1. $\Lambda_2, \Gamma \vdash \text{outputNotSet}() : \text{Bool}$ Rule outputNotSet
Case $\Lambda_1, \Gamma \vdash getOutput() : Event Option$
1. $\Lambda_2, \Gamma \vdash getOutput() : Event Option$ Rule getOutput

Case $\Lambda_1, \Gamma \vdash e_1 : String$ $\Lambda_1, \Gamma \vdash e_2 : TypedVal$
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 $\Lambda_1, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$ $\Lambda_1, \Gamma \vdash e_2 : \tau_1$
1. $\Lambda_1, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$ assumption
2. $\Lambda_1, \Gamma \vdash e_2 : \tau_1$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$ 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 $\Lambda_1, \Gamma \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)$
1. $\Lambda_1, \Gamma \vdash e_1 : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n$ assumption
2. $\Lambda_1, \Gamma \vdash e_1 : \tau$ assumption
3. $\Lambda_1 \subseteq \Lambda_2$ assumption
4. $\Lambda_2, \Gamma \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)$ 1, 3, IH
5. $\Lambda_2, \Gamma \vdash e_1 : \tau$ assumption
6. $\Lambda_2, \Gamma \vdash (case e of \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau$ 4, 5, Rule case

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

Case $\Lambda_1, \Gamma \vdash e' : (evt : \tau' \ Event \times pols : PolList \times os : OS \times vc : VC)$
1. $\Lambda_1, \Gamma \vdash e' : (evt : \tau' \ Event \times pols : PolList \times os : OS \times vc : VC)$ assumption
2. $\Lambda_1 \subseteq \Lambda_2$ assumption
3. $\Lambda_2, \Gamma \vdash e' : (evt : \tau' \ Event \times pols : PolList \times os : OS \times vc : VC)$ 1, 2, IH
4. $\Lambda_2, \Gamma \vdash monitor(\tau', e') : \tau' \ Event$ 3, Rule monitor
Lemma 5 (Weakening). $\Lambda, \Gamma_1 \vdash e : \tau \land \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Lambda, \Gamma_2 \vdash e : \tau$

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

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

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

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

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

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

Case $\Lambda' \cup \{\ell : \tau\}, \Gamma_1 \vdash \ell : \tau \text{ Ref}$ (location)
1. $\Lambda' \cup \{\ell : \tau\}, \Gamma_2 \vdash \ell : \tau \text{ Ref}$ Rule location

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

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

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

Case $\Lambda, \Gamma_1 \vdash e_1 : \tau$ $\Lambda, \Gamma_1 \vdash e_2 : \tau$ $\tau \in \{\text{Int, Bool, String}\}$ (equality)
1. $\Lambda, \Gamma_1 \vdash e_1 : \tau$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \tau$ assumption
3. $\tau \in \{\text{Int, Bool, String}\}$ 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. $\Lambda, \Gamma_2 \vdash e_1 == e_2 : \text{Bool}$ 3, 5, 6, Rule equality

Case $\Lambda, \Gamma_1 \vdash e_1 : \text{Int}$ $\Lambda, \Gamma_1 \vdash e_2 : \text{Int}$ (add)
1. $\Lambda, \Gamma_1 \vdash e_1 : \text{Int}$ assumption
2. $\Lambda, \Gamma_1 \vdash e_2 : \text{Int}$ assumption
3. $\Gamma_1 \subseteq \Gamma_2$ assumption
4. $\Lambda, \Gamma_2 \vdash e_1 : \text{Int}$ 1, 3, IH
5. $\Lambda, \Gamma_2 \vdash e_2 : \text{Int}$ 2, 3, IH
6. $\Lambda, \Gamma_2 \vdash e_1 + e_2 : \text{Int}$ 4, 5, Rule add
Case $\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \Lambda, \Gamma_1 \vdash e_2 : \tau_2$ (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. $\Lambda, \Gamma_2 \vdash e_1 : \tau_2$ 4, 5, Rule sequence

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

1. $\Lambda, \Gamma_1 \vdash e_1 : \text{Bool}$ 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. $\Lambda, \Gamma_2 \vdash e_1 : \text{Bool}$ 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. $\Lambda, \Gamma_2 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau$ 5, 6, 7, Rule If

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

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

Case $\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \quad \Lambda, \Gamma_1 \cup \{x : \tau_1\} \vdash e_2 : \tau_2$ (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. $\Lambda, \Gamma_2 \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2$ 5, 6, Rule let
Case \( \Lambda, \Gamma_1 \vdash e : \tau \) (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. \( \Lambda, \Gamma_2 \vdash \text{ref } e : \tau \) Ref 3, Rule createRef

Case \( \Lambda, \Gamma_1 \vdash e : \tau \) Ref (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 \( \Lambda, \Gamma_1 \vdash e_1 : \tau \) Ref \( \Lambda, \Gamma_1 \vdash e_2 : \tau \) (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. \( \Lambda, \Gamma_2 \vdash e_1 := e_2 : \text{unit} \) 4, 5, Rule assignment

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

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

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

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

Case \( \Lambda, \Gamma_1 \vdash e_1 : \tau_{\text{List}} \) \( \Lambda, \Gamma_1 \vdash e_2 : \tau_{\text{List}} \) (listAppend)

1. \( \Lambda, \Gamma_1 \vdash e_1 : \tau_{\text{List}} \) assumption
2. \( \Lambda, \Gamma_1 \vdash e_2 : \tau_{\text{List}} \) assumption
3. \( \Gamma_1 \subseteq \Gamma_2 \) assumption
4. \( \Lambda, \Gamma_2 \vdash e_1 : \tau_{\text{List}} \) 1, 3, IH
5. \( \Lambda, \Gamma_2 \vdash e_2 : \tau_{\text{List}} \) 2, 3, IH
6. \( \Lambda, \Gamma_2 \vdash e_1 @ e_2 : \tau_{\text{List}} \) 4, 5, Rule listAppend
Case \( \Lambda, \Gamma_1 \vdash e : \tau_{List} \) (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 \( \Lambda, \Gamma_1 \vdash e : \tau_{List} \) (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 \( \Lambda, \Gamma \vdash empty(e) : \text{Bool} \) (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. \( \Lambda, \Gamma_2 \vdash empty(e) : \text{Bool} \) 3, Rule empty

Case \( \Lambda, \Gamma_1 \vdash e_1 : \tau_1 \cdots \Lambda, \Gamma_1 \vdash e_n : \tau_n \) (record)

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

Case \( \Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \) \( i \in \{1, \ldots, n\} \) (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. \( \Lambda, \Gamma_2 \vdash e.i : \tau_i \) 3, 4, Rule projection

Case \( \Lambda, \Gamma_1 \vdash e_i : \tau_i \) \( i \in \{1, \ldots, n\} \) (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. \( \Lambda, \Gamma_2 \vdash (\ell_1.e_i : \ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \) 3, 4, Rule variant
\[
\begin{align*}
&\text{Case } \Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \\
&\text{Case } \Lambda, \Gamma_1 \cup \{x_1 : \tau_1\} \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_1 \cup \{x_n : \tau_n\} \vdash e_n : \tau_1 \\
&\Lambda, \Gamma_1 \vdash (\text{case } e \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau_1 \\
\end{align*}
\]

1. \(\Lambda, \Gamma_1 \vdash e : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)\) assumption
2. \(\Lambda, \Gamma_1 \cup \{x_1 : \tau_1\} \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_1 \cup \{x_n : \tau_n\} \vdash e_n : \tau_1\) 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. \(\Lambda, \Gamma_2 \vdash e_1 : \tau_1 \quad \cdots \quad \Lambda, \Gamma_2 \cup \{x_n : \tau_n\} \vdash e_n : \tau_1\) 2, IH 
6. \(\Lambda, \Gamma_2 \vdash (\text{case } e \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau_1\) 5, 6, Rule case

\[
\begin{align*}
&\text{Case } \Lambda, \Gamma_1 \vdash e_1 : \tau_1 \\
&\text{Case } \Lambda, \Gamma_1 \vdash \text{makeTypedVal}(\tau, e_1) : \text{TypedVal} \\
&\text{Case } \Lambda, \Gamma_1 \vdash e_1 : \text{TypedVal} \\
&\text{Case } \Lambda, \Gamma_1 \vdash e : \text{label} : \tau \\
&\text{Case } \Lambda, \Gamma_1 \vdash \text{getRT()} : \text{Res}_\text{List} \\
\end{align*}
\]
Case $\Lambda, \Gamma_1 \vdash e : Event$  
\(\text{setOutput}\)  
\[1. \quad \Lambda, \Gamma_1 \vdash e : Event \quad \text{assumption} \]
\[2. \quad \Gamma_1 \subseteq \Gamma_2 \quad \text{assumption} \]
\[3. \quad \Lambda, \Gamma_2 \vdash e : Event \quad 1, 2, \text{IH} \]
\[4. \quad \Lambda, \Gamma_2 \vdash \text{setOutput}(e) : \text{Bool} \quad 3, \text{Rule setOutput} \]

Case $\Lambda, \Gamma_1 \vdash \text{outputNotSet}() : \text{Bool}$  
\(\text{outputNotSet}\)  
\[1. \quad \Lambda, \Gamma_2 \vdash \text{outputNotSet}() : \text{Bool} \quad \text{Rule outputNotSet} \]

Case $\Lambda, \Gamma_1 \vdash \text{getOutput}() : \text{Event Option}$  
\(\text{getOutput}\)  
\[1. \quad \Lambda, \Gamma_2 \vdash \text{getOutput}() : \text{Event Option} \quad \text{Rule getOutput} \]

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

Case $\Lambda, \Gamma_1 \vdash e_1 : \text{String}$  
\(\Lambda, \Gamma_1 \vdash e_2 : \text{TypedVal}\)  
\(\text{invoke}\)  
\[1. \quad \Lambda, \Gamma_1 \vdash e_1 : \text{String} \quad \text{assumption} \]
\[2. \quad \Lambda, \Gamma_1 \vdash e_2 : \text{TypedVal} \quad \text{assumption} \]
\[3. \quad \Gamma_1 \subseteq \Gamma_2 \quad \text{assumption} \]
\[4. \quad \Lambda, \Gamma_2 \vdash e_1 : \text{String} \quad 1, 3, \text{IH} \]
\[5. \quad \Lambda, \Gamma_2 \vdash e_2 : \text{TypedVal} \quad 2, 3, \text{IH} \]
\[6. \quad \Lambda, \Gamma_2 \vdash \text{invoke}(e_1, e_2) : \text{TypedVal} \quad 4, 5, \text{Rule invoke} \]

Case $\Lambda, \Gamma_1 \vdash e_1 : \tau_1 \rightarrow \tau_2$  
\(\Lambda, \Gamma_1 \vdash e_2 : \tau_1\)  
\(\text{call}\)  
\[1. \quad \Lambda, \Gamma_1 \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \text{assumption} \]
\[2. \quad \Lambda, \Gamma_1 \vdash e_2 : \tau_1 \quad \text{assumption} \]
\[3. \quad \Gamma_1 \subseteq \Gamma_2 \quad \text{assumption} \]
\[4. \quad \Lambda, \Gamma_2 \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad 1, 3, \text{IH} \]
\[5. \quad \Lambda, \Gamma_2 \vdash e_2 : \tau_1 \quad 2, 3, \text{IH} \]
\[6. \quad \Lambda, \Gamma_2 \vdash \text{call}(e_1, e_2) : \tau_2 \quad 4, 5, \text{Rule call} \]
Lemma 6 (Substitution). \( \Lambda, \Gamma \cup \{ x : \tau \} \vdash e' : \tau' \land \Lambda, \Gamma \vdash e : \tau \rightarrow \Lambda, \Gamma \vdash [e/x]e' : \tau' \)

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

\[ \begin{array}{c}
\Lambda, \Gamma \cup \{ x : \tau \} \vdash e_x : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \\
\text{Case} \quad \Lambda, \Gamma \cup \{ \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n \} \vdash e_\ell : \tau' \\
\end{array} \]

\[ \Lambda, \Gamma \cup \{ x : \tau \} \vdash (\text{case } e_\ell \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau' \quad (\text{case}) \]

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

\[ \begin{array}{c}
\text{Case} \quad \Lambda, \Gamma \cup \{ x : \tau \} \vdash e_1 : \text{String} \\
\Lambda, \Gamma \cup \{ x : \tau \} \vdash e_2 : \text{TypedVal} \\
\end{array} \]

\[ \Lambda, \Gamma \cup \{ x : \tau \} \vdash \text{invoke}(e_1, e_2) : \text{TypedVal Option} \quad (\text{invoke}) \]

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

\[ \begin{array}{c}
\text{Case} \quad \Lambda, \Gamma \cup \{ x : \tau \} \vdash e_1 : \tau_1 \\
\end{array} \]

\[ \Lambda, \Gamma \cup \{ x : \tau \} \vdash \text{makeTypedVal}(\tau_1, e_1) : \text{TypedVal} \quad (\text{makeTypedVal}) \]

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

\[ \begin{array}{c}
\text{Case} \quad \Lambda, \Gamma \cup \{ x : \tau \} \vdash e_1 : \text{TypedVal} \\
\end{array} \]

\[ \Lambda, \Gamma \cup \{ x : \tau \} \vdash \text{tryCast}(\tau_1, e_1) : \tau_1 \text{ Option} \quad (\text{tryCast}) \]

1. \( \Lambda, \Gamma \cup \{ x : \tau \} \vdash e_1 : \text{TypedVal} \quad \text{assumption} \)
2. \( \Lambda, \Gamma \vdash e : \tau \quad \text{assumption} \)
3. \( \Lambda, \Gamma \vdash [e/x]e_1 : \text{TypedVal} \quad 1, 2, \text{IH} \)
4. \( \Lambda, \Gamma \vdash \text{tryCast}(\tau_1, [e/x]e_1) : \tau_1 \text{ Option} \quad 3, \text{Rule tryCast} \)
5. \( \Lambda, \Gamma \vdash [e/x]\text{tryCast}(\tau_1, e_1) : \tau_1 \text{ Option} \quad 4, \text{definition of } \quad [e/x]\text{tryCast}(\tau_1, e_1) \)
\[
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \ldots \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_n : \tau_n \quad (\text{record})
\]

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

\[
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : (\text{evt : } \tau_1 \text{ Event} \times \text{pols : PolList} \times \text{os : OS} \times \text{vc : VC}) \quad (\text{monitor})
\]

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

\[
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash b : \text{Bool} \quad (\text{boolVal})
\]

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

\[
\text{Case } \Lambda', \Gamma \cup \{x : \tau\} \vdash \ell : \tau'' \quad (\text{location})
\]

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

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

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

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

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

Case \( \Lambda, \Gamma \cup \{ x : \tau \} \vdash e_1 : \tau'' \)
\[ \text{(equality)} \]
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. \( \tau'' \in \{ \text{Int, Bool, String} \} \) 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. \( \Lambda, \Gamma \vdash [e/x]e_1 == [e/x]e_2 : \text{Bool} \) 4, 5, 6, Rule equality
8. \( \Lambda, \Gamma \vdash [e/x](e_1 == e_2) : \text{Bool} \) 7, definition of \([e/x](e_1 == e_2)\)
Case $\Lambda, \Gamma \cup \{x : \tau\} \vdash n : \mathit{Int}$ (intVal)

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

   Case $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \mathit{Bool}$

   - $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau'$
   - $\Lambda, \Gamma \cup \{x : \tau\} \vdash e_3 : \tau'$

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

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

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

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

Case $\Lambda, \Gamma' \cup \{y : \tau'\} \cup \{x : \tau\} \vdash y : \tau'$ (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 $[e/x]y$
   b. $\tau = \tau'$ 2, 4
   c. $\Lambda, \Gamma' \cup \{y : \tau'\} \vdash [e/x]y : \tau'$ 1, 4a, 4b

Result is from 3b and 4c
\[
\frac{
\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \text{Int} \\
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \text{Int} \quad \text{(add)}
}{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 + e_2 : \text{Int}}
\]

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

\[
\frac{
\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau' \text{ Ref} \\
\Lambda, \Gamma \cup \{x : \tau\} \vdash e''' : \tau'''}{\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash e' : \tau' \text{ Ref}} \quad \text{(accessRef)}
\]

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

\[
\frac{
\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{unit} : \text{Unit} \\
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash \text{unit} : \tau' \quad \text{unitVal}
}{\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash \text{unit} : \tau'}
\]

1. \(\tau' = \text{Unit}\) assumption
2. \([e/x]\text{unit} = \text{unit}\) definition of \([e/x]\text{unit}\)
3. \(\Lambda, \Gamma \vdash [e/x]\text{unit} : \tau'\) 1, 2, Rule unitVal

\[
\frac{
\tau' = \tau_{\text{List}} \\
\[e/x][\text{[] : \tau_{\text{List}}} = \text{[] : \tau_{\text{List}}}\] \quad \text{definition of } [e/x][\text{[] : \tau_{\text{List}}}]
}{\Lambda, \Gamma \vdash [e/x][\text{[] : \tau_{\text{List}}} : \tau'] \quad 1, 2, \text{Rule listEmptyVal}}
\]

\[
\frac{
\frac{
\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{\text{List}} \\
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash \text{head}(e_1) : \tau_1 \text{ Option}
}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{head}(e_1) : \tau_{\text{List}}} \quad \text{(head)}
}{\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{\text{List}}} \quad \text{head}
\]

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

\[
\frac{
\frac{
\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{\text{List}} \\
\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash \text{tail}(e_1) : \tau_{\text{List}} \quad \text{tail}
}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{tail}(e_1) : \tau_{\text{List}}} \quad \text{tail}
}{\text{Case } \Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_{\text{List}}} \quad \text{tail}
\]

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

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

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

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

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

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

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

1. $\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{ref } e'' : \tau'' \text{ Ref}$ assumption
2. $\Lambda, \Gamma \vdash e : \tau$ assumption
3. $\Lambda, \Gamma \vdash [e/x]e'' : \tau''$ 1, 2, IH
4. $\Lambda, \Gamma \vdash \text{ref } ([e/x]e'') : \tau'' \text{ Ref}$ 3, Rule createRef
5. $\Lambda, \Gamma \vdash [e/x](\text{ref e''}) : \tau'' \text{ Ref}$ 4, definition of $[e/x](\text{ref e''})$
\[
\begin{align*}
\Lambda, \Gamma \cup \{x : \tau\} & \vdash e_1 : \tau_1 \\
\text{Case} & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau_2 \quad \text{(sequence)} \\
\Lambda, \Gamma \cup \{x : \tau\} & \vdash e_1; e_2 : \tau_2 \\
1. \quad \Lambda, \Gamma \cup \{x : \tau\} & \vdash e_1 : \tau_1 \quad \text{assumption} \\
2. \quad \Lambda, \Gamma \cup \{x : \tau\} & \vdash e_2 : \tau_2 \quad \text{assumption} \\
3. \quad \Lambda, \Gamma & \vdash e : \tau \quad \text{assumption} \\
4. \quad \Lambda, \Gamma & \vdash [e/x]e_1 : \tau_1 \quad 1, 3, \text{IH} \\
5. \quad \Lambda, \Gamma & \vdash [e/x]e_2 : \tau_2 \quad 2, 3, \text{IH} \\
6. \quad \Lambda, \Gamma & \vdash [e/x]e_1; [e/x]e_2 : \tau_2 \quad 4, 5, \text{Rule sequence} \\
7. \quad \Lambda, \Gamma & \vdash [e/x](e_1; e_2) : \tau_2 \quad 6, \text{definition of } [e/x](e_1; e_2) \\
\end{align*}
\]

\[
\begin{align*}
\Lambda, \Gamma \cup \{x : \tau\} & \vdash e_1 : \tau'' \quad \text{Ref} \\
\text{Case} & \quad \Lambda, \Gamma \cup \{x : \tau\} \vdash e_2 : \tau'' \quad \text{(assignment)} \\
\Lambda, \Gamma \cup \{x : \tau\} & \vdash e_1 := e_2 : \text{Unit} \\
1. \quad \Lambda, \Gamma \cup \{x : \tau\} & \vdash e_1 : \tau'' \quad \text{Ref} \quad \text{assumption} \\
2. \quad \Lambda, \Gamma \cup \{x : \tau\} & \vdash e_2 : \tau'' \quad \text{assumption} \\
3. \quad \Lambda, \Gamma & \vdash e : \tau \quad \text{assumption} \\
4. \quad \Lambda, \Gamma & \vdash [e/x]e_1 : \tau'' \quad 1, 3, \text{IH} \\
5. \quad \Lambda, \Gamma & \vdash [e/x]e_2 : \tau'' \quad 2, 3, \text{IH} \\
6. \quad \Lambda, \Gamma & \vdash [e/x]e_1 := \{e/x\}e_2 : \text{Unit} \quad 4, 5, \text{Rule assignment} \\
7. \quad \Lambda, \Gamma & \vdash [e/x](e_1 := e_2) : \text{Unit} \quad 6, \text{definition of } [e/x](e_1 := e_2) \\
\end{align*}
\]

\[
\begin{align*}
\Lambda, \Gamma \cup \{x : \tau, x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} & \vdash e'' : \tau_2 \quad \text{(fun)} \\
\Lambda, \Gamma \cup \{x : \tau\} & \vdash \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e'' : \tau_1 \rightarrow \tau_2 \\
1. \quad e' = \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e'' \quad \text{assumption} \\
2. \quad \Lambda, \Gamma \cup \{x : \tau, x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} & \vdash e'' : \tau_2 \quad \text{assumption} \\
3. \quad \Lambda, \Gamma & \vdash e : \tau \quad \text{assumption} \\
4. \quad \Lambda, \Gamma & \vdash \text{fun } x_1(x_2 : \tau_1) : \tau_2 = [e/x]e'' : \tau_2 \quad 2, 3, \text{IH} \\
5. \quad \Lambda, \Gamma & \vdash \text{fun } x_1(x_2 : \tau_1) : \tau_2 = [e/x]e'' : \tau_1 \rightarrow \tau_2 \quad 4, \text{Rule fun} \\
6. \quad \Lambda, \Gamma & \vdash [e/x]e' : \tau_1 \rightarrow \tau_2 \quad 5, \text{definition of } [e/x]e' \\
\end{align*}
\]

\[
\begin{align*}
\Lambda, \Gamma \cup \{x : \tau\} & \vdash e'' : \text{Event} \quad \text{(setOutput)} \\
\Lambda, \Gamma \cup \{x : \tau\} & \vdash \text{setOutput}(e'') : \text{Bool} \\
1. \quad \Lambda, \Gamma \cup \{x : \tau\} & \vdash e'' : \text{Event} \quad \text{assumption} \\
2. \quad \Lambda, \Gamma & \vdash e : \tau \quad \text{assumption} \\
3. \quad \Lambda, \Gamma & \vdash [e/x]e'' : \text{Event} \quad 1, 2, \text{IH} \\
4. \quad \Lambda, \Gamma & \vdash \text{setOutput}(e'' : \text{Bool}) \quad 3, \text{Rule setOutput} \\
5. \quad \Lambda, \Gamma & \vdash [e/x](\text{setOutput}(e'')) : \text{Bool} \quad 4, \text{definition of } [e/x]\text{setOutput}(e'') \\
\end{align*}
\]
\[
\text{Case } \frac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e_1 : \tau_1 \quad \Lambda, \Gamma \cup \{x : \tau, y : \tau_1\} \vdash e_2 : \tau_2}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{let } y = e_1 \text{ in } e_2 \text{ end } : \tau_2} \quad \text{(let)}
\]

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

\[
\text{Case } \frac{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{outputNotSet}() : \text{Bool}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{outputNotSet}() : \tau'} \quad \text{(outputNotSet)}
\]

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

\[
\text{Case } \frac{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{call}(e_1, e_2) : \tau_2}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{call}(e_1, e_2) : \tau_1 \rightarrow \tau_2} \quad \text{(call)}
\]

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

\[
\text{Case } \frac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \tau'}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \{e''\}_{s(v)} : \tau''} \quad \text{(endLabel)}
\]

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

\[
\text{Case } \frac{\Lambda, \Gamma \cup \{x : \tau\} \vdash e'' : \text{Obligation}}{\Lambda, \Gamma \cup \{x : \tau\} \vdash \text{makeCFG}(e'') : \text{CFG}} \quad \text{(makeCFG)}
\]

