Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects

NOAM ZILBERSTEIN, Cornell University, USA
ANGELINA SALILING, Cornell University, USA
ALEXANDRA SILVA, Cornell University, USA

Separation logic's compositionality and local reasoning properties have led to significant advances in scalable static analysis. But program analysis has new challenges—many programs display *computational effects* and, orthogonally, static analyzers must handle *incorrectness* too. We present Outcome Separation Logic (OSL), a program logic that is sound for both correctness and incorrectness reasoning in programs with varying effects. OSL has a frame rule—just like separation logic—but uses different underlying assumptions that open up local reasoning to a larger class of properties than can be handled by any single existing logic.

Building on this foundational theory, we also define symbolic execution algorithms that use bi-abduction to derive specifications for programs with effects. This involves a new *tri-abduction* procedure to analyze programs whose execution branches due to effects such as nondeterministic or probabilistic choice. This work furthers the compositionality promised by separation logic by opening up the possibility for greater reuse of analysis tools across two dimensions: bug-finding vs verification in programs with varying effects.

1 INTRODUCTION

Compositional reasoning using separation logic [Reynolds 2002] and bi-abduction [Calcagno et al. 2009] has helped scale static analyses to industrial software with hundreds of millions of lines of code, making it possible to analyze code changes without disrupting the fast-paced engineering culture that developers are accustomed to [Calcagno et al. 2015; Distefano et al. 2019].

While the ideal of fully automated program verification remains elusive, analysis tools can boost confidence in code correctness by ensuring that a program will not *go wrong* in a variety of ways. In languages like C or C++, this includes ensuring that a program will not crash due to a segmentation fault or leak memory. However, a static analyzer *failing* to prove the absence of bugs does not imply that the program is incorrect; it could be a false positive.

Many programs are, in fact, incorrect. Analysis tools capable of finding bugs are thus in some cases *more useful* than verification tools, since the reported errors lead directly to tangible code improvements [Le et al. 2022]. Motivated by the need to identify bugs, Incorrectness Logic [O'Hearn 2020] and Incorrectness Separation Logic (ISL) [Raad et al. 2020, 2022] were recently introduced.

While ISL enjoys compositionality just like separation logic, the semantics of SL and ISL are incompatible—specifications and analysis tools cannot readily be shared between them. Further, the soundness of local reasoning in each separation logic variant relies on particular assumptions, meaning that no single program logic has a frame rule that can handle all of the following:

Computational Effects. The idea that program commands must be *local actions* [Calcagno et al. 2007] is central to the frame rule. In order to achieve this, the semantics of memory allocation is often forced to be nondeterministic [Yang and O'Hearn 2002], but this approach is not suitable for local reasoning in alternative execution models such as probabilistic computation.

May and Must properties. Separation logic can only express properties that *must* occur, whereas ISL can only express properties that *may* occur. In order to capture both correctness and incorrectness properties, we must handle both *may* and *must* properties.

***Under-approximation*.** Efficiently reasoning about incorrectness involves the ability to inspect a subset of the program paths—the ones that lead to a bug. But many separation logics are *fault*

Authors' addresses: Noam Zilberstein, noamz@cs.cornell.edu, Cornell University, USA; Angelina Saliling, ajs649@cornell.edu, Cornell University, USA; Alexandra Silva, alexandra.silva@cornell.edu, Cornell University, USA.

avoiding; the precondition must specify the resources needed to execute all traces, and therefore the entire program must be inspected.

In this paper, we show that local reasoning is sound under new assumptions, which do not force any particular evaluation model and are compatible with both correctness and incorrectness reasoning. To that end, we introduce *Outcome Separation Logic* (OSL), a single program logic that can handle all of the aforementioned scenarios. Our contributions are as follows.

Outcome Separation Logic. Our main contribution is Outcome Separation Logic (OSL), an extension of the recently introduced Outcome Logic (OL) [Zilberstein et al. 2023] for reasoning about pointer programs. While OL already supports correctness and incorrectness reasoning in programs with a variety of effects and much of the metatheory and inference rules carry over from it, the key novelty of OSL is that it is fundamentally based on separation-logic-style heap assertions, which allowed us to develop a frame rule to enable local reasoning and compositional symbolic execution.

This was no simple feat; while the soundness of the frame rule is intricate and nuanced even in partial correctness Hoare Logic [Yang and O'Hearn 2002], moving to a logic that supports a variety of assertions about termination, reachability, and probabilistic reasoning complicates the story significantly. OSL is the first program logic that supports local reasoning for both correctness and incorrectness, with the ability to also under-approximate program paths. On top of that, using an algebraic representation of choice, the soundness proof of our frame rule extends to other execution models too, such as probabilistic programs.

Symbolic execution algorithms. As a proof of concept for how OSL enables the consolidation of correctness and incorrectness analysis, we present symbolic execution algorithms to analyze C-like pointer programs. The core algorithm finds all the reachable outcomes, and is therefore tailored for correctness reasoning. We also define a *single-path* variant—modeled after Incorrectness Logic based bug-finding algorithms (Pulse [Raad et al. 2020] and Pulse-X [Le et al. 2022])—inferring specifications in which the postcondition is just one of the (possibly many) outcomes. While enjoying the scalability benefit of dropping paths, our algorithm can also soundly re-use procedure summaries generated by the correctness algorithm so as to not re-analyze the same procedure.

These algorithms are based on bi-abduction, which handles *sequential* programs by reconciling the postcondition of one precomputed spec with the precondition of the next Calcagno et al. [2009, 2011]. Since programs with effects are not purely sequential, but rather have branching that arises from, *e.g.,* nondeterministic or probabilistic choice, we also introduce *tri-abduction*, a new form of inference for composing branches in an effectful program.

We begin in Section 2 by outlining the challenges of local reasoning in a highly expressive program logic. Next, in Sections 3 and 4, we define Outcome Separation Logic (OSL), show three instantiations, and prove the soundness of the frame rule. In Section 5, we define our symbolic execution algorithm and tri-abduction, which is inspired by bi-abduction but is used for branching rather than sequential composition. In Section 6, we examine two case studies to show the applicability of these algorithms and finally we conclude in Sections 7 and 8 by discussing outlooks and related work.

2 LOCAL REASONING FOR CORRECTNESS AND INCORRECTNESS WITH EFFECTS

We begin by examining how the local reasoning principles of separation logic, along with bi-abductive inference, underlie scalable analysis techniques. These analyses symbolically execute programs and report the result as Hoare Triples $\{P\}\ C\ \{Q\}$: the postcondition Q describes any result of running C in a state satisfying the precondition P [Hoare 1969]. Hoare triples are compositional;

a specification for the sequence of two commands is constructed from specifications for each one.

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1 \fatsemi C_2\ \{R\}}\textsc{Sequence}$$

The Sequence rule is a good starting point for building scalable program analyses, but it is not quite compositional enough. The postcondition of C_1 must exactly match the precondition of C_2, making it difficult to apply this rule, particularly if C_1 and C_2 are procedure calls for which we already have pre-computed summaries (in the form of Hoare Triples), none of which exactly match. In response, separation logic offers a second form of (spatial) compositionality via the Frame rule, which adds information about unused program resources F to the pre- and postcondition of a completed proof.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\textsc{Frame}$$

But framing does not immediately answer how to sequentially compose triples. Given $\{P_1\}\ C_1\ \{Q_1\}$ and $\{P_2\}\ C_2\ \{Q_2\}$, it is unclear what—*if anything*—can be added to make Q_1 match P_2. This is where bi-abduction comes in—a technique that finds a missing resource M and leftover frame F to make the entailment $Q_1 * M \vDash P_2 * F$ hold. Using bi-abduction, we get a more usable sequence rule that stitches together two precomputed summaries without reexamining either program fragment.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad Q_1 * M \vDash P_2 * F \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * M\}\ C_1 \fatsemi C_2\ \{Q_2 * F\}}\textsc{Bi-Abductive Sequence}$$

While bi-abduction has enabled industrial strength static analyzers to scale to massive codebases, current developments use disjoint algorithms for correctness vs incorrectness, and do not support computational effects such as probabilistic choice. In the remainder of this section, we will examine why this is the case and explain how our logic allows for unified bi-abductive analysis algorithms.