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

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

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

Case \( \Lambda, \Gamma \cup \{ x : \tau \} \vdash e_i : \tau_i \) \( i \in \{ 1, \ldots, n \} \) (variant)

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

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

1. \( \tau' = \text{Event Option} \) assumption
2. \( [e/x]\text{getOutput}() = \text{getOutput}() \) definition of \([e/x]\text{getOutput}()\)
3. \( \Lambda, \Gamma \vdash [e/x]\text{getOutput}() : \tau' \) 1, 2, Rule \text{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 \( \Lambda, \Gamma \vdash n : \text{Int} \) (intVal)

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

Case $\Lambda, \Gamma \vdash s : \text{String}$ (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. $\tau = \text{String}$ 1, 2, rule stringVal

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

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

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

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

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

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \vdash e_2 : \tau_1 \\
\end{align*}
\]

(equality)

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash e_1 : \text{Int} \quad \Delta, \Gamma \vdash e_2 : \text{Int} \\
\end{align*}
\]

(add)

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2 \\
\end{align*}
\]

(sequence)

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash e_1 : \text{Bool} \quad \Delta, \Gamma \vdash e_2 : \tau_1 \quad \Delta, \Gamma \vdash e_3 : \tau_1 \\
\end{align*}
\]

(if)

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash e_1 : \text{Bool} \quad \Delta, \Gamma \vdash e_2 : \tau_1 \\
\end{align*}
\]

(while)

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2 \\
\end{align*}
\]

(let)

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

\[
\begin{align*}
\text{Case} & \quad \Delta, \Gamma \vdash \text{ref } e : \tau_1 \text{ Ref} \\
\end{align*}
\]

(createRef)

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

Case $\Lambda, \Gamma \vdash e_1 : \tau_1 \text{ Ref}, \Lambda, \Gamma \vdash e_2 : \tau_1$ (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. $\tau = \text{ unit}$, $\Lambda, \Gamma \vdash e_1 : \tau_1 \text{ Ref}$, $\Lambda, \Gamma \vdash e_2 : \tau_1$ 1, 2, Rule assignment

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

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

Case $\Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \vdash e_2 : \tau_1$ (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. $\tau = \tau_1$, $\Lambda, \Gamma \vdash e_1 : \tau_1$, $\Lambda, \Gamma \vdash e_2 : \tau_1$ 1, 2, Rule listCons

Case $\Lambda, \Gamma \vdash e_1 : \tau_1, \Lambda, \Gamma \vdash e_2 : \tau_1$ (listAppend)

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

Case $\Lambda, \Gamma \vdash e : \tau_1$ (head)

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

Case $\Lambda, \Gamma \vdash e : \tau_1$ (tail)

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

Case $\Lambda, \Gamma \vdash e : \tau_1$ (empty)

1. $\Lambda, \Gamma \vdash \text{empty}(e) : \tau$ assumption
2. $\Lambda, \Gamma \vdash \text{empty}(e) : \tau$ is only derivable by rule empty inspection of typing rules
3. $\tau = \text{ Bool}$, $\Lambda, \Gamma \vdash e : \tau_1$ 1, 2, Rule empty
Case: $\Lambda, \Gamma \vdash e_1 : \tau_1 \cdots \Lambda, \Gamma \vdash e_n : \tau_n$ (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
3. $\tau = (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$, $\Lambda, \Gamma \vdash v_1 : \tau_1 \cdots \Lambda, \Gamma \vdash v_n : \tau_n$ 1, 2, Rule record

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

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

Case: $\Lambda, \Gamma \vdash (\text{init}_i, e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)$ (variant)

1. $\Lambda, \Gamma \vdash (\text{init}_i, e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : \tau$ assumption
2. $\Lambda, \Gamma \vdash (\text{init}_i, e_i : \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) : \tau$ is only derivable by rule variant
3. $\tau = (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)$, $\Lambda, \Gamma \vdash e_i : \tau_i$, $i \in \{1, \ldots, n\}$ 1, 2, Rule variant

Case: $\Lambda, \Gamma \vdash \{\text{case } e' \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n\} : \tau_x$ (case)

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

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

1. $\Lambda, \Gamma \vdash \text{makeTypedVal}(\tau_1, e_1) : \text{TypedVal}$ assumption
2. $\Lambda, \Gamma \vdash \text{makeTypedVal}(\tau_1, e_1) : \text{TypedVal}$ is only derivable by rule makeTypedVal
3. $\tau = \text{TypedVal}$, $\Lambda, \Gamma \vdash e_1 : \tau_1$ 1, 2, Rule makeTypedVal

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

1. $\Lambda, \Gamma \vdash \text{tryCast}(\tau_1, e_1) : \tau_1$ assumption
2. $\Lambda, \Gamma \vdash \text{tryCast}(\tau_1, e_1) : \tau_1$ is only derivable by rule tryCast
3. $\tau = \tau_1$, $\Lambda, \Gamma \vdash e_1 : \text{TypedVal}$ 1, 2, Rule tryCast

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

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

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

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

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

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

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

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

Case $\Lambda, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$ $\Lambda, \Gamma \vdash e_2 : \tau_1$ (call)
1. $\Lambda, \Gamma \vdash \text{call}(e_1, e_2) : \tau$ assumption
2. $\Lambda, \Gamma \vdash \text{call}(e_1, e_2) : \tau$ is only derivable by rule call inspection of typing rules
3. $\tau = \tau_2$, $\Lambda, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$, $\Lambda, \Gamma \vdash e_2 : \tau_1$ 1, 2, Rule call
Lemma 8 (Canonical Forms).
If $\Lambda, \cdot \vdash v : \tau$ then

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

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

Case $\Lambda, \cdot \vdash n : \text{Int}$ (intVal)
1. $v = n$ assumption

Case $\Lambda, \cdot \vdash b : \text{Bool}$ (boolVal)
1. $v = b$ assumption

Case $\Lambda, \cdot \vdash s : \text{String}$ (stringVal)
1. $v = s$ assumption

Case $\Lambda, \cdot \vdash \text{unit} : \text{Unit}$ (unitVal)
1. $v = \text{unit}$ assumption

Case $\Lambda, \cdot \vdash \text{fun} x_1(x_2 : \tau_1) : \tau_2 = e : \tau_2$ (fun)
1. $v = \text{fun} x_1(x_2 : \tau_1) : \tau_2 = e$ assumption

Case $(\Lambda' \cup \{\ell : \tau\}), \Gamma \vdash \ell : \tau_{\text{Ref}}$ (location)
1. $v = \ell$ assumption
Case $\Lambda, \cdot \cup \{ x : \tau \} \vdash x : \tau$ (var)
1. $x$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \text{Bool}$ $\Lambda, \cdot \vdash e_2 : \text{Bool}$ (con)
$\Lambda, \cdot \vdash e_1 \land e_2 : \text{Bool}$
1. $e_1 \land e_2$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \text{Bool}$ $\Lambda, \cdot \vdash e_2 : \text{Bool}$ (dis)
$\Lambda, \cdot \vdash e_1 \lor e_2 : \text{Bool}$
1. $e_1 \lor e_2$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e : \text{Bool}$ (negation)
$\Lambda, \cdot \vdash \neg e : \text{Bool}$
1. $\neg e$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \tau$ $\Lambda, \cdot \vdash e_2 : \tau$ $\tau \in \{ \text{Int, Bool, String} \}$ (equality)
$\Lambda, \cdot \vdash e_1 == e_2 : \text{Bool}$
1. $e_1 == e_2$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \text{Int}$ $\Lambda, \cdot \vdash e_2 : \text{Int}$ (add)
$\Lambda, \cdot \vdash e_1 + e_2 : \text{Int}$
1. $e_1 + e_2$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \tau_1$ $\Lambda, \cdot \vdash e_2 : \tau_2$ (sequence)
$\Lambda, \cdot \vdash e_1 ; e_2 : \tau_2$
1. $e_1 ; e_2$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \text{Bool}$ $\Lambda, \cdot \vdash e_2 : \tau$ $\Lambda, \cdot \vdash e_3 : \tau$ (if)
$\Lambda, \cdot \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau$
1. $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash \text{while}(e_1) \{ e_2 \} : \text{Bool}$ (while)
1. $\text{while}(e_1) \{ e_2 \}$ is not a value so the lemma holds vacuously in this case

Case $\Lambda, \cdot \vdash e_1 : \tau_1$ $\Lambda, \cdot \cup \{ x : \tau_1 \} \vdash e_2 : \tau_2$ (let)
$\Lambda, \cdot \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2$
1. $\text{let } x = e_1 \text{ in } e_2$ is not a value so the lemma holds vacuously in this case
Case $\Lambda, \bullet \vdash e : \tau$ 

\[ \frac{}{\Lambda, \bullet \vdash \text{ref } e : \tau \text{Ref}} \] (createRef)

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

Case $\Lambda, \bullet \vdash e : \tau \text{Ref}$ 

\[ \frac{}{\Lambda, \bullet \vdash !e : \tau} \] (accessRef)

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

Case $\Lambda, \bullet \vdash e_1 : \tau \text{Ref}$ 

\[ \frac{}{\Lambda, \bullet \vdash e_1 := e_2 : \text{unit}} \] (assignment)

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

Case $\Lambda, \bullet \vdash ([]) : \tau \text{List}$ 

\[ \frac{}{\Lambda, \bullet \vdash v = \text{[]} : \tau \text{List}} \] (listEmptyVal)

1. $v = \text{[]}$ assumption

Case $\Lambda, \bullet \vdash e_1 : \tau \Lambda, \bullet \vdash e_2 : \tau \text{List}$ 

\[ \frac{}{\Lambda, \bullet \vdash e_1 :: e_2 : \tau \text{List}} \] (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 so the lemma holds vacuously in this case

Result from 1, 2

Case $\Lambda, \bullet \vdash e_1 : \tau \text{List} \Lambda, \bullet \vdash e_2 : \tau \text{List}$ 

\[ \frac{}{\Lambda, \bullet \vdash e_1 @ e_2 : \tau \text{List}} \] (listAppend)

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

Case $\Lambda, \bullet \vdash e_1 : \tau \text{List}$ 

\[ \frac{}{\Lambda, \bullet \vdash \text{head}(e_1) : \tau} \] (head)

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

Case $\Lambda, \bullet \vdash e_1 : \tau \text{List}$ 

\[ \frac{}{\Lambda, \bullet \vdash \text{tail}(e_1) : \tau \text{List}} \] (tail)

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

Case $\Lambda, \Gamma \vdash e : \tau \text{List}$ 

\[ \frac{}{\Lambda, \Gamma \vdash \text{empty}(e) : \text{Bool}} \] (empty)

1. empty($e_1$) is not a value so the lemma holds vacuously in this case
Case $\Lambda, \bullet \vdash e_i : \tau_i \; \ldots \; \Lambda, \bullet \vdash e_n : \tau_n$ (record)

1. $e_1 = v_1, \ldots, e_n = v_n \Rightarrow \text{assumption}$
   
   $(\ell_1 = e_1, \ldots, \ell_n = e_n) = (\ell_1 = v_1, \ldots, \ell_n = v_n)$
   
   the lemma holds vacuously in this case
   
   Result from 1, 2

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

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

Case $\Lambda, \bullet \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \; i \in \{1, \ldots, n\}$ (variant)

1. $\tau = (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)$ assumption

2. $v = in_{\ell_i} e_1 : \tau$ assumption, 1

3. $i \in \{1, \ldots, n\}$ assumption

4. $e_1 \neq v_1 \Rightarrow v = in_{\ell_i} v_1 : \tau$ 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 $\Lambda, \bullet \vdash e : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \; i \in \{1, \ldots, n\}$ (case)

1. $(\text{case } e \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) : \tau$

   $(\text{case } e \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n) \text{ is not a value so the lemma holds vacuously in this case}$

Case $\Lambda, \bullet \vdash e_1 : \tau'$ (makeTypedVal)

1. $e_1 \neq v_1 \Rightarrow v = \text{makeTypedVal}(\tau', e_1)$ assumption

2. $e_i \neq v_i \Rightarrow v$ is not a value the lemma holds vacuously in this case

Result is from 1, 2

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

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

Case $\Lambda, \bullet \vdash e : \tau$ (endLabel)

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

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

1. $\text{getRT}()$ is not a value so the lemma holds vacuously in this case
Case \( \Lambda, \bullet \vdash e_1 : Obligation \) (makeCFG)

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

Case \( \Lambda, \bullet \vdash e : Event \) (setOutput)

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

Case \( \Lambda, \bullet \vdash outputNotSet() : Bool \) (outputNotSet)

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

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

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

Case \( \Lambda, \bullet \vdash e_1 : \text{Event} \times \text{PolList} \times \text{OS} \times \text{VC} \) (monitor)

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

Case \( \Lambda, \bullet \vdash e_1 : \text{String} \)

\[ \Lambda, \bullet \vdash e_1 \in F.name \]

Case \( \Lambda, \bullet \vdash F[e_1].\text{fun} : \tau_1 \rightarrow \tau_2 \) (invoke)

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

Case \( \Lambda, \bullet \vdash e_1 : \tau_1 \rightarrow \tau_2 \) (call)

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

\[ \square \]

Lemma 9 (Progress). \( \Lambda \vdash (C, e) : \tau \Rightarrow e \) value \( \lor \exists C', e' : (C, e) \rightarrow (C', e') \)

We will instead prove the equivalent statement \( \Lambda \vdash C \text{ ok} \land \bullet \vdash e : \tau \Rightarrow e \) value \( \lor \exists C', e' : (C, e) \rightarrow (C', e') \).

It can be shown that these two statements are equivalent by inversion of rule TConfig.

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

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

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

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

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

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

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

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

Case \[ \Lambda, \bullet \vdash e_1 : Bool, \Lambda, \bullet \vdash e_2 : Bool \] (con)
1. \( \Lambda \vdash C \) \( \text{ok} \)
   \[ \text{assumption} \]
2. \( \Lambda, \bullet \vdash e_1 : Bool \)
   \[ \text{assumption} \]
3. \( \Lambda, \bullet \vdash e_2 : Bool \)
   \[ \text{assumption} \]
4. \( e_1 = v_1 \) or \( C, e_1 \) \( \rightarrow \) \( C', e'_1 \)
   \[ \text{1, 2, IH} \]
5. \( e_2 = v_2 \) or \( C, e_2 \) \( \rightarrow \) \( C', e'_2 \)
   \[ \text{1, 3, IH} \]
6. \( e_1 = v_1 \Rightarrow e_1 \in \{ \text{true, false} \} \)
   \[ \text{2, Lemma Canonical Forms} \]
7. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 \land e_2) \rightarrow (C', e'_1 \land e_2) \)
   \[ \text{Rule andE1} \]
8. \( (e_1 = v_1) \) and \( (C, e_2) \) \( \rightarrow \) \( (C', e'_2) \Rightarrow (C, e_1 \land e_2) \rightarrow (C', v_1 \land e'_2) \)
   \[ \text{Rule andE2} \]
9. \( (e_1 = \text{true} \) and \( e_2 = v_2) \Rightarrow (C, e_1 \land e_2) \rightarrow (C, v_2) \)
   \[ \text{Rule andTrue} \]
10. \( (e_1 = \text{false} \) and \( e_2 = v_2) \Rightarrow (C, e_1 \land e_2) \rightarrow (C, \text{false}) \)
    \[ \text{Rule andFalse} \]
11. \( \exists C', e' : (C, e_1 \land e_2) \rightarrow (C', e') \)
    \[ 7-10 \]

Case \[ \Lambda, \bullet \vdash e_1 : Bool, \Lambda, \bullet \vdash e_2 : Bool \] (dis)
1. This case is analogous to case con.
Case $\Lambda, \bullet \vdash e_1 : Bool$ (negation)

1. $\Lambda \vdash C \text{ ok}$  
   assumption

2. $\Lambda, \bullet \vdash e_1 : Bool$  
   assumption

3. $e_1 = v_1 \lor (C, e_1) \rightarrow (C', e_1')$  
   1, 2, IH

4. $e_1 = v_1 \Rightarrow e_1 \in \{\text{true}, \text{false}\}$  
   2, Lemma Canonical Forms

5. $e_1 = \text{true} \Rightarrow (C, \neg e_1) \rightarrow (C, \text{false})$  
   notTrue

6. $e_1 = \text{false} \Rightarrow (C, \neg e_1) \rightarrow (C, \text{true})$  
   notFalse

7. $(C, e_1) \rightarrow (C', e_1') \Rightarrow (C, \neg e_1) \rightarrow (C', \neg e_1')$  
   notE

8. $\exists C', e' : (C, \neg e_1) \rightarrow (C', e')$  
   5-7
Case $\Lambda, \cdot \vdash e_1 : \tau, \Lambda, \cdot \vdash e_2 : \tau \quad \tau \in \{\text{Int, Bool, String}\}$ (equality)

1. $\Lambda \vdash C \ ok$
   assumption
2. $\Lambda, \cdot \vdash e_1 : \tau$
   assumption
3. $\Lambda, \cdot \vdash e_2 : \tau$
   assumption
4. $\tau \in \{\text{Int, Bool, String}\}$
   assumption
5. $e_1 = v_1 \ or \ (C, e_1) \rightarrow (C', e'_1)$
   Rule eqE1
6. $e_2 = v_2 \ or \ (C, e_2) \rightarrow (C', e'_2)$
   Rule eqE2
7. $e_1 \rightarrow e'_1 \Rightarrow (C, e_1 == e_2) \rightarrow (C', e'_1 == e_2)$
   Lemma Canonical Forms, 2, 9, 9a
8. $e_1 = v_1 \ and \ e_2 \rightarrow e'_2 \Rightarrow (C, e_1 == e_2) \rightarrow (C', v_1 == e'_2)$
   Lemma Canonical Forms, 3, 9, 9a
9. $e_1 = v_1 \ and \ e_2 = v_2 \Rightarrow$
   a. $\tau = \text{Int} \Rightarrow$
   i. $v_1 = n_1$
   (i), (ii), rule eqIntTrue
   ii. $v_2 = n_2$
   (ii), definition of $n$
   iii. $n_1 = n_2 \Rightarrow (C, e_1 == e_2) \rightarrow (C, \text{true})$
   (iii), rule eqIntFalse
   iv. $n_1 \neq n_2 \Rightarrow (C, e_1 == e_2) \rightarrow (C, \text{false})$
   b. $\tau = \text{Bool} \Rightarrow$
   i. $v_1 \in \{\text{true, false}\}$
   (ii), (iii), rule eqBoolTrue
   ii. $v_2 \in \{\text{true, false}\}$
   (iii), rule eqBoolFalse
   c. $\tau = \text{String} \Rightarrow$
   i. $v_1 = s_1$
   (i), (ii), (iii)
   ii. $v_2 = s_2$
   (iii), rule eqStrTrue
   iii. $s_1 = s_2 \Rightarrow (C, e_1 == e_2) \rightarrow (C, \text{true})$
   (iv), rule eqStrFalse
   iv. $s_1 \neq s_2 \Rightarrow (C, e_1 == e_2) \rightarrow (C, \text{false})$
   d. $\exists C', e' : (C, e) \rightarrow (C', e')$
   (i), (ii), (iii), (iv)
10. $e = v \ or \ \exists C', e' : (C, e) \rightarrow (C', e')$
Case \( \Lambda, \bullet \vdash e_1 : \text{Int} \) \( \Lambda, \bullet \vdash e_2 : \text{Int} \) (add)

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

Case \( \Lambda, \bullet \vdash e_1 : \tau_1 \) \( \Lambda, \bullet \vdash e_2 : \tau_2 \) (sequence)

1. \( \Lambda \vdash C \text{ ok} \)  
   assumption
2. \( \Lambda, \bullet \vdash e_1 : \tau_1 \)  
   assumption
3. \( \Lambda, \bullet \vdash e_2 : \tau_2 \)  
   assumption
4. \( e_1 = v_1 \text{ or } (C, e_1) \rightarrow (C', e'_1) \)  
   1, 2, IH
5. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, (e_1 ; e_2)) \rightarrow (C', (e'_1 ; e_2)) \)  
   Rule sequenceE1
6. \( e_1 = v_1 \Rightarrow (C, (e_1 ; e_2)) \rightarrow (C, e_2) \)  
   Rule sequenceE2
7. \( \exists e, e' : (C, e) \rightarrow (C', e') \)  
   3-5

Case \( \Lambda, \bullet \vdash e_1 : \text{Bool} \) \( \Lambda, \bullet \vdash e_2 : \tau \) \( \Lambda, \bullet \vdash e_3 : \tau \) (if)

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

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

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

111
Case $\Lambda, \bullet \vdash e_1 : \tau_1$, $\Lambda, \bullet \vdash e_2 : \tau_2$ (let)

1. $e = (\text{let } x = e_1 \text{ in } e_2 \text{ end})$
   - assumption
2. $\Lambda \vdash C \text{ ok}$
   - 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 $(C, e_1) \rightarrow (C', e'_1)$
   - 2, 3, IH
6. $e_1 = v_1 \Rightarrow (C, e) \rightarrow (C, [v_1/x]e_2)$
   - Rule letValue
7. $(C, e_1) \rightarrow (C', e'_1)$
   - Rule letE

\[ \text{let } x = e_1 \text{ in } e_2 \text{ end} \rightarrow \text{let } x = e'_1 \text{ in } e_2 \text{ end} \]
8. $\exists C', e', (C, e) \rightarrow (C', e')$
   - 5-7

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

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

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

1. $\Lambda \vdash C \text{ ok}$
   - assumption
2. $e_1 : \tau \text{ Ref}$
   - assumption
3. $e_1 = v_1$ or $(C, e_1) \rightarrow (C', e'_1)$
   - 1, 2, IH
4. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{!e}_1) \rightarrow (C', \text{!e}'_1)$
   - 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. $\Lambda \vdash C \text{ ok}$ is only derivable by Rule C-Ok
   - Inspection of Rule C-Ok
8. $C = (M, R, inOb, rt, out, \tau_{out}) \land M : \Lambda$
   - 1, 7, Inspection of Rule C-Ok
9. $M : \Lambda$ is only derivable by Rule TMem
   - Inspection of Rule M:A
10. $M = M' \cup \{ (l, v) \}$
11. $e_1 = v_1 \Rightarrow (C, \text{!e}_1) \rightarrow (C, v)$
12. $\exists C', e', (C, e) \rightarrow (C', e')$
   - 3, 4, 11
Case \( \Lambda, \bullet \vdash e_1 : \tau \) Ref \( \Lambda, \bullet \vdash e_2 : \tau \) (assignment)

1. \( \Lambda \vdash C \) ok assumption

2. \( \Lambda, \bullet \vdash e_1 : \tau \) assumption

3. \( \Lambda, \bullet \vdash e_2 : \tau \) assumption

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

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

6. \( e_1 = v_1 \Rightarrow e_1 = \ell_1 \)

7. \( \Lambda \vdash \ell_1 : \tau \) Ref is only derivable by Rule location Inspection of the Rule location

8. \( \Lambda = \Lambda' \cup \{ \ell_1 : \tau \} \)

9. \( C = (M, R, inOb, rt, out, \tau_{out}) \land M : \Lambda \)

10. \( M : \Lambda \) is only derivable by TMem Inspection of Rule TMem

11. \( M = M' \cup \{(\ell_1, v_{old})\} \)

12. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 := e_2) \rightarrow (C', e'_1 := e_2) \) Rule assignE1

13. \( e_1 = v_1 \land (C, e_2) \rightarrow (C', e'_2) \Rightarrow (C, e_1 := e_2) \rightarrow (C', v_1 := e'_2) \) Rule assignE2

14. \( e_1 = v_1 \land e_2 = v_2 \Rightarrow \)

(1, 2, IH)

(1, 3, IH)

(2, Lemma Canonical Forms)

(2, 6, 7, Inversion of Rule location)

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

(4-5, 12-14)

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

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

Case \( \Lambda, \bullet \vdash e_1 : \tau, \Lambda, \bullet \vdash e_2 : \tau_{List} \) (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 \( (C, e_1) \rightarrow (C', e'_1) \)

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

6. \( (C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 :: e_2) \rightarrow (C', e'_1 :: e_2) \) listPrependE1

7. \( (C, e_2) \rightarrow (C', e'_2) \Rightarrow (C, e_1 :: e_2) \rightarrow (C', v_1 :: e'_2) \) listPrependE2

8. \( (e_1 = v_1 \land e_2 = v_2 \land v_1 :: v_2 = v) \Rightarrow (C, e_1 :: e_2) \rightarrow (C, v) \) Definition of values