2.1 Interlude: Reasoning about Effects and Incorrectness

While identifying bugs in pure programs is already challenging, effects add more complexity. This is demonstrated below; the program crashes because it attempts to dereference a null pointer.

$$\{\mathsf{ok} : x = \mathsf{null}\}\ [x] \leftarrow 1\ \{\mathsf{er} : x = \mathsf{null}\}$$

The specification is semantically straightforward; if x is null then the program will crash.[1] But now, rather than dereferencing a pointer that is *known* to be invalid, suppose we dereference a pointer that *might* be invalid, and—crucially—whether or not it is allocated comes down to nondeterminism. The following is one such scenario; x is obtained using malloc, which either returns a valid pointer or null. In Hoare Logic, the best we can do is specify this program using a disjunction.

$$\{\mathsf{ok} : \mathsf{emp}\}\ x := \mathsf{malloc}() \fatsemi [x] \leftarrow 1\ \{(\mathsf{ok} : x \mapsto 1) \vee (\mathsf{er} : x = \mathsf{null})\}$$

While the above specification hints that the program has a bug, it is in fact inconclusive since the disjunctive postcondition does not guarantee that both outcomes are *reachable* by actual program executions. Hoare Logic is fundamentally unable to characterize this bug, since the postcondition must describe *all* possible end states of the program; we cannot express something that *may* happen.

Two solutions for characterizing true bugs have been proposed. The first one is Incorrectness Logic (IL), which uses an alternative semantics to express that all states described by the postcondition are reachable from a state described by the precondition [O'Hearn 2020]. Specifying the

[1] Caveat: nontermination is an effect, and the typical *partial correctness* interpretation of SL is not suitable for incorrectness.

aforementioned bug is possible using Incorrectness Logic; the following triple stipulates that all the states described by the postcondition are reachable, including ones where the error occurs.

$$[\mathsf{ok} : \mathsf{emp}]\ x \coloneqq \mathsf{malloc}() \fatsemi [x] \leftarrow 1\ [(\mathsf{ok} : x \mapsto 1) \vee (\mathsf{er} : x = \mathsf{null})]$$

A variant of IL has a frame rule [Raad et al. 2020] and can underlie bi-abductive symbolic execution algorithms [Le et al. 2022]. However—just like separation logic—IL is specialized to nondeterministic programs and is not suitable for other effects. In addition, being inherently under-approximate, the semantics of IL cannot capture correctness properties, which must cover all the reachable outcomes. As such, different analyses and procedure summaries must be used for verification vs bug-finding.

In this paper, we take a different approach based on Outcome Logic (OL), which is compatible with both correctness and incorrectness while also supporting a variety of monadic effects [Zilberstein et al. 2023]. OL is similar to Hoare Logic, but the pre- and postconditions of triples describe *collections* of states rather than individual ones. A new logical connective $\oplus$—the outcome conjunction—guarantees the reachability of multiple outcomes. For instance, the aforementioned bug can be characterized using the following OL specification by replacing the disjunction in the postcondition.

$$\langle \mathsf{ok} : \mathsf{emp}\rangle\ x \coloneqq \mathsf{malloc}() \fatsemi [x] \leftarrow 1\ \langle(\mathsf{ok} : x \mapsto 1) \oplus (\mathsf{er} : x = \mathsf{null})\rangle$$

The above triple says that running the program in the empty heap will result in two reachable outcomes. In this case, the program is nondeterministic and its semantics is accordingly characterized by a *set* of program states S. The outcome conjunction tells us that there exist nonempty sets S_1 and S_2 with $S = S_1 \cup S_2$ such that $S_1 \vDash (\mathsf{ok} : x \mapsto 1)$ and $S_2 \vDash (\mathsf{er} : x = \mathsf{null})$. Since both outcomes are satisfied by nonempty sets, we know that they are both reachable by a real program trace.

However, for efficiency, specifying the bug above should not require recording information about the ok outcome. In incorrectness reasoning, it is desirable to *drop outcomes* so as to only explore some of the program paths [O'Hearn 2020; Le et al. 2022]. We achieve this in OL by replacing the extraneous outcome with $\top$, ensuring that the ok program path will not continue to be analyzed.

$$\langle \mathsf{ok} : \mathsf{emp}\rangle\ x \coloneqq \mathsf{malloc}() \fatsemi [x] \leftarrow 1\ \langle(\mathsf{er} : x = \mathsf{null}) \oplus \top\rangle$$

Outcomes apply to more effects too. For example, Outcome Logic can also be used to reason about probabilistic programs, where the (weighted) outcome conjunction additionally quantifies the likelihoods of outcomes. For example, the following program attempts to ping an IP address, which succeeds 99% of the time, and fails with probability 1% due to an unreliable network connection.

$$\langle \mathsf{ok} : \mathsf{true}\rangle\ x \coloneqq \mathsf{ping}(192.0.2.1)\ \langle(\mathsf{ok} : x = 0) \oplus_{99\%} (\mathsf{er} : x = 1)\rangle$$

Our goal in this paper is to extend local reasoning to Outcome Logic by augmenting it with a frame rule, and to use the resulting theory to build bi-abductive symbolic execution algorithms for both correctness (finding all reachable outcomes) and incorrectness (only exploring one program path at a time) in pointer programs with varying effects. We will next see the challenges behind designing a frame rule powerful enough to achieve that goal.

2.2 Designing a More Powerful Frame Rule

In the development of the frame rule, Yang and O'Hearn [2002] remarked that local reasoning "turns out to be surprisingly delicate, and it is not difficult to find situations where the rule doesn't work. So a careful treatment of soundness, appealing to the semantics of a specific language, is essential." Today, there are many frame rules that appeal to the semantics of their respective situations, relying on properties such as nondeterminism, partial correctness, or under-approximation for soundness.

Our challenge is to design a frame rule supporting *all* of those features, which means that fundamental questions must be addressed head on, without appealing to a *particular* semantic model. We follow the basic formula of standard separation logic [Yang and O'Hearn 2002; Calcagno et al. 2007], as it is semantically closest to Outcome Logic.[2]

Local Actions. Framing is sound if all program actions are *local* [Calcagno et al. 2007]. Roughly speaking, C is local if its behavior does not change as pointers are added to the heap. Most actions are inherently local, *e.g.*, dereferencing a pointer depends only on a single heap cell.

But memory allocation is—or at least, some implementations of it are—inherently non-local; allocating a new address involves reaching into an unknown region of the heap. To see why this is problematic, consider a deterministic allocator that always returns the smallest available address. Allocating a pointer in the empty heap is guaranteed to return the address 1, motivating the SL triple below on the left. However, applying the frame rule can easily invalidate this spec (right).

$$\{\mathsf{emp}\}\ x := \mathsf{alloc}()\ \{x = 1 \land x \mapsto -\} \qquad \frac{\{\mathsf{emp}\}\ x := \mathsf{alloc}()\ \{x = 1 \land x \mapsto -\}}{\{y \mapsto 1\}\ x := \mathsf{alloc}()\ \{x = 1 \land x \mapsto - * y \mapsto 1\}}\textsc{Frame}$$

Since y can have address 1, the fresh pointer x cannot also be equal to 1, so this application of the frame rule is clearly unsound. Making memory allocation nondeterministic can solve the problem [Yang and O'Hearn 2002]. If, in the above program, x could be assigned *any* fresh address, then the postcondition cannot say anything specific about *which* address we got. We cannot conclude that $x = 1$, but rather only that $x = 1 \lor x = 2 \lor \cdots$, which remains true in any larger heap.

Moving from Hoare Logic to Outcome Logic, this approach has two problems. First, Outcome Logic is parametric on an evaluation model, so the availability of nondeterminism is not a given; we need a strategy for allocation in the presence of *any* computational effects. Second, even with nondeterminism, locality is complicated by the ability to reason about reachable states. The problematic interaction between framing and reachability is displayed in the following example, which explicitly states that $x = 1$ is a reachable outcome of allocating x in the empty heap.

$$\frac{\langle \mathsf{ok} : \mathsf{emp}\rangle\ x := \mathsf{alloc}()\ \langle(\mathsf{ok} : x = 1) \oplus (\mathsf{ok} : x \neq 1)\rangle}{\langle \mathsf{ok} : y \mapsto 1\rangle\ x := \mathsf{alloc}()\ \langle(\mathsf{ok} : x = 1 \land y \mapsto 1) \oplus (\mathsf{ok} : x \neq 1 \land y \mapsto 1)\rangle}\textsc{Frame}$$

This inference is invalid; the precondition does not preclude y having address 1, in which case $x = 1$ is no longer a reachable outcome. In OSL, we acknowledge that memory allocation is non-local and instead ensure that triples cannot specify the result of an allocation too finely. This is achieved by altering the semantics of a triple $\langle\varphi\rangle\ C\ \langle\psi\rangle$ to require that if φ holds in some initial state, then ψ holds after running C using *any* allocation semantics. By universally quantifying the allocator, we ensure that any concrete semantics is captured, while retaining soundness of the frame rule.