9. \( e = v \) or \( \exists C', e', (C, e) \rightarrow (C', e') \)

(6-8)
Case $\Lambda \cdot \vdash e_1 : \tau_{List} \quad \Lambda \cdot \vdash e_2 : \tau_{List}$ (listAppend)

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda \cdot \vdash e_1 : \tau_{List}$
3. $\Lambda \cdot \vdash e_2 : \tau_{List}$
4. $e_1 = v_1 \text{ or } (C, e_1) \rightarrow (C', e'_1)$
5. $e_2 = v_2 \text{ or } (C, e_2) \rightarrow (C', e'_2)$
6. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, e_1 @ e_2) \rightarrow (C', e'_1 @ e_2)$
7. $(e_1 = v_1 \text{ and } (C, e_2) \rightarrow (C', e'_2)) \Rightarrow (C, e_1 @ e_2) \rightarrow (C', v_1 @ e'_2)$
8. $e_1 = v_1 \Rightarrow e_1 = v'_1 :: \cdots :: v'_n \text{ or } e_1 = [] : \tau_{List}$
9. $e_2 = v_2 \Rightarrow e_2 = v'_{n+1} :: \cdots :: v'_m \text{ or } e_2 = [] : \tau_{List}$
10. $(e_1 = [] : \tau_{List} \text{ and } e_2 = v_2) \Rightarrow (C, (e_1 @ e_2)) \rightarrow (C, v_2)$
11. $e_1 = v'_1 :: \cdots :: v'_n \text{ and } e_2 = [] \Rightarrow (C, (e_1 @ e_2)) \rightarrow (C, v'_1)$
12. $(e_1 = v'_1 :: \cdots :: v'_n \text{ and } e_2 = v'_{n+1} :: \cdots :: v'_m) \Rightarrow (C, (e_1 @ e_2)) \rightarrow (C, v'_1 :: \cdots :: v'_m)$
13. $e = v \text{ or } \exists C', e', (C, e) \rightarrow (C', e')$

Case $\Lambda \cdot \vdash e_1 : \tau_{List}$ (head)

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda \cdot \vdash e_1 : \tau_{List}$
3. $e_1 = v \text{ or } (C, e_1) \rightarrow (C', e'_1)$
4. $e_1 = v \Rightarrow e_1 = v_1 :: v_2 \text{ or } e_1 = [] : \tau_{List}$
5. $e_1 = [] : \tau_{List} \Rightarrow (C, \text{head}(e_1)) \rightarrow (C, \text{in}_{\text{none}}(\text{unit}) : \tau \text{ Option})$
6. $e_1 = v_1 :: v_2 \Rightarrow (C, \text{head}(e_1)) \rightarrow (C, \text{in}_{\text{some}}(v_1) : \tau \text{ Option})$
7. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{head}(e_1)) \rightarrow (C', \text{head}(e'_1))$
8. $\exists C', e', (C, e) \rightarrow (C', e')$

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

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda \cdot \vdash e_1 : \tau_{List}$
3. $e_1 = v \text{ or } (C, e_1) \rightarrow (C', e'_1)$
4. $e_1 = v \Rightarrow e_1 = v_1 :: v_2 \text{ or } e_1 = [] : \tau_{List}$
5. $e_1 = [] : \tau_{List} \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C, [] : \tau_{List})$
6. $e_1 = v_1 :: v_2 \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C, v_2)$
7. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C', \text{tail}(e'_1))$
8. $\exists C', e', (C, e) \rightarrow (C', e')$
Case \( \Lambda \vdash e_1 : \tau_{List} \):

\( \Lambda \vdash \text{empty}(e_1) : \text{Bool} \)

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda, \cdot \vdash e_1 : \tau_{List} \)
3. \( e_1 = v \; \text{or} \; (C, e_1) \rightarrow (C', e_1') \)
4. \( e_1 = v \Rightarrow e_1 = v_1 \; \text{or} \; e_1 = \text{[]} : \tau_{List} \)
5. \( e_1 = \text{[]} : \tau_{List} \Rightarrow (C, \text{empty}(e_1)) \rightarrow (C, \text{true}) \)
6. \( e_1 = v_1 \; \text{or} \; v_2 \Rightarrow (C, \text{empty}(e_1)) \rightarrow (C, \text{false}) \)
7. \( (C, e_1) \rightarrow (C', e_1') \Rightarrow (C, \text{tail}(e_1)) \rightarrow (C', \text{tail}(e_1')) \)
8. \( \exists C', e', (C, e) \rightarrow (C', e') \)

Case \( \Lambda, \cdot \vdash e_1 : \tau_1 \; \cdots \; \Lambda, \cdot \vdash e_n : \tau_n \):

1. \( \Lambda \vdash C \text{ ok} \)
2. \( e = (\ell_1 = e_1, \ldots, \ell_n = e_n) \)
3. \( \Lambda, \cdot \vdash e_1 : \tau_1 \; \cdots \; \Lambda, \cdot \vdash e_n : \tau_n \)
4. \( e_1 = v_1 \; \text{or} \; (C, e_1) \rightarrow (C', e_1'), \ldots, e_n = v_n \; \text{or} \; (C, e_n) \rightarrow (C', e_n') \)
5. \( (C, e_1) \rightarrow (C', e_1') \Rightarrow (C, e) \rightarrow (C', (\ell_1 = e_1', \ldots, \ell_n = e_n)) \)
   
   \[ \vdots \]
   
   \[ e_1 = v_{i-1}, \ldots, e_{n-1} = v_{n-1}, (C, e_n) \rightarrow (C', e_n') \Rightarrow (C, e) \rightarrow (C', (\ell_1 = v_1, \ldots, \ell_n = v_n)) \]
   
   \[ \vdots \]
   
6. \( e_1 = v_1, \ldots, e_n = v_n \Rightarrow e = v = (\ell_1 = v_1, \ldots, \ell_n = v_n) \)
7. \( e = v \; \text{or} \; \exists C', e', (C, e) \rightarrow (C', e') \)

Case \( \Lambda, \cdot \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \; i \in \{1, \ldots, n\} \):

1. \( \Lambda \vdash C \text{ ok} \)
2. \( \Lambda, \cdot \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \)
3. \( i \in \{1, \ldots, n\} \)
4. \( e_1 = v_1 \; \text{or} \; (C, e_1) \rightarrow (C', e_1') \)
5. \( e_1 = v_1 \Rightarrow e_1 = (\ell_1 : v_1', \ldots, \ell_n : v_n') \)
6. \( (C, e_1) \rightarrow (C', e_1') \Rightarrow (C, e_1, \ell_i) \rightarrow (C', e_1', \ell_i) \)
7. \( e_1 = (\ell_1 : v_1', \ldots, \ell_n : v_n') \Rightarrow (C, e_1, \ell_i) \rightarrow (C, v_i') \)
8. \( e = v \; \text{or} \; \exists C', e', (C, e) \rightarrow (C', e') \)
Case $\Lambda \cdot \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}$

1. $e = \text{in}_\ell e_i : \ell_i : \tau_1 + \ldots + \ell_n : \tau_n$  
   assumption
2. $\Lambda \vdash C \text{ ok}$  
   assumption
3. $\Lambda, \bullet \vdash e_i : \tau_i$  
   assumption
4. $e_i = v_i$ or $(C, e_i) \rightarrow (C', e'_i) \quad 1, 2, \text{IH}$
5. $(C, e_i) \rightarrow (C', e'_i) \Rightarrow (C, e) \rightarrow (C', e'_i) \quad 1, \text{Rule variantE}$
6. $e = v$ or $\exists C', e', (C, e) \rightarrow (C', e') \quad 3, 4$

Case $\Lambda, \bullet \vdash e_x : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$ $\Lambda, \bullet \cup \{x_1 : \tau_1\} \vdash e_1 : \tau$ $\cdots$ $\Lambda, \bullet \cup \{x_n : \tau_n\} \vdash e_n : \tau$

1. $\Lambda \vdash C \text{ ok}$  
   assumption
2. $e = \text{case } e_x \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n$  
   assumption
3. $\Lambda, \bullet \vdash e_x : (\ell_1 : \tau_1 + \ldots + \ell_n : \tau_n)$  
   assumption
4. $\Lambda, \bullet \cup \{x_1 : \tau_1\} \vdash e_1 : \tau$ $\cdots$ $\Lambda, \bullet \cup \{x_n : \tau_n\} \vdash e_n : \tau$  
   assumption
5. $e_x = v_x$ or $(C, e_x) \rightarrow (C', e'_x)$  
   rule caseE
6. $e_x = v_x \Rightarrow \exists i, v_i : i \in \{1, \ldots, n\} \land e_x = \text{in}_\ell v_1 : \ell_1 : \tau_1 + \ldots + \ell_n : \tau_n$  
   3, Lemma Canonical Forms
7. $(C, e_x) \rightarrow (C', e'_x) \Rightarrow (C, e) \rightarrow (C, e_x) \rightarrow (C', e'_x) \quad 6, \text{rule caseV}$
8. $\exists C', e', (C, e) \rightarrow (C', e') \quad 7-8$

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

1. $\Lambda \vdash C \text{ ok}$  
   assumption
2. $\Lambda, \bullet \vdash e_1 : \tau_1$  
   assumption
3. $e_1 = v_1$ or $(C, e_1) \rightarrow (C', e'_1)$  
   1, 2, IH
4. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{makeTypedVal}(\tau_1, e_1)) \rightarrow (C', \text{makeTypedVal}(\tau_1, e'_1))$  
   Rule makeTypedValE
5. $e_1 = v_1 \Rightarrow \text{makeTypedVal}(\tau_1, e_1) = v$  
   Definition of values
6. $e = v$ or $\exists C', e', (C, e) \rightarrow (C', e') \quad 4-5$

Case $\Lambda, \bullet \vdash e_1 : \text{TypedVal}$

1. $\Lambda \vdash C \text{ ok}$  
   assumption
2. $\Lambda, \bullet \vdash e_1 : \text{TypedVal}$  
   assumption
3. $e_1 = v_1$ or $(C, e_1) \rightarrow (C', e'_1)$  
   1, 2, IH
4. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{tryCast}(\tau_1, e_1)) \rightarrow (C', \text{tryCast}(\tau_1, e'_1))$  
   Rule tryCastE
5. $e_1 = v_1 \Rightarrow e_1 = \text{makeTypedVal}(\tau_2, v_2)$  
   2, Lemma Canonical Forms
6. $e_1 = \text{makeTypedVal}(\tau_2, v_2) \land \tau_1 = \tau_2 \Rightarrow (C, \text{tryCast}(\tau_1, e_1)) \rightarrow (C, \text{in some } v_2 : \tau_1 \text{ Option})$  
   5, rule TryCastVOk
7. $e_1 = \text{makeTypedVal}(\tau_2, v_2) \land \tau_1 \neq \tau_2 \Rightarrow (C, \text{tryCast}(\tau_1, e_1)) \rightarrow (C, \text{in none } \text{ unit } : \tau_1 \text{ Option})$  
   5, rule TryCastVBad
8. $\exists C', e', (C, e) \rightarrow (C', e') \quad 4-7$
Case Λ, • ⊢ e₁ : τ → (endLabel)

1. Λ ⊢ C ok
2. Λ ⊢ C ok is only derivable by Rule C-Ok
3. C = (M, R, inOb, rt, out, τout)
4. Λ, • ⊢ e₁ : τ
5. e₁ = v₁ or (C, e₁) → (C', e'₁)
6. (C, e₁) → (C', e'₁) ⇒ (C, { e }s(v₂)) → (C', { e' }s(v₂))
7. (C, e₁) = v₁ ∧ s = "monitor" ⇒ (C, { e }s(v₂)) → (C, v₁)
8. e₁ = v₁ ∧ s = "monitor" ⇒ (C, { e }s(v₂)) → ((M, R, false, rt, out, τout), v₁)
9. ∃ C', e' : (C, e) → (C', e')

5-8

Case Λ, • ⊢ getRt() : ResList → (getRT)

1. Λ ⊢ C ok
2. Λ ⊢ C ok is only derivable by Rule C-Ok
3. C = (M, R, inOb, rt, out, τout)
4. (C, getRt()) → (C, rt)

Case Λ, • ⊢ e₁ : Obligation → (makeCFG)

1. Λ ⊢ C ok
2. Λ, • ⊢ e₁ : Obligation
3. e₁ = v₁ or (C, e₁) → (C', e'₁)
4. (C, e₁) → (C', e'₁) ⇒ (C, makeCFG(e₁)) → (C', makeCFG(e'₁))
5. e₁ = v₁ ⇒ (C, makeCFG(e₁)) → (C, makeCFG(e'₁))
6. ∃ C', e' : (C, e) → (C', e')

1, 2, Inversion of Rule C-Ok
3, 4, IH

Rule endLabelE
Rule endLabelValue
Rule endLabelMonitor

117
Case \( \Lambda \bullet \vdash e_1 : \text{Event} \)
\( \Lambda \bullet \vdash \text{setOutput}(e_1) : \text{Bool} \)

1. \( \Lambda \vdash C \text{ ok} \)
2. \( e_1 : \text{Event} \)
3. \( \Lambda \vdash C \text{ ok} \)
4. \( C = (M, R, inOb, rt, out, \tau_{out}) \)
5. \( \Lambda, \bullet \vdash \text{out} : \tau_{out} \text{ Event Option} \)
6. \( \text{out} \) is a value
7. \( e_1 = v_1 \text{ or } (C, e_1) \rightarrow (C', e_1') \)
8. \( (C, e_1) \rightarrow (C', e_1') \Rightarrow (C, \text{setOutput}(e_1)) \rightarrow (C', \text{setOutput}(e_1')) \)
9. \( e_1 = v_1 \Rightarrow \)
   a. \( \text{out} = \text{in}_\text{none unit} : \tau_{out} \text{ Event Option} \)
   \( \exists v_2 : \text{out} = \text{in}_\text{some} v_2 : \tau \text{ Event Option} \)
   b. \( \text{out} = \text{in}_\text{some} v_2 : \tau \text{ Event Option} \Rightarrow (C, \text{setOutput}(e_1)) \rightarrow (C, \text{false}) \)
   c. \( \text{out} = \text{in}_\text{none unit} : \tau \text{ Event Option} \Rightarrow \)
      i. \( e_1 = \text{in}_\text{act} v_3 : \text{Event} \)
         \( e_1 = \text{in}_\text{res res} v_4, \text{makeTypedVal}(\tau, v_5) : \text{Event} \)
      ii. \( e_1 = \text{in}_\text{act} v_3 : \text{Event} \Rightarrow (C, \text{setOutput}(e_1)) \rightarrow (C, v_3) \)
         \( (M, R, inOb, rt, in\text{res res} v_4, v_5) : \tau \text{ Event Option}, \tau \text{ out Event Option}, \text{true} \)
      iii. \( e_1 = \text{in}_\text{res res} v_4, \text{makeTypedVal}(\tau, v_5) \land \tau = \tau_{out} \Rightarrow (C, \text{setOutput}(e_1)) \rightarrow (C, \text{false}) \)
         \( (M, R, inOb, rt, in\text{some} \text{ in}_\text{res res} v_1, v_2) : \tau_{out} \text{ Event Option}, \tau_{out}, \text{true} \)
      iv. \( e_1 = \text{in}_\text{res res} v_4, \text{makeTypedVal}(\tau, v_5) \land \tau \neq \tau_{out} \Rightarrow (C, \text{setOutput}(e_1)) \rightarrow (C, \text{false}) \)
         \( (M, R, inOb, rt, in\text{res res} v_1, v_2) : \tau_{out} \text{ Event Option}, \tau_{out}, \text{true} \)
      v. \( \exists C', e' : (C, \text{setOutput}(e_1)) \rightarrow (C, e') \)
   d. \( \text{out} = \text{in}_\text{none unit} : \tau \text{ Event Option} \Rightarrow \exists C', e' : (C, \text{setOutput}(e_1)) \rightarrow (C, e') \)
   e. \( \exists C', e' : (C, \text{setOutput}(e_1)) \rightarrow (C, e') \)
10. \( e_1 = v_1 \Rightarrow \exists C', e' : (C, \text{setOutput}(e_1)) \rightarrow (C, e') \)
11. \( \exists C', e' : (C, \text{setOutput}(e_1)) \rightarrow (C, e') \)

1. assumption
2. assumption
3. Inspection of Rule C-Ok
4. 1, 3, Inversion of Rule C-Ok
5. 1, 3, Inversion of Rule C-Ok
6. 4, definition of syntax
7. 1, IH
8. Rule setOutputE
9. 5, 6, Lemma CF
10. 4, 9, Rule setOutputSet
11. 2, 4, 9, definition of types, Lemma CF
12. 4, 9, c, Rule setOutputAct
13. 4, 9, c, Rule setOutputNotSetResGood
14. 4, 9, c, Rule setOutputNotSetResBad
15. 4, 9, c, Rule setOutputAct
16. i-iv
17. 4, 9, c, Rule
18. a, b, d
19. 9, 9(d)
20. 7, 8, 10
Case \(\Lambda, \bullet \vdash \text{outputNotSet}() : \text{Bool}\) (outputNotSet)
1. \(\Lambda \vdash \text{C ok}\)
2. \(\Lambda \vdash \text{C ok}\) is only derivable by Rule C-Ok
3. \(C = (M, R, inOb, rt, out, \tau_{out})\)
4. \(\Lambda, \bullet \vdash \text{out} : \tau_{out} \text{ Event Option}\)
5. \(\text{out}\) is a value
6. \(\text{out} = \text{in\_none unit} : \tau_{out} \text{ Event Option}\)

\[\exists v_2 : \text{out} = \text{in\_some} v_2 : \tau \text{ Event Option}\]
7. \(\text{out} = \text{in\_none unit} : \tau_{out} \text{ Event Option} \Rightarrow (C, \text{outputNotSet}()) \rightarrow (C, \text{true})\)
8. \(\text{out} = \text{in\_some} v_2 : \tau_{out} \text{ Event Option} \Rightarrow (C, \text{outputNotSet}()) \rightarrow (C, \text{false})\)
9. \(\exists C', e' : (C, \text{outputNotSet}()) \rightarrow (C, e')\)

Case \(\Lambda, \bullet \vdash \text{getOutput}() : \text{Event Option}\) (getOutput)
1. \(\Lambda \vdash \text{C ok}\)
2. \(\Lambda \vdash \text{C ok}\) is only derivable by Rule C-Ok
3. \(C = (M, R, inOb, rt, out, \tau_{out})\)
4. \(\Lambda, \bullet \vdash \text{out} : \tau_{out} \text{ Event Option}\)
5. \(\text{out}\) is a value
6. \(\text{out} = \text{in\_none unit} : \tau_{out} \text{ Event Option}\)

\[\exists v_2 : \text{out} = \text{in\_some} v_2 : \tau \text{ Event Option}\]
7. \(\text{out} = \text{in\_none unit} : \tau_{out} \text{ Event Option} \Rightarrow (C, \text{getOutput}()) \rightarrow (C, \text{in\_none unit} : \text{Event Option})\)
8. \(\text{out} = \text{in\_some} v_2 : \tau_{out} \text{ Event Option} \Rightarrow\)

a. \(v_2 = \text{in\_act} v_3 : \tau_{out} \text{ Event Option} \lor v_2 = \text{in\_res res}(v_4, v_5) : \tau_{out} \text{ Event Option}\)

b. \(v_2 = \text{in\_act} v_3 : \tau_{out} \text{ Event Option} \Rightarrow (C, \text{getOutput}()) \rightarrow (C, \text{in\_some} v_3 : \text{Event}) : \text{Event Option}\)

c. \(v_2 = \text{in\_res res}(v_4, v_5) : \tau_{out} \text{ Event Option} \Rightarrow (C, \text{getOutput}()) \rightarrow (C, \text{in\_some} (\text{in\_res res}(v_4, \text{makeTypedVal}(\tau_{out}, v_5))) : \text{Event}) : \text{Event Option}\)

d. \(\exists C', e' : (C, \text{getOutput}()) \rightarrow (C, e')\)

9. \(\exists C', e' : (C, \text{getOutput}()) \rightarrow (C, e')\)
10. \(\exists C', e' : (C, \text{getOutput}()) \rightarrow (C, e')\)

\[\vdots\]
Case $\Lambda, \bullet \vdash e_1 : (\text{evt} : \tau_1 \text{ Event} \times \text{pols} : \text{Pol}_{\text{List}} \times \text{os} : \text{OS} \times \text{vc} : VC)$ (monitor)

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash e_1 : (\text{evt} : \tau_1 \text{ Event} \times \text{pols} : \text{Pol}_{\text{List}} \times \text{os} : \text{OS} \times \text{vc} : VC)$
3. $e_1 = v_1$ or $(C, e_1) \rightarrow (C', e'_1)$
4. $(C, e_2) \rightarrow (C', e'_2) \rightarrow (C, \text{monitor}(\tau_1, e_1)) \rightarrow (C', \text{monitor}(\tau_1, e'_1))$
5. $\Lambda \vdash C \text{ ok}$ is only derivable by Rule C-ok
6. $C = (M, R, \text{inOb}, rt, out, \tau_{out})$
7. $e_1 = v_1 \Rightarrow (C, \text{monitor}(\tau_1, v_1))$ begin\text{monitor(v1)} $(M, R, \text{true}, \text{inOb}, rt, out, \tau_{out}), \{e_{\text{monitor}(\tau_1, v_1)}\}_{\text{monitor(v1)}}$
8. $\exists C', e', (C, e) \rightarrow (C', e')$

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

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash e_1 : \text{String}$
3. $\Lambda, \bullet \vdash e_2 : \text{TypedVal}$
4. $e_1 = v_1$ or $(C, e_1) \rightarrow (C', e'_1)$
5. $e_1 = v_1 \Rightarrow e_1 = s$
6. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C', \text{invoke}(e'_1, e'_2))$
7. $e_2 = v_2$ or $(C, e_2) \rightarrow (C', e'_2)$
8. $e_1 = s_1 \wedge e_2 \Rightarrow e'_2 \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C', \text{invoke}(e'_1, e'_2))$
9. $e_1 = s_1 \wedge e_2 = v_2 \Rightarrow$

- $(\forall s, f : (s, f) \notin F) \vee (\exists x_1, x_2, \tau_1, \tau_2, e : (s, (\text{fun } x_1 x_2 : \tau_1 : \tau_2 = e)) \in F)$
- $(\forall s, f : (s, f) \notin F) \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C, \text{in} \text{none unit} : \text{TypedVal Option})$
- $(\exists x_1, x_2, \tau_1, \tau_2, e : (s, (\text{fun } x_1 x_2 : \tau_1 : \tau_2 = e)) \in F) \Rightarrow$
  - $v_2 = \text{makeTypedVal}(\tau_3, v'_2)$
  - $\tau_1 = \tau_3 \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C, \text{in} \text{some makeTypedVal}(\tau_2, \text{call}((\text{fun } x_1 x_2 : \tau_1 : \tau_2 = e), v'_2)) : \text{TypedVal Option})$
  - $\tau_1 \neq \tau_3 \Rightarrow (C, \text{invoke}(e_1, e_2)) \rightarrow (C, \text{in} \text{none unit} : \text{TypedVal Option})$

iv. $\exists C', e', (C, e) \rightarrow (C', e')$
- $\exists x_1, x_2, \tau_1, \tau_2, e : (s, (\text{fun } x_1 x_2 : \tau_1 : \tau_2 = e)) \in F) \Rightarrow$
  - $\exists C', e', (C, e) \rightarrow (C', e')$
  - $\exists C', e', (C, e) \rightarrow (C', e')$
  - $\exists C', e', (C, e) \rightarrow (C', e')$
  - $\exists C', e', (C, e) \rightarrow (C', e')$
  - $\exists C', e', (C, e) \rightarrow (C', e')$
- $e_1 = s_1 \wedge e_2 = v_2 \Rightarrow \exists C', e', (C, e) \rightarrow (C', e')$
10. $e_1 = s_1 \wedge e_2 = v_2 \Rightarrow \exists C', e', (C, e) \rightarrow (C', e')$
11. $\exists C', e', (C, e) \rightarrow (C', e')$

- assumption
- assumption
-IH
- Rule invokeE1
- Rule invokeE2
- definition of syntax
- invokeValNotExists
- Lemma Canonical Forms
- Lemma Canonical Forms
- Rule invokeValueExistsOk
- Rule invokeValueExistsBad
Case $\Lambda, \bullet \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Lambda, \bullet \vdash e_2 : \tau_1\) (call)

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash e_1 : \tau_1 \rightarrow \tau_2$
3. $\Lambda, \bullet \vdash e_2 : \tau_1$
4. $e_1 = v_1 \text{ or } (C, e_1) \rightarrow (C', e'_1)$
5. $e_2 = v_2 \text{ or } (C, e_2) \rightarrow (C', e'_2)$
6. $(C, e_1) \rightarrow (C', e'_1) \Rightarrow (C, \text{call}(e_1, e_2)) \rightarrow (C, \text{call}(e'_1, e'_2))$
7. $e_1 = v_1 \Rightarrow \exists x_1, x_2, e_z : e_1 = (\text{fun } x_1(x_2 : \tau_1) = e_z)$
8. $e_1 = v_1 \text{ and } (C, e_2) \rightarrow (C', e'_2) \Rightarrow (C, \text{call}(e_1, e_2)) \rightarrow (C', \text{call}(e'_1, e'_2))$
9. $\Lambda \vdash C \text{ ok}$ is only derivable by rule C-ok
10. $C = (M, (F, \text{pols}, os, vc), \text{inOb}, rt, out, \tau_{out})$
11. $\exists v, \text{inOb} = v$
12. $\text{inOb} \in \{\text{true}, \text{false}\}$
13. $e_1 = v_1 \text{ and } e_2 = v_2 \Rightarrow$
   a. $\exists s : (s, v_1) \in F \Rightarrow$
      i. $\text{inOb} = \text{true} \Rightarrow (C, \text{call}(e_1, e_2)) \xrightarrow{\text{begin}_{x_1(v_2)}}
       ((M, (F, \text{pols}, os, vc), \text{true},
           rt \circ \text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v_1)),
           \text{makeTypedVal}(\tau_2, [v_1/x_1, v_2/x_2]e)) \cdot [] : \text{ResList}, out, \tau_{out}),
       \{[v_1/x_1, v_2/x_2]e\}_{x_1(v_2)}$
   ii. $\text{inOb} = \text{false} \Rightarrow (C, \text{call}(e_1, e_2)) \xrightarrow{\text{begin}_{x_2(v_2)}}
       ((M, (F, \text{pols}, os, vc), \text{false}, rt, \text{innone unit} : \tau_2 \text{ Event Option, } \tau_2), e_{\text{procEvt}})$
   b. $f \notin \text{range}(F) \Rightarrow$
      i. $(\ldots, \text{onTrigger} = v_1, \ldots) \in \text{pols}\land
       v_1 = (\text{fun } x_1(x_2 : \text{Event}) : \text{Unit} = e) \Rightarrow
       (C, \text{call}(e_1, e_2)) \xrightarrow{\text{begin}_{x_1(v_2)}}
       ((M, (F, \text{pols}, os, vc), \text{inOb}, [] : \text{ResList}, out, \tau_{out}),
       \{[v_1/x_1, v_2/x_2]e\}_{x_1(v_2)}$
      ii. $(\ldots, \text{onObligation} = v_1, \ldots) \in \text{pols}\land
       v_1 = (\text{fun } x_1(x_2 : \text{ResList}) : \text{Unit} = e) \Rightarrow
       (C, \text{call}(e_1, e_2)) \xrightarrow{\text{begin}_{x_2(v_2)}}
       ((M, (F, \text{pols}, os, vc), \text{inOb}, [] : \text{ResList}, out, \tau_{out}),
       \{[v_1/x_1, v_2/x_2]e\}_{x_1(v_2)}$
      iii. $(\forall p \in \text{pols} : p =
       (\ldots, \text{onTrigger} = f_1, \text{onObligation} = f_2, \ldots) \Rightarrow
       v_1 \notin \{f_1, f_2\} \Rightarrow (C, \text{call}(e_1, e_2)) \xrightarrow{\text{begin}_{x_1(v_2)}}
       (C, \{[v_2/x_2, v_1/x_1]e\}_{x_1(v_2)}$
14. $\exists C', e' : (C, \text{call}(e_1, e_2)) \rightarrow (C', e')$

13a(i), 13a(ii), 13b(i), 13b(ii), 13b(iii)
Lemma 10 (Monitor Type). $\Lambda, \bullet \vdash v : (\text{evt} : \tau \text{ Event} \times \text{pols} : \text{Pol}_\text{List} \times \text{os} : \text{OS} \times \text{vc} : \text{VC}) \Rightarrow \Lambda, \bullet \vdash e_{\text{monitor}}(\tau, v) : \tau \text{ Event}$

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

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

We will instead prove the equivalent statement:

$(\Lambda \vdash C \text{ ok} \land \Lambda, \bullet \vdash e : \tau \land (C, e) \rightarrow (C', e')) \Rightarrow \exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e' : \tau \land \Lambda \subseteq \Lambda')$.

It can be shown that these two statements are equivalent by inversion of rule TConfig. It is assumed throughout this proof that any $\Lambda$ is a subset of itself.

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

Case $(C, e_1) \rightarrow (C', e'_1)$ (andE)

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

Result is from 7, 9

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

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

Result is from 1, 4

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

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

Result is from 1, 4
Case \((C, e_1) \rightarrow (C', e'_1)\)
\((C, e_1 \lor e_2) \rightarrow (C', e'_1 \lor e_2)\) (orE)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash e_1 \lor e_2 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \text{ Bool}\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash e_2 : \text{ Bool}\) 2, Inversion Lemma
6. \(\tau = \text{ Bool}\) 2, Inversion Lemma
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e'_1 : \text{ Bool} \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
8. \(\Lambda', \bullet \vdash e_2 : \text{ Bool}\) 5, 7, \(\Lambda\)-weakening Lemma
9. \(\Lambda', \bullet \vdash e'_1 \lor e_2 : \tau\) 6-8, Rule or
Result is from 7, 9

Case \((C, e) \leftarrow (C', e'_1)\)
\((C, e \lor e_2) \leftarrow (C', e'_1 \lor e_2)\) (notE)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \neg e_1 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \text{ Bool}\) 2, Inversion Lemma
5. \(\tau = \text{ Bool}\) 2, Inversion Lemma
6. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e'_1 : \text{ bool} \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
7. \(\Lambda', \bullet \vdash \neg e'_1 : \tau\) 5, 6, Rule negation
Result is from 6, 7

Case \((C, e) \leftarrow (C, e)\) (notFalse)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \neg \text{ false} : \tau\) assumption
3. \(\tau = \text{ Bool}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{ true} : \tau\) 3, Rule boolVal
Result is from 1, 4
Case $$(C, \text{true}) \rightarrow (C, \text{false})$$ (notTrue)

1. This case is analogous to case notFalse

Case $$(C, e_1) \rightarrow (C', e'_1)$$ (eqE1)

1. $$\Lambda \vdash C \text{ ok}$$
2. $$\Lambda, \bullet \vdash e_1 == e_2 : \tau$$
3. $$(C, e_1) \rightarrow (C', e'_1)$$
4. $$\Lambda, \bullet \vdash e_1 : \tau_1$$
5. $$\Lambda, \bullet \vdash e_2 : \tau_1$$
6. $$\tau_1 = \{\text{Int, Bool, String}\}$$
7. $$\tau = \text{Bool}$$
8. $$\exists \Lambda': (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')$$
9. $$\Lambda', \bullet \vdash e_2 : \tau_1$$
10. $$\Lambda', \bullet \vdash e'_1 == e_2 : \tau$$
Result is from 8, 10

Case $$(C, v_1 == e_2) \rightarrow (C', v_1 == e'_2)$$ (eqE2)

1. $$\Lambda \vdash C \text{ ok}$$
2. $$\Lambda, \bullet \vdash v_1 == e_2 : \tau$$
3. $$(C, e_2) \rightarrow (C', e'_2)$$
4. $$\Lambda, \bullet \vdash v_1 : \tau_1$$
5. $$\Lambda, \bullet \vdash e_2 : \tau_1$$
6. $$\tau_1 = \{\text{Int, Bool, String}\}$$
7. $$\tau = \text{Bool}$$
8. $$\exists \Lambda': (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e'_2 : \tau_1 \land \Lambda \subseteq \Lambda')$$
9. $$\Lambda', \bullet \vdash v_1 == e'_2 : \tau$$
10. Result is from 8, 10

Case $$(C, n_1 == n_2) \rightarrow (C, \text{true})$$ (eqIntTrue)

1. $$\Lambda \vdash C \text{ ok}$$
2. $$\Lambda, \bullet \vdash n_1 == n_2 : \tau$$
3. $$\tau = \text{Bool}$$
4. $$\Lambda, \bullet \vdash \text{true} : \tau$$
Result is from 1, 4
Case $n_1 \neq n_2$ (eqIntFalse)
1. $\Lambda \vdash C \text{ ok}$ assumption
2. $\Lambda, \bullet \vdash n_1 == n_2 : \tau$ assumption
3. $\tau = \text{ Bool}$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash false : \tau$ 3, Rule boolVal
Result is from 1, 4

Case $(C, s_1 == s_2) \rightarrow (C, true)$ (eqStrTrue)
1. This case is analogous to case eqIntTrue

Case $(C, s_1 == s_2) \rightarrow (C, false)$ (eqStrFalse)
1. This case is analogous to case eqIntFalse

Case $(C, true == b_2) \rightarrow (C, b_2)$ (eqBoolTrue)
1. $\Lambda \vdash C \text{ ok}$ assumption
2. $\Lambda, \bullet \vdash true == b_2 : \tau$ assumption
3. $\tau = \text{ Bool}$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash b_2 : \tau$ 2, 3, Inversion Lemma
Result is from 1, 4

Case $(C, false == b_2) \rightarrow (C, \neg b_2)$ (eqBoolFalse)
1. $\Lambda \vdash C \text{ ok}$ assumption
2. $\Lambda, \bullet \vdash false == b_2 : \tau$ assumption
3. $\tau = \text{ Bool}$ 2, Inversion Lemma
4. $\Lambda, \bullet \vdash b_2 : \text{ Bool}$ 2, Inversion Lemma
5. $\Lambda, \bullet \vdash \neg b_2 : \tau$ 3, 4, Rule negation
Result is from 1, 5

Case $(C, e_1) \rightarrow (C', e'_1)$ (addE1)
1. $\Lambda \vdash C \text{ ok}$ assumption
2. $\Lambda, \bullet \vdash e_1 + e_2 : \tau$ assumption
3. $(C, e_1) \rightarrow (C', e'_1)$ assumption
4. $\Lambda, \bullet \vdash e_1 : \text{ Int}$ 2, Inversion Lemma
5. $\Lambda, \bullet \vdash e_2 : \text{ Int}$ 2, Inversion Lemma
6. $\tau = \text{ Int}$ 2, Inversion Lemma
7. $\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash \bullet e'_1 : \text{ Int} \land \Lambda \subseteq \Lambda')$ 1, 3, 4, IH
8. $\Lambda', \bullet \vdash e_2 : \text{ Int}$ 5, 7, A-weakening Lemma
9. $\Lambda', \bullet \vdash e'_1 + e_2 : \text{ Int}$ 6-8, Rule add
Result is from 7, 9
Case \((C, e_2) \rightarrow (C', e'_2)\) (addE2)

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

\(\text{Result is from 6, 8}\)

Case \((n_1 + \alpha n_2 = n)\) (addValue)

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

\(\text{Result is from 1, 4}\)

Case \((C, e_1) \rightarrow (C', e'_1)\) (sequenceE1)

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

\(\text{Result is from 6, 8}\)

Case \((C, v_1; e_2) \rightarrow (C, e_2)\) (sequenceE2)

1. \(\Lambda \vdash C \text{ ok}\)
2. \(\Lambda, \bullet \vdash v_1; e_2 : \tau\)
3. \(\Lambda, \bullet \vdash e_2 : \tau\)

\(\text{Result is from 1, 3}\)
Case \((C, e_1) \rightarrow (C', e'_1)\) (ifE)

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

Result is from 7, 10

Case \((C, if \ true \ then \ e_2 \ else \ e_3) \rightarrow (C, e_2)\) (ifTrue)

1. \(\Lambda \vdash C \text{ ok}\) 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 \((C, if \ false \ then \ e_2 \ else \ e_3) \rightarrow (C, e_3)\) (ifFalse)

1. \(\Lambda \vdash C \text{ ok}\) 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 \((C, while(e_1) \{e_2\}) \rightarrow (C, if \ e_1 \ then \ (e_2; \ while(e_1) \{e_2\}) \ else \ false)\) (whileE)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda , \bullet \vdash while(e_1) \{e_2\} : \tau\) assumption
3. \(\Lambda , \bullet \vdash e_1 : \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda , \bullet \vdash e_2 : \tau_1\) 2, Inversion Lemma
5. \(\tau = \text{Bool}\) 2, Inversion Lemma
6. \(\Lambda , \bullet \vdash false : \text{Bool}\) Rule boolVal
7. \(\Lambda , \bullet \vdash (e_2; \ while(e_1) \{e_2\}) : \text{Bool}\) 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 \((C, e_1) \rightarrow (C', e'_1)\) (letE)
1. \(\Lambda \vdash C\) ok assumption
2. \(\Lambda, \bullet \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau_1\) assumption
5. \(\Lambda, \{x : \tau_1\} \vdash e_2 : \tau_2\) assumption
6. \(\tau = \tau_2\) assumption
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok}) \land \Lambda' \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')\) 2, Inversion Lemma
8. \(\Lambda', \{x : \tau_1\} \vdash e_2 : \tau_2\) 2, Inversion Lemma
9. \(\Lambda', \bullet \vdash \text{let } x = e'_1 \text{ in } e_2 \text{ end} : \tau\) 2, Inversion Lemma
Result is from 7, 9

Case \((C, \text{let } x = v \text{ in } e_2 \text{ end}) \rightarrow (C, [v/x]e_2)\) (letValue)
1. \(\Lambda \vdash C\) ok assumption
2. \(\Lambda, \bullet \vdash \text{let } x = v \text{ in } e_2 \text{ end} : \tau\) assumption
3. \(\Lambda, \bullet \vdash v : \tau_1\) assumption
4. \(\Lambda, \{x : \tau_1\} \vdash e_2 : \tau_2\) assumption
5. \(\tau = \tau_2\) assumption
6. \(\Lambda, \bullet \vdash [v/x]e_2 : \tau\) assumption
Result is from 1, 6

Case \((C, e_1) \rightarrow (C', e'_1)\) (refE)
1. \(\Lambda \vdash C\) ok assumption
2. \(\Lambda, \bullet \vdash \text{ref } e_1 : \tau\) assumption
3. \((C, e_1) \rightarrow (C', e'_1)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau_1\) assumption
5. \(\tau = \tau_1\) Ref assumption
6. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok}) \land \Lambda' \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')\) 2, Inversion Lemma
7. \(\Lambda', \bullet \vdash \text{ref } e'_1 : \tau\) 2, Inversion Lemma
Result is from 6, 7
Case $\ell \notin \text{dom}(M)$

\[
(M, R, \text{inOb}, rt, \text{out}, \tau_{\text{out}}), \text{ref } v \quad \rightarrow \quad ((M \cup \{\ell, v\}), R, \text{inOb}, rt, \text{out}, \tau_{\text{out}}), \ell
\]

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

Result is from 14, 18, 19

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

\[
(C, !e_1) \quad \rightarrow \quad (C', !e'_1)
\]

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash !e_1 : \tau$
3. $(C, e_1) \rightarrow (C', e'_1)$
4. $\Lambda, \bullet \vdash e_1 : \tau \text{ Ref}$
5. $\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda, \bullet \vdash e'_1 : \tau \text{ Ref} \land \Lambda \subseteq \Lambda')$
6. $\Lambda', \bullet \vdash !e'_1 : \tau$

Result is from 5, 6

Case $(M' \cup \{(\ell, v)\}, \ldots, \text{ref } v) \rightarrow (M' \cup \{(\ell, v\}) \cup \{\ell\})$

1. $\Lambda \vdash (M' \cup \{(\ell, v)\}, \ldots) \text{ ok}$
2. $\Lambda, \bullet \vdash \ell : \tau$
3. $\Lambda, \bullet \vdash v : \tau \text{ Ref}$
4. 3 is only derivable with Rule location
5. $\Lambda = \Lambda' \cup \{(\ell : \tau)\}$
6. $M' \cup \{(\ell, v)\} : \Lambda' \cup \{(\ell : \tau)\}$
7. 6 is only derivable by Rule TMem
8. $\Lambda, \bullet \vdash v : \tau$

Result is from 1, 8
Case \((C, e_1) \rightarrow (C, e'_1)\) (assignE1)

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

Result is from 7, 9

Case \((C, e_2) \rightarrow (C', e'_2)\) (assignE2)

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

Result is from 7, 9

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

1. \(\Lambda \vdash (M' \cup \{(\ell, v)\}, R, \text{inOb, rt, out, } \tau_{out}) \text{ ok}\)
2. \(\Lambda, \cdot \vdash \ell := v' : \tau\)
3. \(M' \cup \{(\ell, v)\} : \Lambda\)
4. \(\Lambda \vdash R \text{ ok}\)
5. \(\Lambda, \cdot \vdash \text{inOb} : \text{Bool}\)
6. \(\Lambda, \cdot \vdash \text{rt} : \text{ResList}\)
7. \(\Lambda, \cdot \vdash \text{out} : \tau_{out} \text{ Event Option}\)
8. \(\Lambda, \cdot \vdash \ell : \tau' \text{ Ref}\)
9. \(\Lambda, \cdot \vdash v' : \tau'\)
10. \(\tau = \text{Unit}\)
11. 3 is only derivable by Rule TMem
12. \(\Lambda = \Lambda' \cup \{\ell : \tau'\}\)
13. \(\forall i \in \{1, \ldots, n\}. \Lambda, \cdot \vdash v_i : \tau_i\)
14. \(\forall i \in \{1, \ldots, n-1\}. \Lambda, \cdot \vdash v_i : \tau_i\)
15. \(M' \cup \{(\ell, v')\} : \Lambda\)
16. \(\Lambda \vdash (M' \cup \{(\ell, v')\}, R, \text{inOb, rt, out, } \tau_{out}) \text{ ok}\)
17. \(\Lambda, \cdot \vdash \text{unit} : \tau\)

Result is from 16, 17

130
\[
\begin{align*}
\text{Case } (C, e_1) & \quad \rightarrow 
(C', e'_1) \quad (\text{listPrependE1}) \\
\frac{(C, e_1 :: e_2)}{(C', e'_1 :: e_2)} & \quad \rightarrow 
(C', e'_1 :: e_2) \quad \text{(assumption)}
\end{align*}
\]

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

Result is from 7, 9

\[
\begin{align*}
\text{Case } (C, e_2) & \quad \rightarrow 
(C', e'_2) \quad (\text{listPrependE2}) \\
\frac{(C, v_1 :: e_2)}{(C', v'_1 :: e'_2)} & \quad \rightarrow 
(C', v'_1 :: e'_2) \quad \text{(assumption)}
\end{align*}
\]

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

Result is from 7, 9

\[
\begin{align*}
\text{Case } (C, e_1) & \quad \rightarrow 
(C', e'_1) \quad (\text{listAppendE1}) \\
\frac{(C, e_1 @ e_2)}{(C', e'_1 @ e_2)} & \quad \rightarrow 
(C', e'_1 @ e_2) \quad \text{(assumption)}
\end{align*}
\]

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

Result is from 7, 9
Case \((C, e_2) \rightarrow (C', e'_2)\) (listAppendE2)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash v_1 \circ e_2 : \tau\) assumption
3. \((C, e_2) \rightarrow (C', e'_2)\) assumption
4. \(\Lambda, \bullet \vdash v_1 : \tau'_\text{ List}\) 2, Inversion Lemma
5. \(\Lambda, \bullet \vdash e_2 : \tau'_\text{ List}\) 2, Inversion Lemma
6. \(\tau = \tau'_\text{ List}\) 2, Inversion Lemma
7. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash \bullet \vdash e'_2 : \tau'_\text{ List} \land \Lambda \subseteq \Lambda')\) 1,3, 5, IH
8. \(\Lambda', \bullet \vdash v_1 : \tau'_\text{ List}\) 4, 7, A-weakening Lemma
9. \(\Lambda', \bullet \vdash v_1 \circ e'_2 : \tau\) 6-8, Rule listAppend

Result is from 7, 9

Case \((C, \emptyset : \tau'\text{ List} @ v_3) \rightarrow (C, v_3)\) (appendNil)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash (\emptyset : \tau'\text{ List} @ v_3) : \tau\) assumption
3. \(\tau = \tau'_\text{ List}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash v_3 : \tau\) 2, 3, Inversion Lemma

Result is from 1, 4

Case \((C, (v_1 :: v_2) @ v_3) \rightarrow (C, v_1 :: (v_2 @ v_3))\) (appendCons)

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

Result is from 1, 9

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

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

Result is from 6, 7
Case \((C,head(v_1 :: \cdots :: [] : \tau'_\text{List})) \rightarrow (C,in_{\text{some}}(v_1) : \tau' \text{ Option})\) (listHeadCons)

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

Result is from 1, 6

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

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

Result is from 1, 5

Case \((C,e_1) \rightarrow (C',e'_1)\) (listTailE) (listTailCons)

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

Result is from 6, 7

Case \((C,tail(v_1 :: v_2)) \rightarrow (C,v_2)\) (listTailNil)

1. \(\Lambda \vdash C \text{ ok}\) assumption
2. \(\Lambda, \cdot \vdash tail(v_1 :: v_2) : \tau\) assumption
3. \(\Lambda, \cdot \vdash (v_1 :: v_2) : \tau\_\text{List}\) assumption
4. \(\tau = \tau\_\text{List}\) 2, Inversion Lemma
5. \(\Lambda, \cdot \vdash v_2 : \tau\) 3, 4, Inversion Lemma

Result is from 1, 5

Case \((C,tail([] : \tau'_\text{List})) \rightarrow (C,[]) : \tau'_\text{List}\) (listTailNil)

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

Result is from 1, 4
Case \((C, e_1) \rightarrow (C', e'_1)\)

\((C, \text{empty}(e_1)) \rightarrow (C', \text{empty}(e'_1))\) (listEmptyE)

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

Result is from 6, 7

Case \((C, \text{empty}([])) \rightarrow (C, \text{true})\) (listEmptyNil)

1. \(\Lambda \vdash C \text{ ok} \) assumption
2. \(\Lambda, \bullet \vdash \text{empty}([]) : \tau_{\text{List}}'\) assumption
3. \(\tau = \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{true} : \tau\) 3, Rule boolVal

Result is from 1, 4

Case \((C, \text{empty}(v_1 :: v_2)) \rightarrow (C, \text{false})\) (listEmptyCons)

1. \(\Lambda \vdash C \text{ ok} \) assumption
2. \(\Lambda, \bullet \vdash \text{empty}(v_1 :: v_2) : \tau\) assumption
3. \(\tau = \text{Bool}\) 2, Inversion Lemma
4. \(\Lambda, \bullet \vdash \text{false} : \tau\) 3, Rule boolVal

Result is from 1, 4

Case \(\forall j(1 \leq j < i). e_j = v_j \quad (C, e_i) \rightarrow (C', e'_i) \quad i \in \{1, \ldots, n\}\) (recordE)

\((C, (l_1 = e_1, \ldots, l_n = e_n)) \rightarrow (C', (l_1 = e_1, \ldots, l_i = e'_i, \ldots, l_n = e_n))\)

1. \(\Lambda \vdash C \text{ ok} \) assumption
2. \(\Lambda, \bullet \vdash (l_1 = e_1, \ldots, l_n = e_n) : \tau\) assumption
3. \((C, e_i) \rightarrow (C', e'_i)\) assumption
4. \(\Lambda, \bullet \vdash e_1 : \tau_1 \land \ldots \land \Lambda, \bullet \vdash e_n : \tau_n\) assumption
5. \(\tau = (\tau_1 \times \cdots \times \tau_n)\) 2, Inversion Lemma
6. \(\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda' \vdash e'_1 : \tau_1 \land \Lambda \subseteq \Lambda')\) 1, 3, 4, IH
7. \(\Lambda', \bullet \vdash e_1 : \tau_1 \land \ldots \land \Lambda', \bullet \vdash e_n : \tau_n\) 4, 6, A-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 $\frac{(C, e_1) \longrightarrow (C', e'_1)}{(C, e_1, \ell_i) \longrightarrow (C', e'_1, \ell_i)}$ (projectionE)