Fault Avoidance. Typical formulations of separation logic are *fault avoiding*, meaning that the precondition must imply that the program execution does not encounter a memory fault [Reynolds 2002; Yang and O'Hearn 2002]. Unlike in Hoare Logic, the triple $\{\mathsf{true}\}\ C\ \{\mathsf{true}\}$ is not necessarily valid, which is crucial to the frame rule. If the triple $\{\mathsf{true}\}\ [x] \leftarrow 1\ \{\mathsf{true}\}$ were valid, then the frame rule would give us $\{x \mapsto 2 * \mathsf{true}\}\ [x] \leftarrow 1\ \{x \mapsto 2 * \mathsf{true}\}$, which is clearly false.

The problem with fault avoidance is that it involves inspecting the entire program, whereas OSL must be able to ignore some paths to more efficiently reason about incorrectness. Fortunately, OSL preconditions need not be safe. Following the previous example, $\langle \mathsf{ok} : \mathsf{true}\rangle\ [x] \leftarrow 1\ \langle \mathsf{ok} : \mathsf{true}\rangle$ is not a valid OSL specification since $\mathsf{ok} : \mathsf{true}$ is not a *reachable* outcome of running the program. Rather, if the precondition of an OSL triple is unsafe, then the postcondition can only be $\top$, *i.e.,* $\langle \mathsf{ok} : \mathsf{true}\rangle\ [x] \leftarrow 1\ \langle\top\rangle$. Framing information into the latter triple is perfectly sound since the $\top$ will absorb *any* outcome, including undefined behavior: $\langle \mathsf{ok} : x \mapsto 2 * \mathsf{true}\rangle\ [x] \leftarrow 1\ \langle\top\rangle$.

[2] We remark that other proof strategies exist including using heap monotonicity (ISL) and frame baking (higher-order separation logics), a comparison to these approaches is available in Section 7.

OSL allows us to decide how much of the memory footprint to specify. In a correctness analysis that covers all paths, the precondition must be safe for the entire program. If we instead want to reason about incorrectness and drop paths, then it must only be safe for the paths we explore.

2.3 Symbolic Execution and Tri-Abduction

OSL provides a logical foundation for symbolic execution algorithms that are capable of *both* verification *and* bug-finding. Our approach takes inspiration from industrial strength bi-abductive analyzers (Abductor and Infer [Calcagno et al. 2009, 2015]), but paying greater attention to effects, as the aforementioned tools may fail to find specifications for programs with control flow branching.

To see what goes wrong, let us examine a program that uses disjoint resources in the two nondeterministic branches: $([x] \leftarrow 1) + ([y] \leftarrow 2)$. Using bi-abduction, we could conclude that $x \mapsto -$ is a valid precondition for the first branch whereas $y \mapsto -$ is valid for the second, but there is no straightforward way to find a precondition valid for *both* branches. As a result, the program must be re-evaluated with each candidate precondition to ensure that they are safe for all branches.

Calcagno et al. [2011, §4.3] acknowledged this issue, and suggested fixing it by re-running the abduction procedure until nothing more can be added to each precondition. Rather than using two passes (as Abductor already does), this approach would require a pass for each combination of nondeterministic choices, which is exponential in the worst case. While a single-pass bi-abduction algorithm has been proposed [Sextl et al. 2023], it still cannot handle all cases (*e.g.,* see Remark 2).

We take a different approach, acknowledging that branching is fundamentally different from sequential composition and requires a new type of inference, which we call tri-abduction. As its name suggests, tri-abduction infers three components (to bi-abduction's two). Given P_1 and P_2—preconditions for two branches—the goal is to find a single anti-frame M and two leftover frames F_1 and F_2 such that $M \vDash P_1 * F_1$ and $M \vDash P_2 * F_2$, in order to compose the summaries for two program branches, as demonstrated below.

$$\frac{\langle P_1\rangle\ C_1\ \langle Q_1\rangle \qquad P_1 * F_1 \Dashv M \vDash P_2 * F_2 \qquad \langle P_2\rangle\ C_2\ \langle Q_2\rangle}{\langle M\rangle\ C_1 + C_2\ \langle (Q_1 * F_1) \oplus (Q_2 * F_2)\rangle}\ \textsc{Tri-Abductive Composition}$$

Tri-abduction does not replace bi-abduction; they work in complementary ways—bi-abduction is used for sequential composition whereas tri-abduction composes branches arising from effects.

In addition, we are interested in bug-finding algorithms, which—similar to Pulse and Pulse-X [Raad et al. 2020; Le et al. 2022]—do not traverse all the program paths. We achieve this using a single-path version of the algorithm, producing summaries of the form $\langle \mathsf{ok} : P\rangle\ C\ \langle (\mathsf{er} : Q) \oplus \top\rangle$, with only a single outcome specified and the remaining ones covered by $\top$. The soundness of the single-path approach is motivated by the fact that $P \oplus Q \Rightarrow P \oplus \top$; extraneous outcomes can be converted to $\top$, ensuring that those paths will not be explored. Just like in Pulse and Pulse-X, this ability to *drop outcomes* allows the analysis to retain less information for increased scalability.

We formalize these concepts starting in Sections 3 and 4, where we define a program semantics and Outcome Separation Logic and prove the soundness of the frame rule. Symbolic execution and tri-abduction are defined in Section 5, and we examine case studies in Section 6.

3 PROGRAM SEMANTICS

We begin the technical development by defining the semantics for the underlying programming language of Outcome Separation Logic. All instances of OSL share the same program syntax, but this syntax is interpreted in different ways, corresponding to the choice mechanisms dictated by each instance's computational effects. The syntax of the language is given below.

$$\begin{aligned}
\mathsf{Cmd} \ni C &::= \mathsf{skip} \mid C_1 \fatsemi C_2 \mid C_1 + C_2 \mid \mathsf{assume}\ e \mid \mathsf{while}\ e\ \mathsf{do}\ C \mid c \\
\mathsf{Act} \ni c &::= x := e \mid x := \mathsf{alloc}() \mid \mathsf{free}(e) \mid [e_1] \leftarrow e_2 \mid x \leftarrow [e] \mid \mathsf{error}() \mid f(\vec{e}) \\
\mathsf{Exp} \ni e &::= x \mid \kappa \mid e_1 \diamond e_2 \mid \neg e \qquad\qquad (x \in \mathsf{Var}, \kappa \in \mathsf{Val}, f \in \mathsf{Proc}, \diamond \in \mathsf{BinOp})
\end{aligned}$$

Commands $C \in \mathsf{Cmd}$ are similar to those of Dijkstra's [1975] Guarded Command Language (GCL), containing skip, sequencing ($C_1 \fatsemi C_2$), choice ($C_1 + C_2$) and while loops. Given a test e (*i.e.,* a Boolean-valued expression), conditionals are defined as syntactic sugar in the typical way: $\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \triangleq (\mathsf{assume}\ e \fatsemi C_1) + (\mathsf{assume}\ \neg e \fatsemi C_2)$. But—unlike GCL—assume also accepts expressions that are interpreted over *weights* drawn from a set that has additional algebraic properties described in Section 3.1 (and can also encode the Booleans). For example, weights in randomized programs are probabilities $p \in [0, 1]$, and we can accordingly define a probabilistic choice operator $C_1 +_p C_2 \triangleq (\mathsf{assume}\ p \fatsemi C_1) + (\mathsf{assume}\ 1 - p \fatsemi C_2)$, which runs C_1 with probability p and C_2 with probability $1 - p$.

Atomic actions $c \in \mathsf{Act}$ can assign to variables ($x := e$), allocate ($x := \mathsf{alloc}()$) and deallocate ($\mathsf{free}(e)$) memory, write ($[e_1] \leftarrow e_2$) and read ($x \leftarrow [e]$) pointers, crash ($\mathsf{error}()$), and call procedures ($f(\vec{e})$). Expressions can be variables $x \in \mathsf{Var}$, constants $\kappa \in \mathsf{Val}$ (*e.g.,* integers, Booleans, and *weights*), a variety of binary operations $e_1 \diamond e_2$ where $\diamond \in \mathsf{BinOp} = \{+, -, =, <, \ldots\}$, or logical negation $\neg e$.

In the remainder of this section, we will formally define denotational semantics for the language above. This will first involve discussing the algebraic properties of the program weights, after which we can define a (monadic) execution model to interpret sequential composition.

3.1 Algebraic Preliminaries

We first recall the definitions of some algebraic structures that will be used to instantiate the program semantics for different execution models. *Monoids* model combining and scaling outcomes.

Definition 3.1 (Monoid). A monoid $\langle A, +, \mathbb{0} \rangle$ consists of a carrier set A, an associative binary operator $+\colon A \times A \to A$, and an identity element $\mathbb{0} \in A$ such that $a + \mathbb{0} = \mathbb{0} + a = a$ for all $a \in A$. Additionally, a monoid is *partial* if $+$ is partial ($+\colon A \times A \rightharpoonup A$) and it is *commutative* if $a + b = b + a$.