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash e_1, \ell_i : \tau$
3. $(C, e_1) \longrightarrow (C', e'_1)$
4. $\Lambda, \bullet \vdash e_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n)$
5. $\tau = \tau_i$
6. $i \in \{1, \ldots, n\}$
7. $\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e'_1 : (\ell_1 : \tau_1 \times \cdots \times \ell_n : \tau_n) \land \Lambda \subseteq \Lambda')$
8. $\Lambda', \bullet \vdash e'_1, \ell_i : \tau$

Result is from 2, 3, 4, IH

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

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash (\ell_1 = v_1, \ldots, \ell_n = v_n), \ell_i : \tau$
3. $\Lambda, \bullet \vdash (\ell_1 = v_1, \ldots, \ell_n = v_n) : (\ell_1 : \tau_1, \ldots, \ell_n : \tau_n)$
4. $\tau = \tau_i$
5. $\Lambda, \bullet \vdash v_i : \tau$

Result is from 1, 5

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

1. $\Lambda \vdash C \text{ ok}$
2. $\Lambda, \bullet \vdash (in_{\ell_i}, e_i : \tau') : \tau$
3. $(C, e_i) \longrightarrow (C', e'_i)$
4. $i \in \{1, \ldots, n\}$
5. $\Lambda, \bullet \vdash e_i : \tau_i$
6. $\tau = \ell_1 : \tau_1 + \cdots + \ell_n : \tau_n$
7. $\tau' = \tau$
8. $\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land \Lambda', \bullet \vdash e'_i : \tau_i \land \Lambda \subseteq \Lambda')$
9. $\Lambda', \bullet \vdash (in_{\ell_i}, e'_i : \tau') : \tau$

Result is from 8, 9
Case \((C, e_c) \rightarrow (C', e'_c)\)

\[
(C, \text{case } e_c \text{ of } \ell_1 x_1 \mapsto e_1, \cdots, \ell_n x_n \mapsto e_n) \rightarrow \text{ (caseE)} \quad (C', \text{case } e'_c \text{ of } \ell_1 x_1 \mapsto e_1, \cdots, \ell_n x_n \mapsto e_n)
\]

1. \(? \vdash C \text{ ok}\)  
   assumption
2. \(? \cdot \vdash \text{ (case } e_c \text{ of } \ell_1 x_1 \Rightarrow e_1, \cdots, \ell_n x_n \Rightarrow e_n) : \tau\)  
   assumption
3. \((C, e_c) \rightarrow (C', e'_c)\)  
   assumption
4. \(? \cdot \vdash e_c : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n)\)  
   2, Inversion Lemma
5. \(? \cdot \vdash \{x_1 : \tau_1\} \vdash e_1 : \tau, \cdots, \{x_1 : \tau_1\} \vdash e_n : \tau\)  
   2, Inversion Lemma
6. \(? \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \land \Lambda' \cdot \vdash e'_c : (\ell_1 : \tau_1 + \cdots + \ell_n : \tau_n) \land \Lambda \subseteq \Lambda')\)  
   1, 3, 4, IH
7. \(? \cdot \vdash \text{ (case } e'_c \text{ of } \ell_1 x_1 \Rightarrow e_1, \cdots, \ell_n x_n \Rightarrow e_n) : \tau\)  
   5, 6, A-weakening
8. \(? \cdot \vdash \text{ (case } e'_c \text{ of } \ell_1 x_1 \Rightarrow e_1, \cdots, \ell_n x_n \Rightarrow e_n) : \tau\)  
   6, 7, Rule Case

\[i \in \{1, \ldots, n\}\]

\[
(C, \text{case } \text{ in } i x_i : \tau' \text{ of } \ell x_1 \Rightarrow e_1, \cdots, \ell x_n \Rightarrow e_n) \rightarrow \text{ (caseV)} \quad (C', \text{case } \text{ in } i x_i e_i)\]

1. \(? \vdash C \text{ ok}\)  
   assumption
2. \(? \cdot \vdash \text{ (case } \text{ in } i x_i : \tau' \text{ of } \ell x_1 \Rightarrow e_1, \cdots, \ell x_n \Rightarrow e_n) : \tau\)  
   assumption
3. \((C, e_i) \rightarrow (C', e'_i)\)  
   assumption
4. \(? \cdot \vdash e_i : \tau_i\)  
   2, Inversion Lemma
5. \(? \cdot \vdash \{x_i : \tau_i\} \vdash e_i : \tau\)  
   2, Inversion Lemma
6. \(? \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \land \Lambda' \cdot \vdash e'_i : \tau_i \land \Lambda \subseteq \Lambda')\)  
   1, 3, 4, IH
7. \(? \cdot \vdash \text{ (case } e'_i \text{ of } \ell x_i \Rightarrow e_i) : \tau\)  
   5, 6, Rule makeTypedVal

\[
(C, \text{makeTypedVal}(\tau_1, e_1)) \rightarrow \text{ (makeTypedValE)} \quad (C', \text{makeTypedVal}(\tau_1, e'_1))\]

1. \(? \vdash C \text{ ok}\)  
   assumption
2. \(? \cdot \vdash \text{ makeTypedVal}(\tau_1, e_1) : \tau\)  
   assumption
3. \((C, e_1) \rightarrow (C', e'_1)\)  
   assumption
4. \(? \cdot \vdash e_1 : \tau_i\)  
   2, Inversion Lemma
5. \(? \gamma = \text{TypedVal}\)  
   2, Inversion Lemma
6. \(? \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \land \Lambda' \cdot \vdash e'_i : \tau_i \land \Lambda \subseteq \Lambda')\)  
   1, 3, 4, IH
7. \(? \cdot \vdash \text{ makeTypedVal}(\tau_1, e'_1) : \tau\)  
   5, 6, Rule makeTypedVal

\[
(C, \text{tryCast}(\tau_1, e_1)) \rightarrow \text{ (tryCastE)} \quad (C', \text{tryCast}(\tau_1, e'_1))\]

1. \(? \vdash C \text{ ok}\)  
   assumption
2. \(? \cdot \vdash \text{ tryCast}(\tau_1, e_1) : \tau\)  
   assumption
3. \((C, e_1) \rightarrow (C', e'_1)\)  
   assumption
4. \(? \cdot \vdash e_1 : \text{TypedVal}\)  
   2, Inversion Lemma
5. \(? \gamma = \text{Option}\)  
   2, Inversion Lemma
6. \(? \exists \Lambda' : (\Lambda' \vdash C' \text{ ok } \land \Lambda' \cdot \vdash e'_i : \text{TypedVal} \land \Lambda \subseteq \Lambda')\)  
   1, 3, 4, IH
7. \(? \cdot \vdash \text{ tryCast}(\tau_1, e'_1) : \tau\)  
   5, 6, Rule tryCast

136
Case $(C, \text{tryCast}(\tau', \text{makeTypedVal}(\tau', v))) \rightarrow (C, \text{in}_{\text{some}}(v) : \tau' \ Option)$ (tryCastVOk)

1. $\Lambda \vdash C \ \text{ok}$ assumption
2. $\Lambda, \cdot \vdash \text{tryCast}(\tau', \text{makeTypedVal}(\tau', v)) : \tau$ assumption
3. $\Lambda, \cdot \vdash \text{makeTypedVal}(\tau', v) : \text{TypedVal}$
   2, Inversion Lemma
4. $\tau = \tau' \ Option$ 2, Inversion Lemma
5. $\Lambda, \cdot \vdash v : \tau'$ 3, Inversion Lemma
6. $\Lambda, \cdot \vdash (\text{in}_{\text{some}}(v) : \tau' \ Option) : \tau$ 4, 5, Def of types, Rule variant

Result is from 1, 6

Case $\tau_1 \neq \tau_2$ (tryCastVBad)

$(C, \text{tryCast}(\tau_1, \text{makeTypedVal}(\tau_2, v))) \rightarrow (C, \text{in}_{\text{none}}(\text{unit}) : \tau_1 \ Option)$

1. $\Lambda \vdash C \ \text{ok}$ assumption
2. $\Lambda, \cdot \vdash \text{tryCast}(\tau_1, \text{makeTypedVal}(\tau_2, v)) : \tau$ assumption
3. $\tau = \tau_1 \ Option$ 2, Inversion Lemma
4. $\Lambda, \cdot \vdash \text{unit} : \text{Unit}$ Rule unitVal
5. $\Lambda, \cdot \vdash (\text{in}_{\text{none}}(\text{unit}) : \tau_1 \ Option) : \tau$ 3, 4, Def of types, Rule variant

Result is from 1, 5

Case $(C, e_1) \rightarrow (C', e'_1)$ (endLabelE)

$(C, \{ e_1 \}_{s(v)}) \rightarrow (C', \{ e'_1 \}_{s(v)})$

1. $\Lambda \vdash C \ \text{ok}$ assumption
2. $\Lambda, \cdot \vdash \{ e_1 \}_{s(v)} : \tau$ assumption
3. $(C, e_1) \rightarrow (C', e'_1)$ assumption
4. $\exists \Lambda' : (\Lambda' \vdash C \ \text{ok} \wedge \Lambda', \cdot \vdash e'_1 : \tau \wedge \Lambda' \subseteq \Lambda')$ 2, Inversion Lemma
5. $\Lambda', \cdot \vdash \{ e'_1 \}_{s(v)} : \tau$ 1, 3, 4, IH

Result is from 5, 6

Case $s \neq \text{"monitor"}$ (endLabelValue)

$(C, \{ v_1 \}_{s(v_2)}) \xrightarrow{\text{end}_{s(v_2)}; v_1} (C, v_1)$

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

Result is from 1, 3
Case $s = \text{"monitor\"}$ (endLabelValueMonitor)

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

Result is from 3, 9

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

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

Result is from 1, 4

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

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

Result is from 6, 7

Case $g = \text{makeCFG}_\alpha(v)$ (makeCFGValue)

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

Result is from 1, 6
\[
\begin{align*}
\text{Case} \quad (C, e_1) \rightarrow (C', e_1') \quad (\text{setOutputE}) \\
\text{(C, setOutput(e_1))} \rightarrow (C', \text{setOutput}(e_1')) \quad (\text{setOutputE})
\end{align*}
\]

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

2. \(\Lambda, \bullet \vdash \text{setOutput}(e_1) : \tau\)  
   assumption

3. \((C, e_1) \rightarrow (C', e_1')\)  
   assumption

4. \(\Lambda, \bullet \vdash e_1 : \text{Event}\)  
   2, Inversion Lemma

5. \(\tau = \text{Bool}\)  
   2, Inversion Lemma

6. \(\exists \lambda' : (\lambda' \vdash C' \text{ ok} \land \lambda' \vdash e_1' : \text{Event} \land \lambda \subseteq \lambda')\)  
   1, 3, 4, IH

7. \(\lambda', \bullet \vdash \text{setOutput}(e_1') : \tau\)  
   5, 6, Rule setOutput

Result is from 6, 7

\[
\begin{align*}
\text{Case} \quad \text{(out = in\_some(e) : \tau Event Option)} \quad (\text{setOutputSet})
\end{align*}
\]

1. \(\Lambda \vdash (\ldots, \text{out}, \tau_{out}) \text{ ok}\)  
   assumption

2. \(\Lambda, \bullet \vdash \text{setOutput}(v) : \tau\)  
   assumption

3. \(\tau = \text{Bool}\)  
   2, Inversion Lemma

4. \(\Lambda, \bullet \vdash \text{false} : \tau\)  
   3, Rule boolVal

Result is from 1, 4

\[
\begin{align*}
\text{Case} \quad \text{((M, R, inOb, rt, in\_none(unit) : \tau_{out} Event Option, \tau_{out}), setOutput(in\_act(v) : Event))} \rightarrow ((M, R, inOb, rt, in\_some(in\_act(v) : \tau_{out} Event)) : \tau_{out} Event Option, \tau_{out}), \text{true}) \quad (\text{setOutputNotSetAct})
\end{align*}
\]

1. \(\Lambda \vdash (M, R, inOb, rt, in\_none(unit) : \tau_{out} Event Option, \tau_{out}) \text{ ok}\)  
   assumption

2. \(\Lambda, \bullet \vdash \text{setOutput}(in\_act(v) : \text{Event}) : \tau\)  
   assumption

3. \(\Lambda, \bullet \vdash (in\_act(v) : \text{Event}) : \text{Event}\)  
   2, Inversion Lemma

4. \(\tau = \text{Bool}\)  
   2, Inversion Lemma

5. \(\Lambda, \bullet \vdash v : \text{Act}\)  
   3, Inversion Lemma

6. \(M : \Lambda\)  
   1, C-Inversion Lemma

7. \(\Lambda \vdash R \text{ ok}\)  
   1, C-Inversion Lemma

8. \(\Lambda, \bullet \vdash \text{inOb} : \text{Bool}\)  
   1, C-Inversion Lemma

9. \(\Lambda, \bullet \vdash \text{rt} : \text{ResList}\)  
   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. \(\Lambda \vdash (M, R, inOb, rt, out, \tau_{out}) \text{ ok}\)  
   assumption

13. \(\Lambda, \bullet \vdash \text{true} : \tau\)  
   6-9, 11, 12, Rule C-ok

14. \(\Lambda, \bullet \vdash \text{true} : \tau\)  
   4, Rule boolVal
Case\(\text{((M,R,inOb,rt,in\text{\_none\{}unit\text{\}}}:\tau_{\text{out}}\text{ Event Option},\tau_{\text{out}}\text{},true)}\) \(\rightarrow\)(\text{outputNotSetTrue})

\begin{align*}
1. & \Lambda \vdash (M,R,inOb,rt,in\text{\_none\{}unit\text{\}}:\tau_{\text{out}}\text{ Event Option},\tau_{\text{out}}\text{}) \text{ \text{\_ok} \hspace{1cm} assumption} \\
2. & \Lambda,\cdot \vdash \text{setOutput}(in\text{\_res\}}(res(v_1,\text{makeTypedVal}(\tau_{\text{out}},v_2))):\text{Event}) : \tau \hspace{1cm} \text{2, Inversion Lemma} \\
3. & \Lambda,\cdot \vdash (in\text{\_res\}}(res(v_1,\text{makeTypedVal}(\tau_{\text{out}},v_2))):\text{Event}) : \text{Event} \hspace{1cm} 2, \text{Inversion Lemma} \\
4. & \tau = \text{Bool} \hspace{1cm} \text{3, Inversion Lemma} \\
5. & \Lambda,\cdot \vdash \text{res}(v_1,\text{makeTypedVal}(\tau_{\text{out}},v_2)) : \text{Res} \hspace{1cm} 5, \text{Inversion Lemma} \\
6. & \Lambda,\cdot \vdash v_1 : \text{Act} \hspace{1cm} 5, \text{Inversion Lemma} \\
7. & \Lambda,\cdot \vdash \text{makeTypedVal}(\tau_{\text{out}},v_2) : \text{TypedVal} \hspace{1cm} 7, \text{Inversion Lemma} \\
8. & \Lambda,\cdot \vdash v_2 : \tau_{\text{out}} \hspace{1cm} 1, \text{C-Inversion Lemma} \\
9. & M : \Lambda \hspace{1cm} 1, \text{C-Inversion Lemma} \\
10. & \Lambda \vdash R \text{ \text{\_ok} \hspace{1cm} \text{Def of types}} \\
11. & \Lambda,\cdot \vdash \text{inOb} : \text{Bool} \hspace{1cm} 1, \text{C-Inversion Lemma} \\
12. & \Lambda,\cdot \vdash \text{rt} : \text{Res\_List} \hspace{1cm} 1, \text{C-Inversion Lemma} \\
13. & \Lambda,\cdot \vdash \text{res}(v_1, v_2) : \tau_{\text{out}} \text{ Res} \\
14. & \Lambda,\cdot \vdash (in\text{\_res\}}(res(v_1,v_2))):\tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event} \\
15. & \Lambda,\cdot \vdash (in\text{\_some\}}(in\text{\_res\}}(res(v_1,v_2))):\tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event Option} \\
16. & \text{Let out} = \text{in\_some\}}(in\text{\_res\}}(res(v_1,v_2))):\tau_{\text{out}} \text{ Event}) \\
17. & \Lambda \vdash (M,R,inOb,rt,out,\tau_{\text{out}}) \text{ \text{\_ok} \hspace{1cm} 9-12, 15, 16, Rule C-ok} \\
18. & \Lambda,\cdot \vdash \text{true} : \tau \hspace{1cm} 4, \text{Rule boolVal} \\
\text{Result is from 17, 18}
\end{align*}

Case\(\text{((\ldots, out,\tau_{\text{out}}\text{},false)}\) \(\rightarrow\)\(\text{outputNotSetResBad}\)

\begin{align*}
1. & \Lambda \vdash (\ldots, out,\tau_{\text{out}}) \text{ \text{\_ok} \hspace{1cm} assumption} \\
2. & \Lambda,\cdot \vdash \text{setOutput}(in\text{\_res\}}(res(v_1,\text{makeTypedVal}(\tau',v_2))):\text{Event}) : \tau \hspace{1cm} 2, \text{Inversion Lemma} \\
3. & \tau = \text{Bool} \hspace{1cm} 3, \text{Rule boolVal} \\
4. & \Lambda,\cdot \vdash \text{false} : \tau \\
\text{Result is from 1, 4}
\end{align*}

Case\(\text{((\ldots, out,\tau_{\text{out}}\text{})}, output\_\text{NotSet()}\) \(\rightarrow\)\(\text{outputNotSetTrue}\)

\begin{align*}
1. & \Lambda \vdash (\ldots, out,\tau_{\text{out}}) \text{ \text{\_ok} \hspace{1cm} assumption} \\
2. & \Lambda,\cdot \vdash output\_\text{NotSet()} : \tau \hspace{1cm} \text{assumption} \\
3. & \tau = \text{Bool} \hspace{1cm} 2, \text{Inversion Lemma} \\
4. & \Lambda,\cdot \vdash \text{true} : \tau \hspace{1cm} 3, \text{Rule boolVal} \\
\text{Result is from 1, 4}
\end{align*}
Case \( \text{out} = \text{in}_{\text{some}}(e_1) : \tau_{\text{out}} \text{ Event Option} \)

\[
\frac{((\ldots, \text{out}, \tau_{\text{out}}, \text{outputNotSet}()), \ldots, \text{out}, \tau_{\text{out}}, \text{false})}{((\ldots, \text{out}, \tau_{\text{out}}), \text{outputNotSetFalse})}
\]

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

Result is from 1, 4

Case \( \text{out} = \text{in}_{\text{some}}(\text{in}_{\text{act}}(v) : \tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event Option} \)

\[
\frac{((\ldots, \text{out}, \tau_{\text{out}}), \text{getOutput}())}{((\ldots, \text{out}, \tau_{\text{out}}), \text{getOutput}())}
\]

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

Result is from 1, 9

Case \( \text{out} = \text{in}_{\text{some}}(\text{in}_{\text{res}}(\text{res}(v_1, v_2)) : \tau_{\text{out}} \text{ Event}) : \tau_{\text{out}} \text{ Event Option} \)

\[
\frac{((\ldots, \text{out}, \tau_{\text{out}}), \text{getOutput}())}{((\ldots, \text{out}, \tau_{\text{out}}), \text{getOutput}())}
\]

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

Result is from 1, 13
Case $\text{out} = \text{in}_\text{none}(\text{unit}) : \tau_\text{out} \text{ Event Option}$

$\begin{align*}
((..., \text{out}, \tau_\text{out}), \text{getOutput}()) & \longrightarrow \\
((..., \text{out}, \tau_\text{out}), \text{in}_\text{none}(\text{unit}) : \text{Event Option})
\end{align*}$

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

Result is from 1, 5

Case $(C, e_1) \longrightarrow (C', e'_1)$

$\begin{align*}
(C, \text{monitor}(\tau_1, e_1)) & \longrightarrow (C', \text{monitor}(\tau_1, e'_1))
\end{align*}$

1. $\Lambda \vdash C \text{ ok}$ assumption
2. $\Lambda, \bullet \vdash \text{monitor}(\tau_1, e_1) : \tau$ assumption
3. $(C, e_1) \longrightarrow (C', e'_1)$ assumption
4. $\Lambda, \bullet \vdash e_1 : \\
(\text{evt} : \tau_1 \text{ Event} \times \text{pols} : \text{PolList} \times \text{os} : \text{OS} \times \text{vc} : \text{VC})$
5. $\tau = \tau_1 \text{ Event}$ 2, Inversion Lemma
6. $\exists \Lambda' : (\Lambda' \vdash C' \text{ ok} \land$ 1, 4, IH
7. $\Lambda', \bullet \vdash e'_1 : (\text{evt} : \tau_1 \text{ Event} \times \text{pols} : \text{PolList} \times \text{os} : \text{OS} \times \text{vc} : \text{VC})$
8. $\Lambda' \subseteq \Lambda'$
9. $\Lambda', \bullet \vdash \text{monitor}(\tau_1, e'_1) : \tau$ 5, 6, Rule monitor

Result is from 6, 7

Case $((M, R, \text{inOb}, rt, \text{out}, \tau_\text{out}), \text{monitor}(\tau_1, v))$ begin

$\begin{align*}
((M, R, \text{true}, rt, \text{out}, \tau_\text{out}), \{e_{\text{monitor}(\tau_1, v)}\}_{\text{monitor}(v)})
\end{align*}$

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

Result is from 10, 12
1. \( \Lambda \vdash C \text{ok} \) assumption
2. \( \Lambda, \bullet \vdash \text{invoke}(e_1, e_2) : \tau \) assumption
3. \( (C, e_1) \rightarrow (C', e'_1) \) assumption
4. \( \Lambda, \bullet \vdash e_1 : \text{String} \) assumption
5. \( \Lambda, \bullet \vdash e_2 : \text{TypedVal} \) assumption
6. \( \tau = \text{TypedVal Option} \) assumption
7. \( \exists \Lambda' : (\Lambda' \vdash C' \text{ok} \land \Lambda, \bullet \vdash e'_1 : \text{String} \land \Lambda \subseteq \Lambda') \) 1, 3, 4, IH
8. \( \Lambda', \bullet \vdash e_2 : \text{TypedVal} \) 5, 7, A-weakening Lemma
9. \( \Lambda', \bullet \vdash \text{invoke}(e'_1, e_2) : \tau \) 6-8, Rule Inovke

Result is from 7, 9

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

Result is from 6, 8

1. \( \Lambda \vdash (M, (F, \ldots), \ldots) \text{ok} \) assumption
2. \( \Lambda, \bullet \vdash \text{invoke}(s_1, v_2) : \tau \) assumption
3. \( \tau = \text{TypedVal Option} \) 2, Inversion Lemma
4. \( \Lambda, \bullet \vdash \text{unit} : \text{Unit} \) Rule unitVal
5. \( \Lambda, \bullet \vdash (\text{in} \text{none}(\text{unit}) : \text{TypedVal Option}) : \tau \) 3, 4, Def of types, Rule variant

Result is from 1, 5
Case \((s_1, \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F \)  
\(v_2 = \text{makeTypedVal}(\tau_1, v'_2)\)  
\((M, (F, \ldots), \text{invoke}(s_1, v_2)) \rightarrow ((M, (F, \ldots), \ldots), \in_{\text{some}}(\text{makeTypedVal}(\tau_2, \text{call}(\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1, v'_2))) : \text{TypedVal Option})\)  
\((\text{invokeValueExistsOk})\)

1. \(\Lambda \vdash (M, (F, \ldots), \ldots) \text{ ok}\)
2. \(\Lambda, \bullet \vdash \text{invoke}(s_1, v_2) : \tau\)
3. \(v_2 = \text{makeTypedVal}(\tau_1, v'_2)\)
4. \(\Lambda, \bullet \vdash v'_2 : \tau_1\)
5. \(\tau = \text{TypedVal Option}\)
6. \(\Lambda, \bullet \vdash v_2 : \tau_1\)
7. \(\Lambda \vdash (F, \ldots) \text{ ok}\)
8. \(\Lambda \vdash F \text{ ok}\)
9. \((s_1, \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F\)
10. \(8\) is only derivable by Rule F-Ok

\(\Lambda, \bullet \vdash (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1) : \tau'_1 \rightarrow \tau'_2\)

11. \(11\) is only derivable by Rule fun
12. \(\tau'_1 = \tau_1\) and \(\tau'_2 = \tau_2\)
13. \(\tau = \text{TypedVal Option}\)
14. \(\Lambda, \bullet \vdash \text{call}(\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1, v'_2) : \tau_2\)
15. \(\text{Let } x = \text{call}(\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1, v'_2)\)
16. \(\Lambda, \bullet \vdash \text{makeTypedVal}(\tau_2, x) : \text{TypedVal}\)
17. \(\Lambda, \bullet \vdash (\text{in}_{\text{some}}(\text{makeTypedVal}(\tau_2, x)) : \text{TypedVal Option}) : \tau\)

Result is from 1, 17

Case \((s_1, \text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1) \in F\)
\(v_2 = \text{makeTypedVal}(\tau_3, v'_2)\)
\((\text{invokeValueExistsBad})\)
\((M, (F, \ldots), \text{invoke}(s_1, v_2)) \rightarrow ((M, (F, \ldots), \ldots), \text{in}_{\text{none}}(\text{unit}) : \text{TypedVal Option})\)

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

Result is from 1, 5
Case \((C, e_1) \rightarrow (C', e'_1)\)

\[\frac{(C, \text{call}(e_1, e_2)) \rightarrow (C', \text{call}(e'_1, e_2))}{(\text{callE1})}\]

1. \(\Lambda \vDash C\text{ ok}\)
2. \(\Lambda, \bullet \vdash \text{call}(e_1, e_2) : \tau\)
3. \((C, e_1) \rightarrow (C', e'_1)\)
4. \(\Lambda, \bullet \vdash e_1 : \tau_1 \rightarrow \tau_2\)
5. \(\Lambda, \bullet \vdash e_2 : \tau_1\)
6. \(\tau = \tau_2\)
7. \(\exists \Lambda' : (\Lambda' \vdash C'\text{ ok} \land \Lambda', \bullet \vdash e'_1 : \tau_1 \rightarrow \tau_2 \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash e_2 : \tau_1\)
9. \(\Lambda', \bullet \vdash \text{call}(e'_1, e_2) : \tau\)

Result is from 7, 9

Case \((C, e_2) \rightarrow (C', e'_2)\)

\[\frac{(C, \text{call}(f, e_2)) \rightarrow (C', \text{call}(f, e'_2))}{(\text{callE2})}\]

1. \(\Lambda \vDash C\text{ ok}\)
2. \(\Lambda, \bullet \vdash \text{call}(f, e_2) : \tau\)
3. \((C, e_2) \rightarrow (C', e'_2)\)
4. \(\Lambda, \bullet \vdash f : \tau_1 \rightarrow \tau_2\)
5. \(\Lambda, \bullet \vdash e_2 : \tau_1\)
6. \(\tau = \tau_2\)
7. \(\exists \Lambda' : (\Lambda' \vdash C'\text{ ok} \land \Lambda', \bullet \vdash e'_2 : \tau_1 \land \Lambda \subseteq \Lambda')\)
8. \(\Lambda', \bullet \vdash f : \tau_1 \rightarrow \tau_2\)
9. \(\Lambda', \bullet \vdash \text{call}(f, e'_2) : \tau\)

Result is from 7, 9

\[
f \notin \text{range}(F)
\forall \text{pol} \in \text{pols} \ (f \neq \text{pol.onTrigger} \land f \neq \text{pol.onObligation})
\]

\[f = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1)\]

\[\frac{((M, (F, \text{pols}, \ldots), \ldots), \text{call}(f, v)) \xrightarrow{\text{begin}_f(v)} ((M, (F, \text{pols}, \ldots), \ldots), [(v/x_2, f/x_1)e_1]_{f(v)})}{(\text{callNonMonitoredFunction})}\]

1. \(\Lambda \vDash (M, (F, \ldots), \ldots)\text{ ok}\)
2. \(\Lambda, \bullet \vdash \text{call}(f, v) : \tau\)
3. \(f = (\text{fun } x_1(x_2 : \tau_1) : \tau_2 = e_1)\)
4. \(\Lambda, \bullet \vdash f : \tau_1' \rightarrow \tau_2'\)
5. \(\Lambda, \bullet \vdash v : \tau_1'\)
6. \(\tau = \tau_2'\)
7. \(\Lambda, \{x_1 : \tau_1' \rightarrow \tau_2', x_2 : \tau_1'\} \vDash e_1 : \tau_2'\)
8. \(\Lambda, \bullet \vdash [v/x_2, f/x_1]e_1 : \tau\)
9. \(\Lambda, \bullet \vdash [(v/x_2, f/x_1)e_1]_{f(v)} : \tau\)

Result is from 1, 9

145
\[
\text{(name = s, onTrigger = } f_1, \text{ onObligation = } f_2, \text{ vote = } f_3) \in \text{pol}\]
\[
f_1 = (\text{fun } x_1(x_2 : \text{Event}) : \text{Unit} = e_1)
\]
\[
\text{Case } \frac{((M, (F, \text{pol}, os, vc), \text{inOb}, rt, out, \tau_{out}), \text{call}(f_1, v))}{\text{f_1(v)}} \begin{array}{c}
(M, (F, \text{pol}, os, vc), \text{inOb}[, ] : \text{Res}_{\text{List}}, out, \tau_{out}), ([f_1/x_1, v/x_2]e_1)_{f_1(v)}
\end{array}
\]
1. \( \lambda \vdash (M, (\text{F, pols, os, vc), inOb}, rt, out, \tau_{out}) \text{ ok} \)
2. \( \lambda \vdash \text{call}(f_1, v) : \tau \)
3. \( f_1 = (\text{fun } x_1(x_2 : \text{Event}) : \text{Unit} = e_1) \)
4. \( \lambda \vdash f_1 : \tau_1 \rightarrow \tau_2 \)
5. \( \lambda \vdash v : \tau_1 \)
6. \( \tau = \tau_2 \)
7. \( \lambda, \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e_1 : \tau_2 \)
8. \( \lambda \vdash [f_1/x_1, v/x_2]e_1 : \tau \)
9. \( M : \lambda \)
10. \( \lambda \vdash (\text{F, pols, os, vc}) \text{ ok} \)
11. \( \lambda \vdash \text{inOb} : \text{Bool} \)
12. \( \lambda \vdash \text{out} : \tau_{out} \text{ Event Option} \)
13. \( \lambda \vdash [[]] : \text{Res}_{\text{List}} \)
14. \( \lambda \vdash (M, (\text{F, pols, os, vc}, \text{inOb}, [[]] : \text{Res}_{\text{List}}, out, \tau_{out}) \text{ ok} \)
15. \( \lambda \vdash ([f_1/x_1, v/x_2]e_1)_{f_1(v)} : \tau \)
Result is from 14, 15

\[
\text{(callOnTrigger)}
\]
\[
\text{(name = s, onTrigger = } f_1, \text{ onObligation = } f_2, \text{ vote = } f_3) \in \text{pol}\]
\[
f_2 = (\text{fun } x_1(x_2 : \text{Res}_{\text{List}}) : \text{Unit} = e_1)
\]
\[
\text{Case } \frac{((M, (F, \text{pol}, os, vc), \text{inOb}, rt, out, \tau_{out}), \text{call}(f_2, v))}{\text{f_2(v)}} \begin{array}{c}
(M, (F, \text{pol}, os, vc), \text{inOb}[, ] : \text{Res}_{\text{List}}, out, \tau_{out}), ([f_2/x_1, v/x_2]e_1)_{f_2(v)}
\end{array}
\]
1. \( \lambda \vdash (M, (\text{F, pols, os, vc), inOb}, rt, out, \tau_{out}) \text{ ok} \)
2. \( \lambda \vdash \text{call}(f_2, v) : \tau \)
3. \( f_2 = (\text{fun } x_1(x_2 : \text{Event}) : \text{Unit} = e_1) \)
4. \( \lambda \vdash f_2 : \tau_1 \rightarrow \tau_2 \)
5. \( \lambda \vdash v : \tau_1 \)
6. \( \tau = \tau_2 \)
7. \( \lambda, \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e_1 : \tau_2 \)
8. \( \lambda \vdash [f_2/x_1, v/x_2]e_1 : \tau \)
9. \( M : \lambda \)
10. \( \lambda \vdash (\text{F, pols, os, vc}) \text{ ok} \)
11. \( \lambda \vdash \text{inOb} : \text{Bool} \)
12. \( \lambda \vdash \text{out} : \tau_{out} \text{ Event Option} \)
13. \( \lambda \vdash [[]] : \text{Res}_{\text{List}} \)
14. \( \lambda \vdash (M, (\text{F, pols, os, vc}, \text{inOb}, [[]] : \text{Res}_{\text{List}}, out, \tau_{out}) \text{ ok} \)
15. \( \lambda \vdash ([f_2/x_1, v/x_2]e_1)_{f_2(v)} : \tau \)
Result is from 14, 15

\[
\text{(callOnObligation)}
\]
(s, fun \( x_1(x_2 : \tau_1) : \tau_2 = e_1 \) ∈ \( F \) \( f = (\text{fun} \ x_1(x_2 : \tau_1) : \tau_2 = e_1) \)) (callFromObligation)

\[
\begin{align*}
(M, (F, \text{pols}, os, vc), \text{true}, rt, out, \tau_{out}), \text{call}(f, v)) & \\
\xrightarrow{\text{begin}_{x_1(v)}; \text{begin}_{\text{appendRes}}(\cdot)} ((M, (F, \text{pols}, os, vc), \text{true}, rt) \odot) & \\
\text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v)), \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1)) & \equiv \\
\[] : \text{Res}_{\text{List}}, out, \tau_{out}), \{[f/x_1, v/x_2]e_1\}_{x_1(v)} & \\
\end{align*}
\]

1. \( \Lambda \vdash (M, (F, \text{pols}, os, vc), \text{true}, rt, out, \tau_{out}) \text{ ok} \)  
2. \( \Lambda \vdash \text{call}(f, v) : \tau \)  
3. \( f = (\text{fun} \ x_1(x_2 : \tau_1) : \tau_2 = e_1) \)  
4. \( \Lambda \vdash f : \tau_1' \rightarrow \tau_2' \)  
5. \( \Lambda \vdash v : \tau_1' \)  
6. \( \tau = \text{tau}_2' \)  
7. \( 4 \) is only derivable by Rule fun  
8. \( \tau_1 = \tau_1' \land \tau_2 = \tau_2' \)  
9. \( \Lambda, \{x_1 : \tau_1 \rightarrow \tau_2, x_2 : \tau_1\} \vdash e_1 : \tau_2 \)  
10. \( \Lambda \vdash [f/x_1, v/x_2]e_1 : \tau_2 \)  
11. \( \Lambda \vdash \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1) : \text{TypedVal} \)  
12. \( \Lambda \vdash \text{makeTypedVal}(\tau_1, v) : \text{TypedVal} \)  
13. \( \Lambda \vdash s : \text{String} \)  
14. \( \Lambda \vdash \text{act}(s, \text{makeTypedVal}(\tau_1, v)) : \text{Act} \)  
15. \( \Lambda \vdash \text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v)), \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1)) : \text{Res} \)  
16. \( \text{let } r = \text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v)), \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1)) \)  
17. \( \Lambda \vdash ([] : \text{Res}_{\text{List}}) : \text{Res}_{\text{List}} \)  
18. \( \Lambda \vdash (r :: []) : \text{Res}_{\text{List}} \)  
19. \( \Lambda \vdash rt : \text{Res}_{\text{List}} \)  
20. \( \Lambda \vdash rt \odot (r :: []) : \text{Res}_{\text{List}} : \text{Res}_{\text{List}} \)  
21. \( M : \Lambda \)  
22. \( \Lambda \vdash (F, \text{pols}, os, vc) \text{ ok} \)  
23. \( \Lambda \vdash \text{true} : \text{Bool} \)  
24. \( \Lambda \vdash out : \tau_{out} \text{ Event Option} \)  
25. \( \Lambda \vdash (M, (F, \text{pols}, os, vc), \text{true}, rt \odot (r :: []) : \text{Res}_{\text{List}}, out, \tau_{out}) \text{ ok} \)  
26. \( \Lambda \vdash ([f/x_1, v/x_2]e_1)_{x_1(v)} : \tau \)  

Result is from 25, 26

\[
\begin{align*}
\text{res}(\text{act}(s, \text{makeTypedVal}(\tau_1, v)), \text{makeTypedVal}(\tau_2, [f/x_1, v/x_2]e_1)) & \\
\{[f/x_1, v/x_2]e_1\}_{x_1(v)} & \\
\end{align*}
\]
Case \((s,f) \in F \quad f = \text{(fun } x_1(x_2 : \tau_1) : \tau_2 = e_1)\) (callFromApplication)

\[ (M, (F, \text{pols, os, vc}), \text{false, rt, out, } \tau_{old}), \text{call}(f, v) \xrightarrow{\text{begin}_f(v)} \]

\((M, (F, \text{pols, os, vc}), \text{false, rt, } \text{in}_{\text{none}}(\text{unit}) : \tau_2 \text{ Event Option, } \tau_2, e_{\text{procEvt}})\)

where \(e_{\text{procEvt}} = \{
\text{let } aux = \text{(fun aux(event : } \tau_2 \text{ Event) : } \tau_2 \text{ Res = case event of act a } \Rightarrow \\
\text{case invoke(a.name, a.arg) of some r } \Rightarrow \\
\text{case tryCast(r2, r1) of some v1 } \Rightarrow \\
\text{let action\_out = inres(res(a, v1) : } \tau_2 \text{ Event in} \\
\text{let mon\_out = monitor(r2,}
\text{(evt = action\_out, pols = pols, os = os, vc = vc)) in} \\
\text{call(aux, mon\_out)}
\text{end end} \\
\text{| none u1 } \Rightarrow \text{call(aux, event)} \\
\text{| none u2 } \Rightarrow \text{call(aux, event)} \\
\text{| res } r_2 \Rightarrow r_2
\text{) in} \\
\text{call(aux, inact act(s, v) : } \tau_2 \text{ Event).result end}\}
\]

1. \(\Lambda \vdash (M, (F, \text{pols, os, vc}), \text{false, rt, out, } \tau_{old}) \text{ ok}\) assumption
2. \(\Lambda, \bullet \vdash \text{call}(f, v) : \tau\) assumption
3. \(f = \text{(fun } x_1(x_2 : \tau_1) : \tau_2 = e_1)\) assumption
4. \(M : \Lambda\) 1, C-Inversion Lemma
5. \(\Lambda \vdash (F, \text{pols, os, vc}) \text{ ok}\) 1, C-Inversion Lemma
6. \(\Lambda, \bullet \vdash \text{rt} : \text{ResList}\) 1, C-Inversion Lemma
7. \(\Lambda, \bullet \vdash \text{false : Bool}\) Rule boolVal
8. \(\Lambda, \bullet \vdash \text{unit : Unit}\) Rule unitVal
9. \(\Lambda, \bullet \vdash (\text{in}_{\text{none}}(\text{unit}) : \tau_2 \text{ Event Option}) : \tau_2 \text{ Event Option}\) 8, Rule variant, Def of types
10. \(\Lambda \vdash (M, (F, \text{pols, os, vc}), \text{false, rt, } \text{in}_{\text{none}}(\text{unit}) : \tau_2 \text{ Event Option, } \tau_2, e_{\text{procEvt}}) \text{ ok}\) 4-7, 9, Rule C-ok
11. \(\text{Let } \Gamma_0 = \{\text{aux : } \tau_2 \text{ Event } \rightarrow \tau_2 \text{ Res, event : } \tau_2 \text{ Event,}
\text{a : Act, r2 : } \tau_2 \text{ Res, r1 : TypedVal, u2 : Unit, v1 : } \tau_2, u1 : \text{Unit,}
\text{action\_out : } \tau_2 \text{ Event, mon\_out : } \tau_2 \text{ Event}\}\) assumption
12. \(\Lambda, \Gamma_0 \vdash \text{aux : } \tau_2 \text{ Event } \rightarrow \tau_2 \text{ Res}\) 11, Rule var
13. \(\Lambda, \Gamma_0 \vdash \text{mon\_out : } \tau_2 \text{ Event}\) 11, Rule var
14. \(\Lambda, \Gamma_0 \vdash \text{call(aux, mon\_out) : } \tau_2 \text{ Res}\) 12, 13, Rule call
15. \(\text{Let } \Gamma_1 = \)
\{ aux : \tau_2 \text{ Event} \to \tau_2 \text{ Res}, event : \tau_2 \text{ Event}, \\
a : \text{Act}, \tau_2 : \tau_2 \text{ Res}, r_1 : \text{TypedVal}, u_2 : \text{Unit}, v_1 : \tau_2, \\
u_1 : \text{Unit}, \text{action\_out} : \tau_2 \text{ Event}\} \\
16. \Lambda, \bullet \vdash \text{pols} : \text{Pol\_List} \\
17. \Lambda, \bullet \vdash \text{os} : \text{OS} \\
18. \Lambda, \bullet \vdash \text{vc} : \text{VC} \\
19. \Lambda, \Gamma_1 \vdash \text{pols} : \text{Pol\_List} \\
20. \Lambda, \Gamma_1 \vdash \text{os} : \text{OS} \\
21. \Lambda, \Gamma_1 \vdash \text{vc} : \text{VC} \\
22. \text{Let } c = ((\text{evt} = \text{action\_out}, \text{pols} = \text{pols}, \text{os} = \text{os}, \text{vc} = \text{vc}) \\
23. \Lambda, \Gamma_1 \vdash c : (\text{evt} : \tau_2 \text{ Event} \times \text{pol} : \text{Pol\_List} \times \text{os} : \text{OS} \times \text{vc} : \text{VC}) \\
24. \Lambda, \Gamma_1 \vdash \text{monitor}(\tau_2, c) : \tau_2 \text{ Event} \\
25. \Lambda, \Gamma_1 \vdash \text{let mon\_out} = \\
\text{monitor}(\tau_2, c) \text{ in case } (\text{aux, mon\_out}) \text{ end} : \tau_2 \text{ Res} \\
26. \text{Let } \Gamma_2 = \{ \text{aux} : \tau_2 \text{ Event} \to \tau_2 \text{ Res}, \text{event} : \tau_2 \text{ Event}, \\
a : \text{Act}, \tau_2 : \tau_2 \text{ Res}, r_1 : \text{TypedVal}, u_2 : \text{Unit}, v_1 : \tau_2, u_1 : \text{Unit}\} \\
27. \Lambda, \Gamma_2 \vdash a : \text{Act} \\
28. \Lambda, \Gamma_2 \vdash v_1 : \tau_2 \\
29. \Lambda, \Gamma_2 \vdash \text{res}(a, v_1) : \tau_2 \text{ Res} \\
30. \Lambda, \Gamma_2 \vdash (\text{in\_res res}(a, v_1) : \tau_2 \text{ Event}) : \tau_2 \text{ Event} \\
31. \text{Let } e_2 = (\text{let mon\_out} = \text{monitor}(\tau_2, c) \text{ in} \\
\text{call}(\text{aux, mon\_out}) \text{ end}) \\
32. \Lambda, \Gamma_2 \vdash \text{let action\_out} = \text{in\_res res}(a, v_1) : \\
\tau_2 \text{ Event in e_2 end} : \tau_2 \text{ Res} \\
33. \Lambda, \Gamma_2 \vdash \text{event} : \tau_2 \text{ Event} \\
34. \Lambda, \Gamma_2 \vdash \text{aux} : \tau_2 \text{ Event} \to \tau_2 \text{ Res} \\
35. \Lambda, \Gamma_2 \vdash \text{call}(\text{aux, event}) : \tau_2 \text{ Res} \\
36. \text{Let } \Gamma_3 = \\
\{ \text{aux} : \tau_2 \text{ Event} \to \tau_2 \text{ Res}, \text{event} : \tau_2 \text{ Event}, a : \text{Act}, \\
\tau_2 : \tau_2 \text{ Res}, r_1 : \text{TypedVal}, u_2 : \text{Unit}\} \\
37. \Lambda, \Gamma_3 \vdash r_1 : \text{TypedVal} \\
38. \Lambda, \Gamma_3 \vdash \text{tryCast}(\tau_2, r_1) : \text{TypedVal Option} \\
39. \text{Let } e_3 = (\text{let action\_out} = \text{in\_res res}(a, v_1 : \tau_2 \text{ Event in e_2 end}) \\
40. \Lambda, \Gamma_3 \vdash \text{case tryCast}(\tau_2, r_1) \text{ of some } r_1 \Rightarrow e_3 | \\
\text{none } u_1 \Rightarrow \text{call}(\text{aux, event}) : \tau_2 \text{ Res} \\
41. \text{Let } e_4 = (\text{case tryCast}(\tau_2, r_1) \text{ of some } r_1 \Rightarrow e_3 | \\
\text{none } u_1 \Rightarrow \text{call}(\text{aux, event})) \\
42. \Lambda, \Gamma_3 \vdash \text{aux} : \tau_2 \text{ Event} \to \tau_2 \text{ Res} \\
43. \Lambda, \Gamma_3 \vdash \text{event} : \tau_2 \text{ Event} \\
44. \Lambda, \Gamma_3 \vdash \text{call}(\text{aux, event}) : \tau_2 \text{ Res} \\
45. \text{Let } \Gamma_4 = \\
\{ aux : \tau_2 \text{ Event} \to \tau_2 \text{ Res, event : \tau_2 \text{ Event}, } \\
a : \text{Act}, \tau_2 : \tau_2 \text{ Res}, r_1 : \text{TypedVal}, u_2 : \text{Unit}, v_1 : \tau_2, \\
u_1 : \text{Unit, action\_out} : \tau_2 \text{ Event}\}
\{ \text{aux} : \tau_2 \text{Event} \to \tau_2 \text{Res}, \text{event} : \tau_2 \text{Event}, a : \text{Act}, r_2 : \tau_2 \text{Res} \} \\
46. \Lambda, \Gamma_4 \vdash a : \text{Act} \\
47. \Lambda, \Gamma_4 \vdash a.\text{name} : \text{String} \\
48. \Lambda, \Gamma_4 \vdash a.\text{arg} : \text{TypedVal} \\
49. \Lambda, \Gamma_4 \vdash \text{invoke}(a.\text{name}, a.\text{arg}) : \text{TypedVal Option} \\
50. \Lambda, \Gamma_4 \vdash (\text{case invoke}(a.\text{name}, a.\text{arg}) \text{ of some } r_1 \Rightarrow e_4 \mid \text{none } u_2 \Rightarrow \text{call(aux, event)}) : \tau_2 \text{Res} \\
51. \text{Let } e_5 = (\text{case invoke}(a.\text{name}, a.\text{arg}) \text{ of some } r_1 \Rightarrow e_4 \mid \text{none } u_2 \Rightarrow \text{call(aux, event)}) \\
52. \Lambda, \Gamma_4 \vdash r_2 : \tau_2 \text{Res} \\
53. \text{Let } \Gamma_5 = \{ \text{aux} : \tau_2 \text{Event} \to \tau_2 \text{Res}, \text{event} : \tau_2 \text{Event} \} \\
54. \Lambda, \Gamma_5 \vdash \text{event} : \tau_2 \text{Event} \\
55. \Lambda, \Gamma_5 \vdash (\text{case event of act } a \Rightarrow e_5 \mid \text{res } r_2 \Rightarrow r_2) : \tau_2 \text{Res} \\
56. \text{Let } e_6 = (\text{case event of act } a \Rightarrow e_5 \mid \text{res } r_2 \Rightarrow r_2) \\
57. \text{Let } \Gamma_6 = \{ \text{aux} : \tau_2 \text{Event} \to \tau_2 \text{Res} \} \\
58. \Lambda, \Gamma_6 \vdash \text{aux} : \tau_2 \text{Event} \to \tau_2 \text{Res} \\
59. \Lambda, \Gamma_6 \vdash s : \text{String} \\
60. \Lambda, \bullet \vdash v : \text{TypedVal} \\
61. \Lambda, \Gamma_6 \vdash \text{act}(s, v) : \text{Act} \\
62. \Lambda, \Gamma_6 \vdash (\text{inact } \text{act}(s, v) : \tau_2 \text{Event}) : \tau_2 \text{Event} \\
63. \Lambda, \Gamma_6 \vdash \text{call}(\text{aux, inact } \text{act}(s, v) : \tau_2 \text{Event}) : \tau_2 \text{Res} \\
64. \Lambda, \Gamma_6 \vdash \text{call}(\text{aux, inact } \text{act}(s, v) : \tau_2 \text{Event}).\text{result} : \tau_2 \\
65. \Lambda, \bullet \vdash (\text{fun aux}(\text{event} : \tau_2 \text{Event}) : \tau_2 \text{Res} = e_6 : \tau_2 \text{Event} \to \tau_2 \text{Res}) \\
66. \Lambda, \bullet \vdash (\text{let aux} = (\text{fun aux}(\text{event} : \tau_2 \text{Event}) : \tau_2 \text{Res} = e_6) \in \text{call}(\text{aux, inact } \text{act}(s, v) : \tau_2 \text{Event}).\text{result end}) : \tau_2 \\
67. \Lambda, \bullet \vdash (\text{let aux} = (\text{fun aux}(\text{event} : \tau_2 \text{Event}) : \tau_2 \text{Res} = e_6) \in \text{call}(\text{aux, inact } \text{act}(s, v) : \tau_2 \text{Event}).\text{result end})_{s(v)} : \tau_2 \\
68. \tau = \tau_2 \\
69. e_{\text{procEvt}} : \tau \\
70. \text{Result is from 10, 70}
Appendix B: Proof of Obligation Properties

This appendix proves five important properties of PoCo obligations. These proofs are based on twelve lemmas that can be found on pages 151 to 156. Page 156 presents proof of the Atomic-Obligation Theorem, page 158 presents the proof for the Conflict-Resolution Theorem, page 159 presents proof of the Obligation-Reaction Theorem, page 163 presents proof of the Pre-Obligation Completeness Theorem, and page 164 presents the proof of the Obligation-Permutability Theorem.