For example, $\langle [0, 1], +, 0 \rangle$ is a partial commutative monoid that is commonly used in probabilistic computation since probabilities come from the interval $[0, 1]$ and addition is undefined if the sum is greater than 1. Scalar multiplication $\langle [0, 1], \cdot, 1 \rangle$ is another monoid with with same carrier set, but it is total rather than partial. Two monoids can be combined to form a semiring, as follows.

Definition 3.2 (Semiring). A semiring $\langle A, +, \cdot, \mathbb{0}, \mathbb{1} \rangle$ consists of a carrier set A, along with an addition operator $+$, a multiplication operator $\cdot$ and two elements $\mathbb{0}, \mathbb{1} \in A$ such that:

(1) $\langle A, +, \mathbb{0} \rangle$ is a commutative monoid.
(2) $\langle A, \cdot, \mathbb{1} \rangle$ is a monoid (we sometimes omit $\cdot$ and write $a \cdot b$ as ab).
(3) Multiplication distributes over addition: $a \cdot (b + c) = ab + ac$ and $(b + c) \cdot a = ba + ca$
(4) $\mathbb{0}$ is the annihilator of multiplication: $a \cdot \mathbb{0} = \mathbb{0} \cdot a = \mathbb{0}$

A semiring is *partial* if $\langle A, +, \mathbb{0} \rangle$ is instead a partial commutative monoid (PCM), but multiplication remains total. In the case of a partial semiring, distributive rules only apply if $b + c$ is defined.

We now define *Outcome Algebras* that give the interpretation of choice and weighting. The carrier set A is used to represent the weight of an outcome. In deterministic and nondeterministic evaluation models, this weight can be 0 or 1 (a Boolean), but in the probabilistic model, it can be any probability in $[0, 1]$. The rules for combining these weights vary by execution model.

Definition 3.3 (Outcome Algebra). An *Outcome Algebra* is a structure $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$, which is a complete, Scott continuous, naturally ordered, partial semiring, and:

(1) $\langle A, \leq\rangle$ is a complete partial order (cpo) and $\sup(A) = \mathbb{1}$.
(2) If $\sum_{i\in I} a_i$ is defined, there exist $(b_i)_{i\in I}$ such that $\sum_{i\in I} b_i = \mathbb{1}$ and $\forall i \in I. a_i = (\sum_{j\in I} a_j) \cdot b_i$.

Appendix A defines Scott continuity, completeness, and more details about property (2).

Outcome Algebras can encode the following three interpretations of choice:

Definition 3.4 (Deterministic Outcome Algebra). A deterministic program has *at most* one outcome (zero if it diverges). To encode this, we use an Outcome Algebra $\langle\{0, 1\}, +, \cdot, 0, 1\rangle$ where the elements $\{0, 1\}$ are Booleans indicating whether or not an outcome has occurred. The sum operation is usual integer addition, but is undefined for $1 + 1$, since two outcomes cannot simultaneously occur in a deterministic setting, and $\cdot$ is typical integer multiplication.

Definition 3.5 (Nondeterministic Outcome Algebra). The Boolean semiring $\langle\{0, 1\}, \vee, \wedge, 0, 1\rangle$ represents nondeterminism. Similar to Definition 3.4, the elements indicate whether an outcome has occurred, but now addition is logical disjunction so that multiple outcomes can occur.

Definition 3.6 (Probabilistic Outcome Algebra). Let $\langle[0, 1], +, \cdot, 0, 1\rangle$ be an outcome algebra where $+$ is real-valued addition (and undefined if $a + b > 1$) and $\cdot$ is real-valued multiplication. The carrier set $[0, 1]$ indicates that each outcome has a probability of occurring.

In the style of Moggi [1991], the language semantics is monadic in order to sequence effects. We now show how to construct a monad given any Outcome Algebra.

Definition 3.7 (Outcome Monad). Given an Outcome Algebra $\mathcal{A} = \langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$, we define a monad $\langle\mathcal{W}_{\mathcal{A}}, \mathsf{unit}, \mathsf{bind}\rangle$, where $\mathcal{W}_{\mathcal{A}}(S) = \{m : S \to A \mid \sum_{s\in\mathsf{supp}(m)} m(s) \text{ is defined}\}$ is the set of countably supported *weighting* functions and the monad operations are defined as follows:

$$\mathsf{unit}(s)(t) = \begin{cases} \mathbb{1} & \text{if } s = t \\ \mathbb{0} & \text{if } s \neq t \end{cases} \qquad\qquad \mathsf{bind}(m, f)(t) = \sum_{s\in\mathsf{supp}(m)} m(s) \cdot f(s)(t)$$

We also let $\mathsf{supp}(m) = \{s \mid m(s) \neq \mathbb{0}\}$ and $|m| = \sum_{s\in\mathsf{supp}(m)} m(s)$. This monad is similar to the Giry [1982] monad, but it is generalized to work over any partial semiring, rather than probabilities $[0, 1] \subseteq \mathbb{R}$. It is fairly easy to see that $\mathcal{W}_{\mathcal{A}}$ obeys the monad laws, given the semiring laws.

So, a weighting function $m \in \mathcal{W}_{\mathcal{A}}(S)$ assigns a weight $a \in A$ to each program state $s \in S$. Definitions 3.4 to 3.6 gave interpretations for $\mathcal{A}$ in which $\mathcal{W}_{\mathcal{A}}(S)$ encodes deterministic, nondeterministic, and probabilistic computation, respectively. In the (non)deterministic cases, $m(s) \in \{0, 1\}$, indicating whether or not s is present in the collection of outcomes m. Due to the interpretation of $+$ in Definition 3.4, the constraint that $\sum_{s\in\mathsf{supp}(m)} m(s)$ is defined guarantees that m can contain at most one outcome, whereas in the nondeterministic case, m can contain arbitrarily many. In the probabilistic case, $m(s) \in [0, 1]$ and gives the probability of the outcome s in the distribution m.

The semiring operations can be lifted to weighting functions. We will overload some notation to also refer to pointwise liftings as follows: $m_1 + m_2 = \lambda s.(m_1(s) + m_2(s))$, $\mathbb{0} = \lambda s.\mathbb{0}$, and $a \cdot m = \lambda s.(a \cdot m(s))$. When the nondeterministic algebra (Definition 3.5) is lifted in this way, the result is isomorphic to the powerset monad with $m_1 + m_2 \cong m_1 \cup m_2$ and $\mathbb{0} \cong \emptyset$.

Now, in order to represent errors and undefined states in the language semantics, we will define a monad transformer [Liang et al. 1995] based on the coproduct $S + E + 1$ where S is the set of program states, E is the set of errors, and we additionally include an undefined symbol. We define the following three injection functions, plus shorthand for the undefined element:

$$\mathbb{i}_{\mathsf{ok}} : S \to S + E + 1 \qquad \mathbb{i}_{\mathsf{er}} : E \to S + E + 1 \qquad \mathbb{i}_{\mathsf{undef}} : 1 \to S + E + 1 \qquad \mathsf{undef} = \mathbb{i}_{\mathsf{undef}}(\star)$$

Commands $\quad [\![-]\!]_{\mathsf{alloc}} : \mathsf{Cmd} \to \mathcal{S} \times \mathcal{H} \to \mathcal{W}_{\mathcal{A}}(\mathsf{St})$

$$
\begin{aligned}
[\![\mathsf{skip}]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{unit}(s,h)\\
[\![C_1 \mathbin{⨟} C_2]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{bind}([\![C_1]\!]_{\mathsf{alloc}}(s,h), [\![C_2]\!]_{\mathsf{alloc}})\\
[\![C_1 + C_2]\!]_{\mathsf{alloc}}(s,h) &= [\![C_1]\!]_{\mathsf{alloc}}(s,h) + [\![C_2]\!]_{\mathsf{alloc}}(s,h)\\
[\![\mathsf{assume}\ e]\!]_{\mathsf{alloc}}(s,h) &= [\![e]\!](s) \cdot \mathsf{unit}(s,h) \quad \text{if } [\![e]\!](s) \in A\\
[\![\mathsf{while}\ e\ \mathsf{do}\ C]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{lfp}(F_{\langle C,e,\mathsf{alloc}\rangle})(s,h)\\
\text{where } F_{\langle C,e,\mathsf{alloc}\rangle}(f)(s,h) &= \begin{cases} \mathsf{bind}([\![C]\!]_{\mathsf{alloc}}(s,h), f) & \text{if } [\![e]\!](s) = \mathbb{1}\\ \mathsf{unit}(s,h) & \text{if } [\![e]\!](s) = \mathbb{0}\end{cases}
\end{aligned}
$$

Actions $\quad [\![-]\!]_{\mathsf{alloc}} : \mathsf{Act} \to \mathcal{S} \times \mathcal{H} \to \mathcal{W}_{\mathcal{A}}(\mathsf{St})$

$$
\begin{aligned}
[\![x := e]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{unit}(s[x \mapsto [\![e]\!](s)], h)\\
[\![x := \mathsf{alloc}()]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{bind}_{\mathcal{W}}(\mathsf{alloc}(s,h), \lambda(\ell,v).\mathsf{unit}(s[x \mapsto \ell], h[\ell \mapsto v]))\\
[\![\mathsf{free}(e)]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{update}(s,h,[\![e]\!](s), s, h[[\![e]\!](s) \mapsto \bot])\\
[\![[e_1] \leftarrow e_2]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{update}(s,h,[\![e_1]\!](s), s, h[[\![e_1]\!](s) \mapsto [\![e_2]\!](s)])\\
[\![x \leftarrow [e]]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{update}(s,h,[\![e]\!](s), s[x \mapsto h([\![e]\!](s))], h)\\
[\![\mathsf{error}()]\!]_{\mathsf{alloc}}(s,h) &= \mathsf{error}(s,h)\\
[\![f(\vec{e})]\!]_{\mathsf{alloc}}(s,h) &= [\![C]\!]_{\mathsf{alloc}}(s[\vec{x} \mapsto [\![\vec{e}]\!](s)], h) \quad \text{where } (C,\vec{x}) = \mathsf{P}(f)
\end{aligned}
$$

Basic Operations

$$
\mathsf{error}(s,h) = \mathsf{unit}_{\mathcal{W}}(\mathbb{i}_{\mathsf{er}}(s,h)) \qquad \mathsf{update}(s,h,\ell,s',h') = \begin{cases} \mathsf{unit}(s',h') & \text{if } h(\ell) \in \mathsf{Val}\\ \mathsf{error}(s,h) & \text{if } h(\ell) = \bot \vee \ell = \mathsf{null}\\ \mathsf{unit}_{\mathcal{W}}(\mathsf{undef}) & \text{if } \ell \notin \mathsf{dom}(h)\end{cases}
$$

Fig. 1. Denotational semantics of program commands, parametric on an outcome algebra $\mathcal{A} = \langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$, an allocator function $\mathsf{alloc} : \mathcal{S} \times \mathcal{H} \to \mathcal{W}_{\mathcal{A}}(\mathsf{Addr})$, and a procedure table $\mathsf{P} \colon \mathsf{Proc} \to \mathsf{Cmd} \times \vec{\mathsf{Var}}$.

Borrowing the notation of Incorrectness Logic [O'Hearn 2020], we use ok and er to denote states in which the program terminated successfully or crashed, respectively. We will also write $\mathbb{i}_\epsilon$ to refer to one of the above injections, where $\epsilon \in \{\mathsf{ok}, \mathsf{er}\}$.

Definition 3.8 (Error Monad Transformer). Let $\langle \mathcal{W}_{\mathcal{A}}, \mathsf{bind}_{\mathcal{W}}, \mathsf{unit}_{\mathcal{W}}\rangle$ be the monad described in Definition 3.7 and E be a set of error states. We define a new monad $\langle \mathcal{W}_{\mathcal{A}}(- + E + 1), \mathsf{bind}, \mathsf{unit}\rangle$ where the monad operations are defined as follows:

$$
\mathsf{unit}(s) = \mathsf{unit}_{\mathcal{W}}(\mathbb{i}_{\mathsf{ok}}(s)) \qquad \mathsf{bind}(m,f) = \mathsf{bind}_{\mathcal{W}}\left(m, \lambda\sigma. \begin{cases} f(s) & \text{if } \sigma = \mathbb{i}_{\mathsf{ok}}(s)\\ \mathsf{unit}_{\mathcal{W}}(\sigma) & \text{otherwise}\end{cases}\right)
$$

3.2 Denotational Semantics

The semantics of commands $[\![-]\!]_{\mathsf{alloc}} : \mathsf{Cmd} \to \mathcal{S} \times \mathcal{H} \to \mathcal{W}_{\mathcal{A}}(\mathsf{St})$ is defined in Figure 1 and is parametric on an outcome algebra $\mathcal{A} = \langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$ and an allocator function $\mathsf{alloc} \in \mathsf{Alloc}_{\mathcal{A}}$, described below. The semantics of expressions $[\![-]\!] : \mathsf{Exp} \to \mathcal{S} \to \mathsf{Val}$ is omitted, but is defined in the obvious way. The set of states is $\mathsf{St} = \mathcal{S} \times \mathcal{H} + \mathcal{S} \times \mathcal{H} + 1$, where $\mathcal{S} = \{s : \mathsf{Var} \cup \mathsf{LVar} \to \mathsf{Val}\}$ are stores (assigning values to both program variables $x \in \mathsf{Var}$ and logical variables $X \in \mathsf{LVar}$) and $\mathcal{H} = \{h : \mathsf{Addr} \rightharpoonup \mathsf{Val} \cup \{\bot\}\}$ are heaps. In the style of Raad et al. [2020], a heap is both a *partial* mapping and also includes $\bot$ in the codomain, distinguishing between cases with no information about an address ($\ell \notin \mathsf{dom}(h)$) vs cases where a pointer is explicitly deallocated ($h(\ell) = \bot$). We additionally assume that $\{\mathsf{null}\} \cup A \subseteq \mathsf{Val}$, $\mathsf{Addr} \subseteq \mathsf{Val}$ and $\mathsf{null} \notin \mathsf{Addr}$.

$$\begin{array}{lll}
m \vDash \top & \text{always} \\
m \vDash \varphi \vee \psi & \text{iff} & m \vDash \varphi \quad \text{or} \quad m \vDash \psi \\
m \vDash \varphi \oplus \psi & \text{iff} & \exists m_1, m_2. \quad m = m_1 + m_2 \quad \text{and} \quad m_1 \vDash \varphi \quad \text{and} \quad m_2 \vDash \psi \\
m \vDash (\varphi)_a & \text{iff} & a = \mathbb{0} \text{ and } m = \mathbb{0} \quad \text{or} \quad \exists m'. \quad m = a \cdot m' \quad \text{and} \quad m' \vDash \varphi \\
m \vDash \epsilon : P & \text{iff} & |m| = \mathbb{1} \quad \text{and} \quad \forall \sigma \in \mathsf{supp}(m). \quad \exists (s,h). \quad \sigma = \mathbb{i}_\epsilon(s,h) \quad \text{and} \quad (s,h) \in P
\end{array}$$

Fig. 2. Semantics of outcome assertions given an outcome algebra $\langle A, +, \cdot, \mathbb{0}, \mathbb{1} \rangle$.

Allocators are functions mapping (s, h) to a collection of addresses and values with total mass $\mathbb{1}$, and each address ℓ is not in the domain of h. Formally, allocators come from the following set.

$$\mathsf{Alloc}_{\mathcal{A}} = \left\{ f \colon \mathcal{S} \times \mathcal{H} \to \mathcal{W}_{\mathcal{A}}(\mathsf{Addr} \times \mathsf{Val}) \;\middle|\; \forall (s,h). \begin{array}{l} |f(s,h)| = \mathbb{1} \quad \text{and} \\ \forall (\ell, v) \in \mathsf{supp}(f(s,h)).\ \ell \notin \mathsf{dom}(h) \end{array} \right\}$$

A deterministic allocator $\mathsf{alloc\_det}(s,h) = \mathsf{unit}(\min(\mathsf{Addr} \setminus \mathsf{dom}(h)), 0)$ that always picks the first unused address is valid in all OSL instances. Note that while logical variables $X \in \mathsf{LVar}$ cannot appear in program expressions e, allocators *can* depend on them. A global procedure table $\mathsf{P} \colon \mathsf{Proc} \to \mathsf{Cmd} \times \vec{\mathsf{Var}}$ that returns a command and vector of variable names (the arguments) given a procedure name $f \in \mathsf{Proc}$ is used to interpret procedure calls. We assume that all procedures used in programs are defined and pass the correct number of arguments.

The monad operations give semantics to skip and $\fatsemi$, while loops are defined as a least fixed point, and $C_1 + C_2$ is defined as a sum whose meaning depends on the Outcome Algebra. Since the semiring plus is partial, this sum may be undefined; in Appendix A we discuss simple syntactic checks to ensure totality. The semantics of assume e weights the branch by the value of e, as long as it is a valid program weight. Boolean expressions evaluate to the semiring $\mathbb{0}$ (false) or $\mathbb{1}$ (true).

We define two operations before giving the semantics of atomic actions: $\mathsf{error}(s,h)$ constructs an error state and $\mathsf{update}(s,h,\ell,s',h')$ returns (s',h') if the address ℓ is allocated in h, it returns an error if ℓ is deallocated, and returns undef if $\ell \notin \mathsf{dom}(h)$. Assignment is defined in the usual way by updating the program store; memory allocation uses alloc to obtain a fresh address and initial value (or collection thereof); deallocation, reads, and writes are implemented using update and errors use error. Procedure names are looked up in P to obtain C and $\vec{x}$ before running C on a store updated by setting $\vec{x}$ to have the values of the inputs $\vec{e}$. We will additionally occasionally use the Kleisli extension of the semantics, which is defined as $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) = \mathsf{bind}(m, [\![C]\!]_{\mathsf{alloc}})$.