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

Proof.
1. $e_2$ cannot be reduced until $e_1 \rightarrow^* v_1$ Rule sequenceE2
2. after $e_1 \rightarrow^* v_1$ producing trace $t_1$, $t = t_1$ 1, sequenceE1
3. values are always added to end of trace Definition of label: $v \rightarrow \beta$
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 $e = (\text{case } e_x \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n)$: if $e_x \rightarrow^* \text{in}_i v_i$ 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. $e = (\text{case } e_x \text{ of } \ell_1 x_1 \Rightarrow e_1 | \cdots | \ell_n x_n \Rightarrow e_n)$ assumption
2. $e_x$ must be reduced before $e_1, \ldots, e_n$ are reduced Rule CaseE, Rule caseV
3. $e_x \rightarrow^* \text{in}_i v_i$ while producing traces $t_x$ assumption
4. after $e_x \rightarrow^* \text{in}_i v_i$ producing trace $t_x$, $t = t_x$ 2, 3, Rule CaseE
5. after $e_x \rightarrow^* \text{in}_i v_i$, $e \rightarrow^* [v_i/x_i] e_i$ 1, 2, Rule caseV
Lemma 14 (While-statement Traces).
Given a well-typed expression $e$ where $e = \text{while} \ (e_1) \{e_2\}$, if $e_1 \rightarrow^* v_1$ and $e_2 \rightarrow^* v_2$ while producing traces matching expression $t_1$ and $t_2$ respectively, then $e \rightarrow^* e'$ while producing trace $t$ that matches $\infty$-language expression $(t_1, t_2)\infty t_1$.

Proof.

1. $e = \text{while} \ (e_1) \{e_2\}$ assumption
2. $e \rightarrow \text{if} \ e_1 \text{ then } (e_2; \text{while} \ (e_1)\{e_2\}) \text{ else } \text{unit}$ 1, Rule whileE
3. $e_1 \rightarrow^* v_1$ producing trace $t_1$ assumption
4. assume $v_1 = \text{false}$ while producing trace $t_1$ in the first iteration assumption
5. $e \rightarrow^* \text{unit}$ while producing trace $t_1$ 2, 3, 4, Rule ifFalse
6. assume $v_1 = \text{true}$ while producing trace $t_1$ in the first iteration assumption
7. $e_2 \rightarrow^* (e_2; \text{while} \ (e_1)\{e_2\})$ 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 $\infty$-regular expression $(t_1, t_2)\infty t_1$ Rules of $\infty$-expression 10, 11
12. the trace produced after first iteration matches $\infty$-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 $\infty$-regular expression $(t_1, t_2)\infty t_1$ assumption
14. values are always added to end of trace Definition of $\text{label} : v \rightarrow \beta$
15. the trace $t_{n+1}$ produced after $(n + 1)^{th}$ iteration will either [10, 13, 14]
16. both $[t_n, t_1]$ and $[t_n, t_1, t_2]$ matches $\infty$-regular expression $(t_1, t_2)\infty t_1$ 11, 13, Rules of $\infty$-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 $e = (\text{let } x = e_1 \text{ in } e_2 \text{ end})$: 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 $\text{letE}$, Rule $\text{letValue}$
2. after $e_1 \rightarrow^* v_1$ producing trace $t_1$, $t = t_1$  
   1, Rule $\text{letE}$
3. values are always added to end of trace  
   Definition of $\frac{\text{label} : v}{\rightarrow} \beta$
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 $e = \text{if } e_3 \text{ then } e_1 \text{ else } e_2$: if $e_1 \rightarrow^* v_1$, $e_2 \rightarrow^* v_2$, 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 $v_3 = \text{true}$ or $e \rightarrow^* v_2$ while producing trace $[t_3, t_2]$ when $v_3 = \text{false}$.

Proof.
1. $e_3$ must be reduced before $e_1$ or $e_2$ are reduced  
   Rule $\text{ifE}$, Rule $\text{ifTrue}$, Rule $\text{ifFalse}$
2. after $e_3 \rightarrow^* v_3$ producing trace $t_3$, $t = t_3$  
   1, Rule $\text{ifE}$
3. if $v_3 = \text{false}$, if $v_3$ then $e_1$ else $e_2$ $\rightarrow^* e_2$  
   Rule $\text{ifFalse}$
4. values are always added to end of trace  
   Definition of $\frac{\text{label} : v}{\rightarrow} \beta$
5. if $v_3 = \text{false}$, after if $v_3$ then $e_1$ else $e_2$ $\rightarrow^* v_2$, $t = [t_3, t_2]$  
   2, 3, 4
6. if $v_3 = \text{true}$, if $v_3$ then $e_1$ else $e_2$ $\rightarrow^* e_1$  
   Rule $\text{ifTrue}$
7. if $v_3 = \text{true}$, 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 $e = \text{while}(\neg \text{empty}(!\ell))\{ \text{item ::= head}(\ell); \ell ::= \text{tail}(\ell); e_2; \}$ where $\ell : \tau_{\text{ListRef}}$: 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 $\ell$, then the trace $t$ produced when $e \rightarrow^* v$ matches regular expression $t_2^N$ where $N$ is the length of $\ell$.

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 $\ell$  
   assumption
3. the trace produced as the expression $\neg\text{empty}(!\ell)$ reduces to $v$ is $[]$

4. $e \rightarrow^* v$ while producing trace $t$ that matches $([], t_2)^\infty$

5. $t$ matches $t_2^\infty$

6. $|\text{tail}(!\ell)| = |\ell| - 1$

7. $\ell ::= \text{tail}(!\ell)$ reduce $!\ell$ by 1

8. $\text{head}(!\ell)$ does not modify $|!\ell|$

9. each iteration of while loop reduces $|!\ell|$ by 1

10. when $|!\ell| > 0$, $\neg\text{empty}(!\ell) \rightarrow^* \text{true}$

11. when $|!\ell| == 0$, $\neg\text{empty}(!\ell) \rightarrow^* \text{false}$

12. $|!\ell| > 0 \rightarrow^* \text{true}$ N times

13. $e \rightarrow^* v$ while producing trace $t$ that matches $t_2^N$

---

**Lemma 18 (CallOnTrigger-statement Traces).**

Given an expression $e_1 = \text{call(onTrigger, e)}$: if $\Lambda, \bullet \vdash e_1 \text{ ok}$, and $e_1 \rightarrow^* v_1$ while producing a trace $t$, then $t$ matches the $\infty$-expression $(\text{beginOb}(e) (\neg\text{ob}(e))^\infty \text{endOb}(e))$.

**Proof.**

1. $\text{beginOb}(e)$ can not occur directly from $e_{\text{onTrigger}}$

2. $\text{endOb}(e)$ can not occur directly from $e_{\text{onTrigger}}$

3. $\text{call(onTrigger, e)} \rightarrow^* \{v/x\}e_{\text{onTrigger}}_{\text{onTrigger(e)}}$

   while producing trace $\text{begin}_{\text{onTrigger(e)}}$

4. the trace $t_1$ produced while $[v/x]\text{onTrigger} \rightarrow^* v'$ matched $(\neg\text{ob}(e))^\infty$

5. values are always added to end of trace

6. after $\{v/x\}e_{\text{onTrigger}}_{\text{onTrigger(e)}} \rightarrow^* \{v'/x\}e_{\text{onTrigger}}_{\text{onTrigger(e)}}$,

   $t = \text{begin}_{\text{onTrigger(e)}, t_1}$

7. $\{v'/x\}_{\text{onTrigger(e)}} \rightarrow^* v_1$ while producing trace $\text{end}_{\text{onTrigger(e)}}$

8. after $\{v'/x\}_{\text{onTrigger(e)}} \rightarrow^* v_1$, the produced trace

   $t = \text{begin}_{\text{onTrigger(e)}, t_1, \text{end}_{\text{onTrigger(e)}}}$

9. the trace $t$ produced while $e_1 \rightarrow^* v_1$ matches the $\infty$-expression

   $(\text{beginOb}(e) (\neg\text{ob}(e))^\infty \text{endOb}(e))$

---

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

**Proof.**

1. $\text{beginOb}(e)$ cannot occur directly from $e_{\text{onObligation}}$ 
   Definition of $\cdots \vdash e_1 \text{ ok}$
2. $\text{endOb}(e)$ cannot occur directly from $e_{\text{onObligation}}$ 
   Definition of $\cdots \vdash e_1 \text{ ok}$
3. $\text{call}(\text{onObligation}, e) \rightarrow^* \{[v/x]e_{\text{onObligation}}\}_{\text{onObligation}(e)}$ 
   3, Rule callNonMonitoredFunction
   while producing trace $\text{begin}_{\text{onObligation}}(e)$

4. values are always added to end of trace 
   Definition of $\xrightarrow{\text{label}: v} \beta$
5. after $\{[v/x]e_{\text{onObligation}}\}_{\text{onObligation}(e)} \rightarrow^* \{v'\}_{\text{onObligation}(e)}$ 
   3, 4, 5, rule endLabelE
6. $\{v'\}_{\text{onObligation}(e)} \rightarrow^* v_1$ while producing trace 
   7, Rule endLabelValue
   $\text{end}_{\text{onObligation}}(e)$

7. After $\{v'\}_{\text{onObligation}(rt)} \rightarrow^* v_1$, the produced trace 
   5, 6, 7
   $t = \text{begin}_{\text{onObligation}(e)}, t_1, \text{end}_{\text{onObligation}(e)}$

8. the trace $t$ produced while $e_1 \rightarrow^* v_1$ matches the 
   4, 8, Definition of 
   $\infty$-expression (beginOb(e) (sim ob(e))^{\infty} endOb(e)) 
   beginOb(e) and endOb(e), 
   Rules of $\infty$-expression

\[ \square \]

**Lemma 20 (No-monitored-function Traces).**

Given a program $p$ which is an untrusted application $e_{\text{app}}$ with enforced policies, if $\Lambda, \bullet \vdash p \text{ ok}$, and $e_{\text{app}}$ does not contain any monitored functions and $p \rightarrow^* p'$ while producing a trace $t$, then $t$ matches the $\infty$-expression $(\sim \text{beginOb}(e))^{\infty}$.

**Proof.**

1. $\text{beginOb}(e)$ cannot occur directly from $e_{\text{app}}$ 
   Definition of $\cdots \vdash p \text{ ok}$
2. $\text{call}(\text{monitor}, v)$ cannot occur directly from $e_{\text{app}}$ 
   Definition of $\cdots \vdash p \text{ ok}$
3. $\text{call}(\text{monitor}, v)$ happens only when a monitored function is 
   called 
   Rule callFromApplication
4. $\text{beginOb}(e)$ cannot exist in $t$ 
   1, 2, 3
5. $t$ matched $(\sim \text{beginOb}(e))^{\infty}$ 
   4

\[ \square \]

**Lemma 21 (App-no-append-result Traces).**

Given a program $p$ which is an untrusted application $e_{\text{app}}$ with enforced policies, if $\Lambda, \bullet \vdash p \text{ ok}$, and $e_{\text{app}}$ does not contain any monitored functions, and $p \rightarrow^* p'$ while producing a trace $t$, then $t$ matches the $\infty$-expression $(\sim \text{beginappendRes}(r))^{\infty}$.

**Proof.**
1. Rule callFromObligation is the only rule that adds 
\texttt{begin}appendRes(r)  
Inspection of the dynamic semantics
2. no function calls from \texttt{e}app can trigger rule callFromObligation  
Rule callFromApplication
3. \texttt{begin}appendRes(r) cannot exist in \texttt{t}  
1, 2
4. \texttt{t} matched \((\neg\texttt{begin}appendRes(r))\infty\)  
3

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

Proof.
1. Rule callFromObligation is the only rule that adds \(\texttt{begin}appendRes(r)\)  
Inspection of the dynamic semantics
2. no function calls from \(\texttt{e}app\) can trigger rule callFromObligation  
Rule callFromApplication
3. no non-function calls outside of \(\texttt{e}app\) can trigger rule callFromObligation  
Rule callFromApplication
4. \texttt{begin}appendRes(r) cannot exist in \(\texttt{t}\)  
1, 2, 3
5. \(\texttt{t}\) matched \((\neg\texttt{begin}appendRes(r))\infty\)  
4

Lemma 23 (No-makeCFG Traces).
Given a program \(p\) which is an untrusted application \(\texttt{e}app\) with enforced policies, if \(\Lambda, \bullet \vdash p \texttt{ ok}\), and \(\texttt{e}app\) does not contain any monitored functions, and \(p \rightarrow^* p'\) while producing a trace \(t\), then \(t\) matches the \(\infty\)-expression \((\neg\texttt{makeCFG}(v_1,v_2))\infty\).

Proof.
1. \texttt{end}makeCFG(v_1,v_2):g cannot occur directly from \(\texttt{e}app\)  
Definition of \(\cdots \vdash p \texttt{ ok}\)
2. \texttt{call}(\texttt{monitor}, v) cannot occur directly from \(\texttt{e}app\)  
Definition of \(\cdots \vdash p \texttt{ ok}\)
3. \texttt{call}(\texttt{monitor}, v) happens only when a monitored function is called  
Rule callFromApplication
4. \texttt{end}makeCFG(v_1,v_2):g cannot exist in \(\texttt{t}\)  
1, 2, 3
5. \(\texttt{t}\) matches \((\neg\texttt{makeCFG}(v_1,v_2))\infty\)  
4

Theorem 3 (Atomic Obligation).
For all \(p, t, \text{ and } t'\), if \(\Lambda, \bullet \vdash p \texttt{ ok}\), and \(p \rightarrow^* p'\), and \(t\) matches the \(\infty\)-expression \((((\infty) \begin{aligned} &\texttt{beginOb}(e_n) \ t' \texttt{beginOb}(e_m) \ ((\infty)) \end{aligned})\) then \(t'\) matches the \(\infty\)-expression \((((\infty) \texttt{endOb}(e_n) \ ((\infty))\)
Proof.

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

1. $t$ matches $\neg \text{beginOb}(e) \infty$  
   \[\text{Lemma 9}\]
2. The theorem holds vacuously in this case because $t$ does not match $((\infty) \text{beginOb}(e_n) \ t' \text{beginOb}(e_m) (\infty))$  
   \[1, \text{Rules of } \infty\text{-expression}\]

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

1. $\text{beginOb}(e)$ cannot occur directly from $e_{app}$  
   \[\text{Definition of } \cdots \vdash p \text{ ok}\]
2. $\text{call}(\text{monitor}, v)$ cannot occur directly from $e_{app}$  
   \[\text{Definition of } \cdots \vdash p \text{ ok}\]
3. $\text{onTrigger} \notin F$ and $\text{onObligation} \notin F$  
   \[3, \text{Definition of monitor, Rule } \text{callNonMonitoredFunction}\]
4. only lines 25 and 26 of monitor can result in a trace including $\text{beginOb}(e)$  
   \[\text{Lemma 2, Definition of monitor}\]
5. the trace produced while $\text{call}(o1, \text{onTrigg}, o1, \text{evt})$ on line 25 reducing to $v_{ot}$ is $[\text{beginOb}(e), (\neg \text{ob}(e)) \infty, \text{endOb}(e)]$  
   \[\text{Lemma 7, Definition of monitor}\]
6. the trace produced while $\text{call}(o2, \text{onOblig}, o2, \text{rt})$ on line 26 reducing to $v_{oo}$ is $[\text{beginOb}(e), (\neg \text{ob}(e)) \infty, \text{endOb}(e)]$  
   \[\text{Lemma 8, Definition of monitor}\]
7. containing of the two call expressions, the case expression spans from line 25 to 26 does not introduce additional $\text{beginOb}(e)$ in the overall trace  
   \[\text{Lemma 2, Definition of monitor}\]
8. the trace $t_1$ produced while case expression that spans from line 25 to 26 reducing to $v_1$ is $[\text{beginOb}(e), (\neg \text{ob}(e)) \infty, \text{endOb}(e)]$  
   \[5, 6, 7, \text{Lemma 2, Rules of } \infty\text{-expression}\]
9. taken the case expression that spans from line 25 to 26 as its else branch, the if-then-else expression spans from line 24 to 27 does not introduce additional $\text{beginOb}(e)$ in the overall trace  
   \[\text{Lemma 5, Definition of monitor}\]
10. the trace $t_2$ produced while the if-then-else expression spans from line 24 to 27 reducing to $v_2$ is either $[(-\text{beginOb}(e)) \infty]$ or $[(-\text{beginOb}(e)) \infty, t_1]$  
    \[8, 9, \text{Lemma 5, Rules of } \infty\text{-expression}\]
11. containing the if-else-then expression that spans from line 24 to 27, the let expression that spans from line 16 to 38 does not introduce additional $\text{beginOb}(e)$ in the overall trace  
    \[\text{Lemma 4, Definition of monitor}\]
12. the trace produced while the let expression spans from line 16 to 38 reducing to $v_3$ matches either $[(-\text{beginOb}(e)) \infty, t_1, (-\text{beginOb}(e)) \infty]$ or $[(-\text{beginOb}(e)) \infty]$  
    \[10, 11, \text{Lemma 4, Rules of } \infty\text{-language expression}\]
13. containing the let expression that spans from line 16 to 38, the while expression that spans from line 5 to 38 does not introduce additional $\text{beginOb}(e)$ in the overall trace  
    \[\text{Lemma 6, Definition of monitor}\]
14. the trace produced while the while expression spans from line 5 to 38 reducing to $v_4$ is either $[(-\text{beginOb}(e)) \infty]$ or $[((-\text{beginOb}(e)) \infty, t_1, (-\text{beginOb}(e)) \infty) \infty, (-\text{beginOb}(e)) \infty]$  
    \[12, 13, \text{Lemma 3, Rules of } \infty\text{-expression}\]
15. containing the while expression that spans from line 5 to 38, the monitor function doesn’t introduce additional beginOb(e) in the overall trace

16. the trace t produced when the overall monitor function reducing to v is either

\[
(((\neg \text{beginOb}(e))^\infty, t_1, (\neg \text{beginOb}(e))^\infty, (\neg \text{beginOb}(e))^\infty]\]
or \[
[(-\text{beginOb}(e))^\infty]\]

17. When t is \[(-\text{beginOb}(e))^\infty\], the theorem holds vacuously because t does not match \((.\infty)\ \text{beginOb}(en)\) \(t'\) \text{beginOb}(em) (.\infty))

18. When t is \[((\neg \text{beginOb}(e))^\infty, t_1, (\neg \text{beginOb}(e))^\infty, (\neg \text{beginOb}(e))^\infty]\) and \((\text{beginOb}(e) \neg \text{ob}(e)^\infty \text{endOb}(e))\) exists more than once in t, \(t'\) must always include endOb(e),

\[
\text{Results from 17 and 18} \]

\[\square\]

**Theorem 4** (Conflict Resolution).

For all programs p such that \(\Lambda \vdash p \text{ ok}\) and \(p \xrightarrow{\cdot} \ast p'\), t matches the \(\infty\)-expression \((-\text{beginOb}(e))^{\infty}\) \(v_{\text{true}}(e_n)\) \text{beginOb}(e_n) (\neg \text{beginOb}(e))^{\infty}\) where:

\[
v_{\text{true}}(e) := (\text{begin\_vote}(e) (\neg \text{beginOb}(e))^{\infty} \text{end\_vote}(e : v_n))^{N} \text{begin\_vc}(v_1 : \ldots : v_N) (\neg \text{beginOb}(e))^{\infty} \text{end\_vc}(v_1 : \ldots : v_N) : \text{true}.
\]

**Proof.**

Case 1: \(e_{\text{app}}\) does not contain any monitored function call:

1. \(t\) matches \((- \text{beginOb}(e))^{\infty}\)
2. \(t\) matches \(((\neg \text{beginOb}(e))^{\infty} v_{\text{true}}(e_N) \text{beginOb}(e_N)^{\infty} (\neg \text{beginOb}(e))^{\infty}\)

**Lemma 9**

1, Rules of \(\infty\)-expression

Case 2: \(e_{\text{app}}\) contains at least one monitored function call:

1. \(\text{beginOb}(e)\) cannot occur directly from \(e_{\text{app}}\)
2. \(\text{call}(\text{monitor}, v)\) cannot occur directly from \(e_{\text{app}}\)
3. \(\text{onTrigger} \notin F\) and \(\text{onObligation} \notin F\)
4. only lines 25 and 26 of monitor can result in a trace including \(\text{beginOb}(e)\)
5. the trace \(t_{\text{tot}}\) produced while \(\text{call}(o_1, \text{onTrig}, o_1.\text{evt})\) on line 25 reducing to \(v_{\text{tot}}\) is \([\text{beginOb}(e), (\neg \text{ob}(e))^{\infty}, \text{endOb}(e)]\)
6. the trace \(t_{\text{oo}}\) produced while \(\text{call}(o_2, \text{onOblig}, o_2.\text{rt})\) on line 26 reducing to \(v_{\text{oo}}\) matches \([\text{beginOb}(e), (\neg \text{ob}(e))^{\infty}, \text{endOb}(e)]\)

**Lemma 7**, Definition of monitor

3, Definition of monitor, Rule

**Lemma 8**, Definition of monitor

Definition of \(\cdots \vdash p \text{ ok}\)

Definition of \(\cdots \vdash p \text{ ok}\)

Definition of function scope for F

3, Definition of monitor, Rule
callNonMonitoredFunction

158
For all programs $p$

8. the trace $t_1$ produced while the case expression reducing to $v_1$ is $[\text{beginOb}(e), (-\text{ob}(e))^\infty, \text{endOb}(e)]$

9. taken the case expression that spans from line 25 to 26 as its then branch, the if-then-else expression spans from line 24 to 27 does not introduce additional $\text{beginOb}(e)$ in the overall trace

10. the trace $t_2$ produced while the condition expression $\text{call}((c\\text{.vc}, !\text{votes})$ on line 24 reducing to $v_2$ is

$$[\text{begin}_{\text{vc}}(v_1 :: ... :: v_N), \text{end}_{\text{vc}}(v_1 :: ... :: v_N)]$$

11. the trace $t_3$ produced while the if-then-else expression spanned from line 24 to 27 reducing to $v_3$ is either

$$[\text{begin}_{\text{vc}}(v_1 :: ... :: v_n), \text{end}_{\text{vc}}(v_1 :: ... :: v_n) : \text{false}]$$ or

$$[\text{begin}_{\text{vc}}(v_1 :: ... :: v_n), \text{end}_{\text{vc}}(v_1 :: ... :: v_n) : \text{true}, t_1]$$

12. the trace $t_4$ produced while the expression $\text{call}((\text{pol} \cdot \text{policy} \cdot \text{vote}, ob)$ on line 22 reducing to $v_4$ is

$$[\text{begin}_{\text{vote}}(e), \text{end}_{\text{vote}}(e : v_N)]$$

13. the trace $t_5$ produced while the while expression spans from line 19 to 23 reducing to $v_5$ is $[\text{begin}_{\text{vote}}(e), \text{end}_{\text{vote}}(e : v_N)]^N$