4 OUTCOME SEPARATION LOGIC

We now proceed to formalize Outcome Separation Logic (OSL) and the frame rule. First, we define an assertion logic for the pre- and postconditions of outcome triples. These assertions are based on the outcome assertions of Zilberstein et al. [2023], using SL assertions as basic predicates.

Outcome Assertions. The syntax for OSL assertions is below and the semantics is in Figure 2. Both are parametric on an Outcome Algebra $\langle A, +, \cdot, \mathbb{0}, \mathbb{1} \rangle$.

$$\varphi ::= \top \mid \varphi \vee \psi \mid \varphi \oplus \psi \mid (\varphi)_a \mid \epsilon : P \qquad (a \in A, \epsilon \in \{\mathsf{ok}, \mathsf{er}\}, P \in 2^{\mathcal{S} \times \mathcal{H}})$$

Outcome assertions include familiar constructs such as $\top$ and disjunctions. The outcome conjunction $\varphi \oplus \psi$ splits m into two pieces $m_1 \vDash \varphi$ and $m_2 \vDash \psi$ summing to m. Weighting $(\varphi)_a$ guarantees that if $m \vDash \varphi$, then $a \cdot m \vDash (\varphi)_a$. Combining these, we also define probabilistic choice: $\varphi \oplus_p \psi \triangleq (\varphi)_p \oplus (\psi)_{1-p}$, meaning that φ occurs with probability p and ψ occurs with probability $1 - p$.

Finally, basic assertions $(\mathsf{ok} : P)$ and $(\mathsf{er} : Q)$ require that $|m| = \mathbb{1}$—ensuring that the set of outcomes is nonempty in the (non)deterministic cases (Definitions 3.4 and 3.5) or that the assertion occurs with probability 1 (Definition 3.6)—and that all states in $\mathsf{supp}(m)$ terminated successfully and satisfy P or crashed and satisfy Q, respectively, where $P, Q \in 2^{\mathcal{S} \times \mathcal{H}}$ are semantic heap assertions.

These heap assertions are defined as in standard separation logic, *e.g.*, $\mathsf{emp} \triangleq \{(s, \emptyset) \mid s \in \mathcal{S})\}$, $P * Q \triangleq \{(s, h_1 \uplus h_2) \mid (s, h_1) \in P, (s, h_2) \in Q\}$, and $e_1 \mapsto e_2 \triangleq \{(s, \{\llbracket e_1 \rrbracket (s) \mapsto \llbracket e_2 \rrbracket (s)\}) \mid s \in \mathcal{S}\}$.

We were motivated to pick this particular set of outcome assertions in light of our goal to define symbolic execution algorithms in the style of Calcagno et al. [2009], which compute procedure summaries of the form $\{P\}\ f(\vec{e})\ \{Q_1 \vee \cdots \vee Q_n\}$ and the disjunctive post indicates a series of possible outcomes. In our case, we exchange those disjunctions for outcome conjunctions in the cases where the outcomes arise due to computational effects. We include $\top$ in order to drop outcomes using the assertion $\varphi \oplus \top$ (as discussed in Section 2). Finally, we include disjunctions to express joins of outcomes that occur due to logical choice, and also to express partial correctness; while $\epsilon : P$ guarantees reachability, $(\epsilon : P) \vee (\top)_{\mathbb{0}}$ also permits nontermination (no outcomes).

OSL Triples. The semantics of OSL triples requires that the specification holds when the program is run using *any* allocator, ensuring that the postcondition cannot say anything specific about which addresses were obtained, as those addresses may change if the program is executed in a larger heap.

Definition 4.1 (Outcome Separation Logic Triples). For any outcome algebra $\mathcal{A} = \langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$:

$$\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \text{iff} \qquad \forall m \in \mathcal{W}_{\mathcal{A}}(\mathsf{St}), \mathsf{alloc} \in \mathsf{Alloc}_{\mathcal{A}}.\quad m \vDash \varphi \quad \implies \quad \llbracket C \rrbracket^{\dagger}_{\mathsf{alloc}}(m) \vDash \psi$$

The inference rules for OSL carry over from standard Outcome Logic [Zilberstein et al. 2023, Fig. 4], along with the small axioms of separation logic [O'Hearn et al. 2001]. It is also fairly straightforward to derive the rules from our symbolic execution algorithm (Figure 5). Instead of repeating the rules here, we discuss the new local reasoning principle that OSL supports, namely the frame rule.

4.1 The Frame Rule

We now build the necessary foundations to introduce and prove the soundness of the frame rule. First, we define a new separating conjunction as a transformation on outcome assertions, using the symbol $\circledast$ to distinguish it from the usual separating conjunction $*$ on symbolic heaps; $\circledast$ is a binary operation taking an outcome assertion and an SL assertion (rather than two SL assertions like $*$). The operation is defined below, where $\bowtie ::= \oplus \mid \vee$.

$$\top \circledast F \triangleq \top \qquad (\varphi \bowtie \psi) \circledast F \triangleq (\varphi \circledast F) \bowtie (\psi \circledast F) \qquad (\varphi)_a \circledast F \triangleq (\varphi \circledast F)_a \qquad (\epsilon : P) \circledast F \triangleq \epsilon : P * F$$

So, $\circledast$ has no effect on $\top$, it distributes over $\vee$, $\oplus$, and $(-)_a$, and for basic assertions $\epsilon : P$, we simply join $P * F$ with the regular separating conjunction. We can now express the frame rule.

$$\frac{\langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \mathsf{mod}(C) \cap \mathsf{fv}(F) = \emptyset}{\langle \varphi \circledast F \rangle\ C\ \langle \psi \circledast F \rangle}\ \textsc{Frame}$$

This rule resembles the frame rule of other separation logics, with the same side condition that F cannot mention any modified program variables. However, as we described in Section 2.2, OSL's expressivity goes beyond that of existing separation logics, meaning that this frame rule can be used for both may and must properties in nondeterministic programs, as well as quantitative properties in probabilistic programs. Further, all of these capabilities stem from a single soundness proof.

The key to the proof is the *frame property* (Lemma C.7), which roughly states that for any $(s, h) \in \mathcal{S} \times \mathcal{H}$, allocator alloc, and h' such that $(s, h') \vDash F$, we can construct a new allocator alloc' such that adding h' to each end state of $\llbracket C \rrbracket_{\mathsf{alloc}'} (s, h)$ gives us $\llbracket C \rrbracket_{\mathsf{alloc}} (s, h \uplus h')$ (modulo the undef states), guaranteeing that if $\llbracket C \rrbracket_{\mathsf{alloc}'} (s, h) \vDash \psi$, then $\llbracket C \rrbracket_{\mathsf{alloc}} (s, h \uplus h') \vDash \psi \circledast F$. We formalize this idea in the remainder of the section. It relies on *lifted relations*, defined below, which describe the relation between two weighted collections in terms of a relation on individual states.

Definition 4.2 (Relation Liftings). Given a relation $R \subseteq X \times Y$, we define a lifting of R on weighting functions $\overline{R} \subseteq \mathcal{W}_{\mathcal{A}}(X) \times \mathcal{W}_{\mathcal{A}}(Y)$ as follows:

$$\overline{R} = \left\{ (m_1, m_2) \;\middle|\; \exists m \in \mathcal{W}_{\mathcal{A}}(R). \quad m_1 = \lambda x. \sum_{y \in \mathsf{supp}(m_2)} m(x,y) \quad \text{and} \quad m_2 = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m(x,y) \right\}$$

Semantics of the Outcome Separating Conjunction. We now prove semantic properties about $\circledast$ that allow us to relate program configurations satisfying φ to ones that satisfy $\varphi \circledast F$. First, we need to relate states satisfying P to states satisfying $P * F$.

$$\mathsf{frame}(F) = \left\{ (\mathbb{1}_\epsilon(s,h), \mathbb{1}_\epsilon(s, h \uplus h')) \mid \mathbb{1}_\epsilon(s,h) \in \mathsf{St}, (s,h') \in F \right\} \cup \left\{ (\mathsf{undef}, \mathsf{undef}) \right\}$$

Any state $\mathbb{1}_\epsilon(s,h)$ is related to all states $\mathbb{1}_\epsilon(s, h \uplus h')$ such that $(s,h') \in F$, which guarantees that if $(s,h) \in P$, then $(s, h \uplus h') \in P * F$. Undefined states are only related to themselves. Now, we can express the semantics of $\circledast$ by lifting this relation.

Lemma 4.3. *If $m \vDash \varphi$ and $(m, m') \in \overline{\mathsf{frame}(F)}$, then $m' \vDash \varphi \circledast F$*

It is tempting to say that the converse should also hold, but that is not quite right. We took $\top \circledast F$ to be equal to $\top$, therefore if $m \vDash \top \circledast F$, then we cannot guarantee that all the states in m contain information about F. We therefore characterize the semantics only for the states that are not covered by $\top$, leaving the other states unconstrained.

Lemma 4.4. *If $m \vDash \varphi \circledast F$, then there exist m_1, m_1', and m_2 such that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$, $m = m_1' + m_2$ and $m_1 + m_2' \vDash \varphi$ for any m_2' such that $|m_2'| \leq |m_2|$.*

In the lemma above, m_1' represents the nontrivial portion of m and m_2 is the portion of m that is covered by $\top$. As such, m_1' must be the result of framing F into some m_1. Since m_2 is covered by $\top$, we can replace it with anything smaller than m_2—$\top$ can absorb at least $|m_2|$ worth of mass. These two lemmas provide a semantic basis to reason about what it means for $\varphi \circledast F$ to hold relative to φ.

Remark 1 (Asymmetry of the Separating Conjunction). One could imagine a symmetric definition of $\circledast$, defined semantically as follows: $m \vDash \varphi \circledast \psi$ iff there exist $m_1 \vDash \varphi$ and $m_2 \vDash \varphi$, such that m_1 and m_2 are obtained by *marginalizing* m. More precisely, $m_1(\mathbb{1}_\epsilon(s,h))$ is equal to the sum of $m(\mathbb{1}_\epsilon(s, h \uplus h'))$ over all h' such that $\mathbb{1}_\epsilon(s,h') \in \mathsf{supp}(m_2)$ (and a similar formula holds for m_2).

This gives expected properties, for example $(\mathsf{ok} : x \mapsto 1 * y \mapsto 2) \oplus (\mathsf{ok} : x \mapsto 3 * y \mapsto 4)$ implies $((\mathsf{ok} : x \mapsto 1) \oplus (\mathsf{ok} : x \mapsto 3)) \circledast ((\mathsf{ok} : y \mapsto 2) \oplus (\mathsf{ok} : y \mapsto 4))$. The problem with this approach is that it is incompatible with the frame rule. For a simple counterexample, consider the triple $\langle \mathsf{ok} : x \not\mapsto \rangle$ $[x] \leftarrow 1$ $\langle \mathsf{er} : x \not\mapsto \rangle$, using the frame rule with $(\mathsf{ok} : y \mapsto 2)$ gives us the precondition $(\mathsf{ok} : x \not\mapsto * \, y \mapsto 2)$, but the postcondition is false since there is no way to combine an er assertion with an ok one. Other problems occur when some of the program paths diverge.

So, the asymmetric definition of $\circledast$ is the one we want, as the spirit of the frame rule is to run a computation in a larger *heap*—adding pointers, not outcomes. That is not to say that the symmetric $\circledast$ is useless. In fact, it could be used in a concurrent variant of Outcome Logic to divide resources among two parallel branches in the style of Concurrent Separation Logic [O'Hearn 2004].

Replacement of Unsafe States. Undefined states arise from dereferencing pointers not in the domain of the heap. Those pointers may be in the domain of a larger heap, so previously undefined outcomes can become defined after framing in more pointers. The soundness of the frame rule depends on this not affecting the truth of the postcondition.

Whereas standard separation logic is *fault avoiding*—it requires that all states satisfying the precondition will not encounter unknown pointers—we omit this requirement in OSL in order to efficiently reason about incorrectness by only exploring a subset of the program paths. For example, in the triple $\langle \mathsf{ok} : x \not\mapsto \rangle\ \mathsf{free}(x) + C\ \langle (\mathsf{er} : x \not\mapsto) \oplus \top \rangle$, the fact that the left path leads to a memory error is enough to conclude that the program is incorrect; exploring the right path would be wasted effort. However, the right path may use other pointers not mentioned in the precondition, meaning that executing C will lead to undef. This is fine, since the assertion $\top$ covers undefined states.

However, if we use the frame rule to add information about more pointers to the precondition, the result may no longer be undef. This is still valid—*any* outcome from running C trivially satisfies $\top$. To formalize this, we use $\mathsf{Rep} \subseteq \mathsf{St} \times (\mathsf{St} \cup \{\lightning\})$, which relates undef to any state $\sigma \in \mathsf{St}$ or $\lightning$ (representing nontermination)[3]. All other states (ok and er) are related only to themselves. In addition, since $\lightning$ is not a state, we define an operation $\mathsf{prune}(m)$ to remove it. The formal definitions of both Rep and prune are given in Appendix C.2. The replacement lemma guarantees that undefined states can be replaced by anything without affecting the validity of an assertion.

Lemma 4.5 (Replacement). *If* $m \vDash \varphi$ *and* $(m, m') \in \overline{\mathsf{Rep}}$, *then* $\mathsf{prune}(m') \vDash \varphi$

Soundness. We now have all the ingredients needed to prove the soundness of the frame rule.

Theorem 4.6 (The Frame Rule). *If* $\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle$ *and* $\mathsf{mod}(C) \cap \mathsf{fv}(F) = \emptyset$, *then* $\vDash \langle \varphi \circledast F \rangle\ C\ \langle \psi \circledast F \rangle$.

We briefly sketch the proof here, the full version is in Appendix C.3. Suppose $m \vDash \varphi \circledast F$ and take any $\mathsf{alloc} \in \mathsf{Alloc}$. We now enumerate all the defined states of m and (using Lemma 4.4) we know that each state has the form $\sigma_n = \mathbb{1}_{\epsilon_n}(s_n, h_n \uplus h'_n)$ where $(s_n, h'_n) \in F$. We pick a fresh logical variable X and construct m' by modifying each $\sigma_n \in \mathsf{supp}(m)$ to be $\mathbb{1}_{\epsilon_n}(s_n[X \mapsto n], h_n)$. That is, we augment the variable store such that $X = n$ and remove the h'_n portion of the heap. Note that $m' \vDash \varphi$.

We construct a new allocator as follows: $\mathsf{alloc}'(s, h) = \mathsf{alloc}(s[X \mapsto s_{s(X)}(X)], h \uplus h'_{s(X)})$. So, $\mathsf{alloc}'(s, h)$ uses the value $s(X) = n$ to select the appropriate h'_n to add to h, guaranteeing that $[\![C]\!]_{\mathsf{alloc}'}(s_n, h_n)$ allocates the same addresses as $[\![C]\!]_{\mathsf{alloc}}(s_n, h_n \uplus h'_n)$ for all n. So, $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m)$ is related to $[\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m')$ via the two relations described previously. By the premise $\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle$, we know that $[\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m') \vDash \psi$, and by Lemmas 4.3 and 4.5, we conclude that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash \psi \circledast F$.

We have devised a frame rule for Outcome Separation Logic, which supports a rich variety of computational effects and properties about those effects. More concretely, OSL can be used to reason about both correctness and incorrectness across nondeterministic and probabilistic programs. In the next sections, we will build on this result to create compositional symbolic execution algorithms.

5 SYMBOLIC EXECUTION

With a sound frame rule, we are now ready to design symbolic execution algorithms based on OSL. The core algorithm is similar to Abductor and Infer [Calcagno et al. 2009, 2015], but adapted to better handle program choices arising from computational effects. We also show how minor modifications to this algorithm make it suitable for other use cases such as bug-finding and partial correctness. First, we introduce a restricted assertion syntax to make OSL compatible with bi-abduction, then we define tri-abduction (Section 5.1) and the main algorithm (Section 5.2).

Symbolic Heaps. In the remainder of the paper, we will work with a subset of SL assertions known as symbolic heaps. Although they have limited expressivity—particularly for pure assertions—implications of symbolic heaps are decidable [Berdine et al. 2005a], which is necessary for bi-abductive analysis algorithms. These same symbolic heaps are used by Calcagno et al. [2009, 2011].

[3]Previously undefined states may diverge after adding more pointers, *e.g.*, running $\mathsf{free}(x)\ \fatsemi$ while true do skip in an empty heap leads to undef whereas it will not terminate if x is allocated.