14. the trace $t_6$ produced while the expressions spans from line 19 to 27 reducing to $v_6$ is

$$[t_5, \text{begin}_{\text{vc}}(v_1 :: ... :: v_n), \text{end}_{\text{vc}}(v_1 :: ... :: v_n) : \text{false}]$$ or

$$[t_5, \text{begin}_{\text{vc}}(v_1 :: ... :: v_n), \text{end}_{\text{vc}}(v_1 :: ... :: v_n) : \text{true}, t_1]$$

15. the trace $t_7$ produced while the while expression spans from line 5 to 38 reducing to $v_7$ is $[t_6^\infty]$

16. the trace $t$ produced when the overall monitor function reducing to $v$ $[(\neg \text{beginOb}(e))^\infty, t_7, (\neg \text{beginOb}(e))^\infty]$

17. $t$ can be simplified to

$$[(\neg \text{beginOb}(e))^\infty, ((t_5, \text{begin}_{\text{vc}}(v_1 :: ... :: v_n)) \text{end}_{\text{vc}}(v_1 :: ... :: v_n) : \text{true}, \text{beginOb}(e_n)))]^\infty]$$

18. all possible traces produced by $p$ match the $\infty$-expression $[\neg \text{beginOb}(e)]^\infty, v_{\text{true}}(e_n), \text{beginOb}(e_n)]^\infty (\neg \text{beginOb}(e))^\infty)$ Definition of $v_{\text{true}}(e_n)$

**Theorem 5** (Obligation Reaction Part 1).

For all programs $p$ such that $\Lambda, \bullet \vdash p \text{ ok}$ and $p \xrightarrow{t} p'$, $t$ matches the $\infty$-expression

$$((-\text{begin}_{\text{appendRes}})^\infty \text{begin}_{\text{appendRes}} \cdot \infty \text{endOb}(e) \cdot \infty \end{\text{makeCFG}(\text{onObligation}, v) : g)^N)^\infty)$$

**Proof.**

Case 1: $e_{\text{app}}$ does not contain any monitored functions calls:
1. \( t \) matches \((\neg \text{begin} \text{appendRes}())^{\infty} \)

2. \( t \) matches \((\neg \text{begin} \text{appendRes}())^{\infty} (\text{begin} \text{appendRes}())^{\infty} \text{endOb}(e) \)

\((\infty \text{end} \text{makeCFG}(\text{onObligation}, v : g)^{N})^{?})^{\infty} \)

**Lemma 10**

1, Rules of \( \infty \)-expression

Case 2: no obligations contain any monitored functions calls:

1. \( t \) matches \((\neg \text{begin} \text{appendRes}())^{\infty} \)

2. \( t \) matches \((\neg \text{begin} \text{appendRes}())^{\infty} (\text{begin} \text{appendRes}())^{\infty} \text{endOb}(e) \)

\((\infty \text{end} \text{makeCFG}(\text{onObligation}, v : g)^{N})^{?})^{\infty} \)

**Lemma 11**

1, Rules of \( \infty \)-expression

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

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

2. only lines 25 and 26 of monitor can result in a trace including \( \text{begin} \text{appendRes} \)

3. the trace produced while \( \text{call}(o_1.\text{onTrig}, o_1.\text{evt}) \) on line 25 reducing to \( v_{\text{ot}} \) is

\( ((\neg \text{begin} \text{appendRes}())^{\infty}, \text{begin} \text{appendRes}(), \infty, \text{endOb}(e)) \)

4. the trace produced while \( \text{call}(o_2.\text{onOblig}, o_2.\text{rt}) \) on line 26 reducing to \( v_{\text{ot}} \) is

\( ((\neg \text{begin} \text{appendRes}())^{\infty}, \text{begin} \text{appendRes}(), \infty, \text{endOb}(e)) \)

5. containing of the two call expressions, the trace produced while the case expression spans from line 25 to 26 reducing to \( v_2 \) is

\( ((\neg \text{begin} \text{appendRes}())^{\infty}, \text{begin} \text{appendRes}(), \infty, \text{endOb}(e)) \)

6. the trace produced while the \( \text{then} \) branch spans from line 29 to 36 reducing to \( v_3 \) is \([\infty, \text{end} \text{makeCFG}(\text{onObligation}, v : g)^{N}]^{N}\)

7. the trace produced while the \( \text{if-then-else} \) expression spans from line 28 to 37 evaluates to \( v_{\text{ot}} \) is

\( ([\infty, \text{end} \text{makeCFG}(\text{onObligation}, v : g)^{N}]^{N})^{?} \)

8. the trace \( t_6 \) produced while the expression spans from line 24 to 37 reducing to \( v_6 \) is

\( ((\neg \text{begin} \text{appendRes}())^{\infty}, \text{begin} \text{appendRes}(), \infty, \text{endOb}(e), (\infty \text{end} \text{makeCFG}(\text{onObligation}, v : g)^{N})^{?})^{N} \)

9. the trace \( t_7 \) produced while the \( \text{while} \) expression spans from line 5 to 38 reducing to \( v_7 \) is \([t_6]^{\infty}\)

10. the trace \( t_8 \) produced when the overall monitor function is reduced to \( v_8 \) is

\( ((\neg \text{begin} \text{appendRes}())^{\infty}, t_7, (\neg \text{begin} \text{appendRes}())^{\infty}) \)

11. \( p \) produces the trace

\( t = ((\neg \text{begin} \text{appendRes}())^{\infty}, t_8^{\infty}, (\neg \text{begin} \text{appendRes}())^{\infty}) \)

Lemma 8, Definition of monitor

6, Definition of monitor, Lemma 5

7, Lemma 1, Lemma 2, Lemma 4, Definition of monitor

8, Lemma 3, Lemma 4

9, Lemma 1, Definition of monitor

2, 9, Lemma 1, Definition of monitor

2, 10
12. \( t \) can be simplified to \( \left( -\text{beginAppendRes}() \right) \infty \), 
\( \left( -\text{beginAppendRes}() \right) \infty \left( \text{beginAppendRes}() \right) \infty \endOb(e) \)
\( \left( \infty \text{end} \text{makeCFG}(\text{onObligation}, v_1) : g \right)^N \)?

13. \( t \) matches \( \infty \)-expression \( \left( -\text{beginAppendRes}() \right) \infty \)
\( \left( \text{beginAppendRes}() \right) \infty \endOb(e) \left( \infty \text{end} \text{makeCFG}(\text{onObligation}, v_1) : g \right)^N \)?

\[\square\]

**Theorem 6** (Obligation Reaction Part 2).

For all programs \( p \) s.t. \( \Lambda, \bullet \vdash p \ \text{ok} \), and \( p \xrightarrow{\ast} p' \), and \( p \)'s monitor is \((M, \text{fun}_{mon}, \)
\( p_1 :: \cdots :: p_n, e_{os}, e_{vc})\) where the functions \( e_{vc}, p_1.onTrigger, \ldots, p_n.onTrigger, p_1.onObligation, \)
\( \ldots, p_n.onObligation \) terminate, for each event \( \text{end} \text{makeCFG}(v_1, v_2): g \) in \( t \) there must exist a \( v_{\text{true}}(g) \)
or \( v_{\text{false}}(g) \) in \( t \) where:

\[ v_{\text{true}}(e) ::= \left( \begin{array}{c}
\text{begin} \text{vote}(e) \left( -\text{beginOb}(e) \right) \infty \\
\text{end} \text{vote}(e) : v_n \end{array} \right)^N \begin{array}{c}
\text{end} \text{vc}(v_1 :: \cdots :: v_N) : \text{true} \\
\text{end} \text{vc}(v_1 :: \cdots :: v_N) : \text{false}
\end{array} \]

\[ v_{\text{false}}(e) ::= \left( \begin{array}{c}
\text{begin} \text{vote}(e) \left( -\text{beginOb}(e) \right) \infty \\
\text{end} \text{vote}(e) : v_n \end{array} \right)^N \begin{array}{c}
\text{begin} \text{vc}(v_1 :: \cdots :: v_N) : \text{true} \\
\text{begin} \text{vc}(v_1 :: \cdots :: v_N) : \text{false}
\end{array} \]

**Proof.**

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

1. \( t \) matches \( \left( -\text{end} \text{makeCFG}(v_1, v_2): g \right) \infty \)
2. The theorem holds vacuously in this case because \( t \) does not match either \( v_{\text{true}}(e) \) or \( v_{\text{false}}(e) \)

Lemma 10

1, Rules of \( \infty \)-expression

Case 2: \( e_{\text{app}} \) contains at least one monitored functions call:

1. \( \text{end} \text{makeCFG}(v_1, v_2): g \) cannot occur directly from \( e_{\text{app}} \)
2. \( \text{call}(\text{monitor}, v) \) cannot occur directly from \( e_{\text{app}} \)
3. only the two expressions \( \text{makeCFG} \) on lines 7 and 34 of monitor code can result in a trace including \( \text{end} \text{makeCFG}(v_1, v_2): g \)
4. the result of the expression \( \text{makeCFG} \) on line 7 is immediately appended to a list stored in \( \ell_{\text{obQueue}} \)

Definition of \( \cdots \vdash p \ \text{ok} \)

Definition of \( \cdots \vdash p \ \text{ok} \)

1, Definition of monitor

Rules listAppendE1, listAppendE2, Rule
listAppendValue, assignValue, Rule
derefValue, Definition of monitor

4, Rules listPrependE1, listPrependE2, Rule
listPrependValue, assignValue, Rule
derefValue, Definition of monitor

5. the list stored in \( \ell_{\text{obQueue}} \) is prepended to a list stored in \( \ell_{\text{obStack}} \) on the line 11 of the monitor code resulting in all results of \( \text{makeCFG} \) in line 7 being included in \( \ell_{\text{obStack}} \)

161
6. the result of the expression makeCFG on line 34 immediately appended to a list stored in \( \ell_{obQueue} \)

7. the list stored in \( \ell_{obQueue} \) is prepended to a list stored in \( \ell_{obStack} \) on line 36 of the monitor code resulting in all results of makeCFG in line 34 being included in \( \ell_{obStack} \)

8. results from all possible calls to makeCFG are stored in \( \ell_{obStack} \)

9. the trace produced while code from lines 19 to 27 is reduced to a value is either
   
   \[
   [(\text{begin}_{\text{vote}}(e), \text{end}_{\text{vote}}(e):v_N)^N, \begin{align*}
   &\text{begin}_{\text{vc}}(v_1::\ldots::v_N), \text{end}_{\text{vc}}(v_1::\ldots::v_N):\text{false}] \text{ or } \end{align*}
   \]
   
   \[
   [(\text{begin}_{\text{vote}}(e), \text{end}_{\text{vote}}(e):v_N)^N, \begin{align*}
   &\text{begin}_{\text{vc}}(v_1::\ldots::v_N), \text{end}_{\text{vc}}(v_1::\ldots::v_N):\text{true}, \text{beginOb}(e), (\neg\text{ob}(e))^{\infty}, \text{endOb}(e)]
   \]

10. the expression \( \ell_{obQueue} := \text{head}(\ell_{obStack}) \) on line 13 of the monitor code takes \( \ell_{obStack} \)'s first list of obligations and stores it in \( \ell_{obQueue} \)

11. the expression \( \text{let } ob = \text{head}(\ell_{obQueue}) \) on line 14 of the monitor code takes the first obligation stored in \( \ell_{obQueue} \)

12. if \( \ell_{obQueue} = [ob] \), then the expression \( \neg\text{empty}(\text{tail}(\ell_{obQueue})) \) on line 17 of of the monitor code will evaluate to \( false \)

13. If \( \neg\text{empty}(\text{tail}(\ell_{obQueue})) \) evaluates to \( false \), then the first element of \( \ell_{obQueue} \) will be removed after executing the expression \( \ell_{obStack} := \text{tail}(\ell_{obStack}) \) on line 18 of the monitor code

14. if \( \ell_{obQueue} = [ob,\ldots] \), then \( \neg\text{empty}(\text{tail}(\ell_{obQueue})) \) on line 17 of of the monitor code will evaluate to \( true \)

15. If \( \neg\text{empty}(\text{tail}(\ell_{obQueue})) \) evaluates to \( true \), the first element of \( \ell_{obQueue} \) will be replaced with \( \text{tail}(\ell_{obQueue}) \) after executing the expression on line 17 of the monitor code

Theorem 4(14)
Theorem 7 (Pre-obligation Completeness).

There exists well-typed programs $p_1$, $p_2$, and $p_3$ where $p_1 \to^* p_1'$, $p_2 \to^* p_2'$, and $p_3 \to^* p_3'$ while producing traces $t_1$, $t_2$, and $t_3$ respectively such that $t_1$, $t_2$, and $t_3$ match the $\infty$-expressions $e_{\text{pre}}$ for pre-obligations, $e_{\text{post}}$ for post-obligations and $e_{\text{ongoing}}$ for ongoing obligations where:

$$
e_{\text{pre}} = (\infty) \text{begin}_{f(x)}(\infty) \text{begin}_{\text{monitor}(\text{act}(f,x))}(\infty) \text{end}_{f(x);v}(\infty),$$

$$
e_{\text{post}} = (\infty) \text{end}_{f(x);v}(\infty) \text{begin}_{\text{monitor}(\text{res}(\text{act}(f,x),rt))}(\infty),$$

$$
e_{\text{ongoing}} = (e_{\text{pre}} | e_{\text{post}})(\infty) \left(e_{\text{pre}} | e_{\text{post}}\right).$$

Proof. Case 1: pre-obligation:

1. assume $p_1$ has a policy $\text{Pol}_1$ with the onTrigger
   
   \[
e_{\text{onTrigger}}(e) = \text{case e of act a ⇒ if a.name == "print" then call(log,e) else unit res r ⇒ unit}
   \]
   
   Assumption

2. $\text{print} \in F$
   
   1, Definition of $F$

3. when $e_{\text{app}}$ attempts to execute $\text{call}(|\text{print}, v)$ expression, $\text{begin}_{\text{print}(v)}$ gets added to the trace resulting in a trace of $[(\infty), \text{begin}_{\text{print}(v)}]$  
   
   2, Rule \text{callFromApplication}

4. when $e_{\text{app}}$ attempts to execute the expression $\text{call}(|\text{print}, v)$,  
   
   $\text{call}(\text{monitor}, \text{act}(|\text{print"}, v))$ will be triggered  
   
   2, Rule \text{callFromApplication}

5. after $\text{call}(\text{monitor}, \text{act}(|\text{print"}, v))$ is executed,  
   
   $\text{begin}_{\text{monitor}(\text{act}(|\text{print"}, v))}$ gets added to the trace resulting in a trace of $[(\infty), \text{begin}_{\text{print}(v)}, \text{begin}_{\text{monitor}(\text{act}(|\text{print"}, v))}]$  
   
   3, 4, Rule \text{callNonMonitoredFunctions}

6. after expression added by Rule \text{callFromApplication} are fully executed, expression $\text{print}(v)$ will be executed next  
   
   Rule \text{callFromApplication}

7. after the execution of $\text{print}(v)$, the trace is  
   
   $[(\infty), \text{begin}_{\text{print}(v)}, \text{begin}_{\text{monitor}(\text{act}(|\text{print"}, v))}, (\infty), \text{end}_{\text{print}(x);v}, (\infty)]$  
   
   5, 6, Rule \text{endLabelValue}

\[\square\]
8. execution of $p_1$ results in a trace of $[(\infty), \begin{array}{ll} \text{begin}_{\text{print}}(v), \text{begin}_{\text{monitor}}(\text{act}(\text{"print"}, v)), \text{end}_{\text{print}}(x), (\infty) \end{array}]$

9. execution of $p_1$ results in a trace of that matches $e_{\text{preOb}}$

8. Rules of $\infty$-expression

Case 2: post-obligation:

1. assume $p_2$ has a policy $Pol_2$ with the onTrigger
   $e_{\text{onTrigger}}(e) = \text{case } e \text{ of } \text{res } r \Rightarrow \text{if } r.\text{act.name }== \text{"print" then } \text{call} (\text{log}, e) \text{ else unit } \text{act } a \Rightarrow \text{unit}$
   Assumption
2. $\text{print} \in F$
3. when $e_{\text{app}}$ attempts to execute the expression $\text{call}(\text{print}, v)$, the $\text{call}(\text{monitor}, \text{res}(\text{"print"}, v))$ will be triggered
4. since $Pol_2$ does not consider $\text{call}(\text{print}, v)$ as security relevant, $\text{setOutput()}$ is never called in $Pol_2$
5. $\text{call}(\text{print}, v)$ will be executed
6. after $\text{call}(\text{print}, v)$ is executed $\text{end}_{\text{log}}(v)$ gets added to the trace resulting in a trace of $[(\infty), \text{end}_{\text{log}}(v)]$
7. a successful execution of $\text{call}(\text{print}, v)$ triggers $\text{call}(\text{monitor}, \text{res}(\text{act}(\text{"print"}, v), r))$
8. after a successful execution of $\text{call}(\text{monitor}, \text{res}(\text{act}(\text{"print"}, v), r))$, the trace is $[(\infty), \text{end}_{\text{log}}(v); r, (\infty), \text{begin}_{\text{monitor}}(\text{res}(\text{act}(\text{"print"}, v)), r)]$
9. execution of $p_2$ results in a trace of $[(\infty), \text{end}_{\text{log}}(v); r, (\infty), \text{begin}_{\text{monitor}}(\text{res}(\text{act}(\text{print}, v)), r)]$
10. execution of $p_2$ results in a trace that matches $e_{\text{postOb}}$

8. Rule callFromApplication, Rule while
5. Rule callFromObligation, Rule endLabelValue
7. Rule callNonMonitored-Functions
9. Rules of $\infty$-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 $\infty$-expression $re_{\text{preOb}}$ and $re_{\text{postOb}}$
   Case 1, Case 2
3. execution of $p_3$ matches the $\infty$-expression $re_{\text{ongoingOb}}$
   2. Rules of $\infty$-expression

Theorem 8 (Policy Permutability).

If $p$ is a well-typed program with monitor $(\text{fun}_{\text{mon}}, p_1 :: \cdots :: p_n, e_{\text{os}}, e_{\text{vc}})$ and $p \rightarrow^* p'$ while producing trace $t$, then there exists $e_{\text{os}}$ and $e_{\text{vc}}$ such that if $q$ is a well-typed program with monitor $(\text{fun}_{\text{mon}}, p'_1 :: \cdots :: p'_n, e_{\text{os}}, e_{\text{vc}})$ 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 $e_{\text{onTrigger}}, e_{\text{onObligation}}, e_{\text{os}}$, and $e_{\text{vc}}$ must be pure functions).

Proof.
1. only \( \text{call}(e_1, e_2) \) and \( \text{makeCFG}(e) \) expressions can result in values being added to traces

2. the parts of \( t \) and \( t' \) generated outside of calls to the monitor are identical

3. calls to monitor from \( p \) and \( q \) will differ only in the order of policies in parameter

4. calls to monitor from \( p \) and \( q \) result in traces that are equivalent if the current traces are equivalent

5. only lines 6, 10, 23, 25, 26 and 41 of monitor result in values being added to the trace

6. calls to the expression \( \text{makeCFG}(e) \) on line 6 of the monitor code in \( p \) and \( q \) will differ only in parameters

7. the traces result from calls to \( \text{makeCFG}(e) \) on line 6 in \( p \) and \( q \) will be equivalent if the current traces are equivalent

8. the traces result from the execution of the expression line 25 in \( p \) and \( q \) will be identical if \( pol \) in \( p \) and \( q \) are identical

9. \( pol \) on line 25 in \( p \) and \( q \) will be identical for each iteration of the loop if the values of the \( votingPols \) in \( p \) and \( q \) are identical

10. \( votingPols \) in \( p \) and \( q \) are identical if \( votingPolsList \) in \( p \) and \( q \) are identical

11. \( votingPolsList \) in \( p \) and \( q \) will be identical at each iteration if the \( votingPolsList \) in \( p \) is initially assigned with a value that is identical to the value that is assigned to \( votingPolsList \) in \( q \)

12. \( votingPolsList \) in \( p \) and \( q \) will be assigned with an identical value if the results of \( \text{call}(os, p_1 :: \cdots :: p_N) \) in \( p \) and the result of \( \text{call}(os, p'_1 :: \cdots :: p'_N) \) in \( q \) are identical

13. assume the results of \( \text{call}(os, p_1 :: \cdots :: p_N) \) in \( p \) and the result of \( \text{call}(os, p'_1 :: \cdots :: p'_N) \) in \( q \) will be identical when \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

14. the traces result from the execution of the expression line 25 in \( p \) and \( q \) will be identical if \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

15. the traces result from the execution of the expression line 25 in \( p \) and \( q \) will be identical when the values of \( votes \) in \( p \) and \( q \) are identical
16. *votes* in *p* and *q* will be identical if traces produced on line 25 of *p* and *q* are identical

17. the traces result from each iteration of line 25 will be identical when \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

18. the traces result from the execution of the expression line 24 in *p* and *q* will be identical when \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

19. the traces result from the execution of the case expression that spans from lines 25 to 26 in *p* and *q* will be identical for each iteration if the results of \( \text{call}(c.vc, !votes) \) in *p* and *q* on line 24 are identical as well as \( ob \) in *p* and *q* are identical

20. the traces result from the execution of the \( \text{call}(vc, !votes) \) expression in *p* and *q* will be identical when \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

21. \( ob \) in *p* and *q* will be identical for each iteration if *pols* on line 2 of the monitor code in *p* and *q* are identical

22. the traces result from the execution of the expressions span from lines 25 to 26 in *p* and *q* will be identical for each iteration with \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

23. the traces result from the execution of the expressions on line 9 for each iteration in *p* and *q* will be identical if *pols* in *p* and *q* are identical

24. the traces result from the execution of the expressions on line 9 for each iteration in *p* and *q* will be identical if \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \)

25. the traces result from the execution of the expressions on line 42 for each iteration in *p* and *q* will be identical if *votingPols* in *p* and *q* are identical

26. *votingPols* in *p* and *q* will be identical when \( e_{os}(obs) \) is defined as \( \text{call}(orderByPolicyName, obs) \)

27. the traces result from the execution of the expressions on line 42 in *p* and *q* will be identical when \( e_{os}(obs) \) is defined as \( \text{call}(orderByPolicyName, obs) \)

28. when \( e_{os}(obs) = \text{call}(orderByPolicyName, obs) \), \( t = t' \)
Appendix C: Copyright Permissions

The following permission is for the use of the content presented in Chapters 1–7 and 9 of this dissertation.
Title of the Work: PoCo: A Language for Specifying Obligation-Based Policy Compositions
Submission ID: A98
Author/Presenter(s): Danielle Ferguson; University of South Florida, USA; Yan Albright; University of South Florida, USA; Daniel Lomsak; University of South Florida, USA; Tyler Hanks; University of South Florida, USA; Kevin Orr; University of South Florida, USA; Jay Ligatti; University of South Florida, USA
Type of material: Full Paper
Publication and/or Conference Name: 2020 9th International Conference on Software and Computer Applications Proceedings

1. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author.

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal
Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work. (x) If your paper is withdrawn before it is published in the ACM Digital Library, the rights revert back to the author(s).

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

NOTE: For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

Please put the following LaTeX commands in the preamble of your document - i.e., before \begin{document}:

\copyrightyear{2020}
\acmYear{2020}
\setcopyright{acmcopyright}
\acmConference[ICSCA 2020]{2020 9th International Conference on Software and Computer Applications}{February 18--21, 2020}{Langkawi, Malaysia}
\acmBooktitle{2020 9th International Conference on Software and Computer Applications (ICSCA 2020), February 18--21, 2020, Langkawi, Malaysia}
\acmPrice{15.00}
\acmDOI{10.1145/3384544.3384585}
\acmISBN{978-1-4503-7665-5/20/02}

NOTE: For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

If you are using the ACM Interim Microsoft Word template, or still using or older versions of the ACM SIGCHI template, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library. Once you have your camera ready copy ready, please send your source files and PDF to your event contact for processing.

☐ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? ☐ Yes ☐ No

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ☐ Yes ☐ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ☐ Yes ☐ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.
VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☐ I agree to the Representations, Warranties and Covenants

DATE: 02/24/2020 sent to danid0729@gmail.com at 14:02:42