The syntax is shown below and the semantics is standard as defined by Berdine et al. [2005b, §2].

$$P ::= \exists \vec{X}.\Delta \quad \text{(Symbolic Heaps)} \qquad \Pi ::= \mathsf{true} \mid \Pi_1 \wedge \Pi_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \quad \text{(Pure Assertions)}$$
$$\Delta ::= \Pi \wedge \Sigma \quad \text{(Quantifier-Free)} \qquad \Sigma ::= \mathsf{true} \mid \mathsf{emp} \mid \Sigma_1 * \Sigma_2 \mid e_1 \mapsto e_2 \mid \mathsf{ls}(e_1, e_2) \quad \text{(Spatial Assertions)}$$

A symbolic heap P consists of existentially quantified logical variables, along with a pure part Π and a spatial part Σ. Pure assertions are conjunctions of equalities and inequalities, whereas spatial assertions are heap assertions joined by *separating* conjunctions. The separating conjunction requires that the heap can be split into disjoint components to satisfy the two assertions separately.

$$(s, h) \vDash \Sigma_1 * \Sigma_2 \qquad \text{iff} \qquad \exists h_1, h_2. \quad h = h_1 \uplus h_2 \quad \text{and} \quad (s, h_1) \vDash \Sigma_1 \quad \text{and} \quad (s, h_2) \vDash \Sigma_2$$

The points-to predicate $e_1 \mapsto e_2$ specifies a singleton heap in which the address e_1 points to the value e_2. Negative heap assertions $e \not\mapsto$ are syntactic sugar for $e \mapsto \bot$. These assertions were introduced in Incorrectness Separation Logic to prove that a program crashes when dereferencing invalidated memory [Raad et al. 2020]. Finally, list segments $\mathsf{ls}(e_1, e_2)$ are inductive predicates stating that a sequence of pointers starts with e_1 and ends with e_2. Formally, it is the least solution of $\mathsf{ls}(e_1, e_2) \Leftrightarrow (e_1 = e_2 \wedge \mathsf{emp}) \vee (\exists X.e_1 \mapsto X * \mathsf{ls}(X, e_2))$. We also overload $*$ and $\wedge$ as follows:

$$(\exists \vec{X}.\Pi \wedge \Sigma) * (\exists \vec{Y}.\Pi' \wedge \Sigma') \triangleq \exists \vec{X}\vec{Y}.(\Pi \wedge \Pi') \wedge (\Sigma * \Sigma') \qquad P \wedge \Pi \triangleq P * (\Pi \wedge \mathsf{emp})$$

5.1 Tri-Abduction

We now address the matter of composing paths in programs whose execution branches due to nondeterminism or random sampling. When symbolically executing such programs, we must unify the preconditions for the two branches. For example, the following program chooses to execute $[x] \leftarrow 1$ or $[y] \leftarrow 2$, so we need a precondition that mentions both x and y, and we need to know what leftover resources to add to the two resulting outcomes.

$$\begin{array}{ccccc} & \swarrow \langle \mathsf{ok} : x \mapsto X \rangle & [x] \leftarrow 1 & \langle \mathsf{ok} : x \mapsto 1 \rangle \searrow & \\ \langle \mathsf{ok} : ? \rangle & & + & & \langle (\mathsf{ok} : x \mapsto 1 * ?) \ \oplus \ (\mathsf{ok} : y \mapsto 2 * ?) \rangle \\ & \nwarrow \langle \mathsf{ok} : y \mapsto Y \rangle & [y] \leftarrow 2 & \langle \mathsf{ok} : y \mapsto 2 \rangle \nearrow & \end{array}$$

Tri-abduction provides us the power to reconcile the preconditions of the two program branches. Given P_1 and P_2, the goal is to find the *anti-frame* M and two *leftover* frames F_1 and F_2 that make $P_1 * F_1 \Dashv M \vDash P_2 * F_2$ hold. Using this, we can compose program branches as follows:

$$\frac{\langle P_1 \rangle\ C_1\ \langle Q_1 \rangle \qquad P_1 * F_1 \Dashv M \vDash P_2 * F_2 \qquad \langle P_2 \rangle\ C_2\ \langle Q_2 \rangle}{\langle M \rangle\ C_1 + C_2\ \langle (Q_1 * F_1) \oplus (Q_2 * F_2) \rangle}\ \textsc{Tri-Abductive Composition}$$

Tri-abduction would have also been useful in Abductor, which is unable to analyze the program above despite supporting nondeterminism. Abductor operates in two passes; first finding candidate preconditions for each trace, and then re-evaluating the program with each candidate in hopes that one is valid for the entire program [Calcagno et al. 2009]. Since the program above uses disjoint resources in the two branches, no candidate is valid for all traces. Using tri-abduction, we infer *more* summaries and do so in a single pass. While a single-pass bi-abduction algorithm has more recently been proposed [Sextl et al. 2023], it still cannot handle the case in the following remark.

Remark 2 (Solving Tri-Abduction using Bi-Adbuction). Our initial approach to tri-abduction was to simply use bi-abduction: given P_1 and P_2, bi-abduction can give us M and F such that $P_1 * M \vDash P_2 * F$. Using $P_1 * M$ as the anti-frame, $P_1 * M \Dashv (P_1 * M) \vDash P_2 * F$ is a tri-abduction solution.

However, this approach is inherently asymmetric, with the left branch being favored. While it would be possible to also bi-abduce in the opposite direction ($P_2 * ? \vDash P_1 * ?$) for symmetry, this still precludes valid solutions, *e.g.*, there is no bi-abduction solution for $X \mapsto Y * \mathsf{ls}(Y, Z)$ and

Base Cases

$$\frac{\Pi \land \Pi' \nvDash \mathsf{false}}{\Pi \land \mathsf{emp} \lhd [\Pi \land \Pi' \land \mathsf{emp}] \rhd \Pi' \land \mathsf{emp}}\ \textsc{Base-Emp} \qquad \frac{\Pi \land \Pi' \land \Sigma' \nvDash \mathsf{false}}{\Pi \land \mathsf{true} \lhd [\Pi \land \Pi' \land \Sigma'] \rhd \Pi' \land \Sigma'}\ \textsc{Base-True-L}$$

Quantifier Elimination

$$\frac{\Delta \lhd [M] \rhd \Delta' \qquad \vec{X} \cap (\mathsf{fv}(\Delta') \setminus \vec{Y}) = \emptyset \qquad \vec{Y} \cap (\mathsf{fv}(\Delta) \setminus \vec{X}) = \emptyset}{\exists \vec{X}.\Delta \lhd [\exists \vec{X}\vec{Y}.M] \rhd \exists \vec{Y}.\Delta'}\ \textsc{Exists}$$

Resource Matching

$$\frac{\Delta * \mathsf{ls}(e_3, e_2) \lhd [M] \rhd \Delta'}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M * e_1 \mapsto e_3] \rhd \Delta' * e_1 \mapsto e_3}\ \textsc{Ls-Start-L} \qquad \frac{\Delta \land e_2 = e_3 \lhd [M] \rhd \Delta' \land e_2 = e_3}{\Delta * e_1 \mapsto e_2 \lhd [M * e_1 \mapsto e_2] \rhd \Delta' * e_1 \mapsto e_3}\ \textsc{Match}$$

$$\frac{\Delta * \mathsf{ls}(e_3, e_2) \lhd [M] \rhd \Delta'}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M * \mathsf{ls}(e_1, e_3)] \rhd \Delta' * \mathsf{ls}(e_1, e_3)}\ \textsc{Ls-End-L}$$

Resource Adding

$$\frac{\Delta \lhd [M] \rhd \Pi' \land (\Sigma' * \mathsf{true}) \qquad \Pi' \land \Sigma' * B(e_1, e_2) \nvDash \mathsf{false}}{\Delta * B(e_1, e_2) \lhd [M * B(e_1, e_2)] \rhd \Pi' \land (\Sigma' * \mathsf{true})}\ \textsc{Missing-L} \qquad \frac{\Delta \land e_1 = e_2 \lhd [M] \rhd \Delta' \land e_1 = e_2}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M] \rhd \Delta'}\ \textsc{Emp-Ls-L}$$

Fig. 3. Selected abduction proof rules. Similarly to Calcagno et al. [2009], $B(e_1, e_2)$ represents either $\mathsf{ls}(e_1, e_2)$ or $e_1 \mapsto e_2$. Rules ending in -L have symmetric versions not shown here; see Figure 6 for the full proof system.

$\mathsf{ls}(X, Y) * Y \mapsto Z$ (in either direction), whereas tri-abduction finds the anti-frame $X \mapsto Y * Y \mapsto Z$. Tri-abduction is a fundamentally different operation that is precisely designed for branching.

Similar to [Calcagno et al. 2011, Algorithm 3], tri-abduction is done in two stages. First, we describe the *abduction* stage, in which only the anti-frame M is inferred. Next, we describe how abduction is used as a subroutine to tri-abduce all three parameters M, F_1, and F_2.

Abduction. The abductive inference step abduce-par(P, Q) is performed as a proof search—similar to Calcagno et al. [2011, Algorithm 1]—using the rules in Figure 3 to infer judgements of the form $P \lhd [M] \rhd Q$, indicating that $M \vDash P$ and $M \vDash Q$. As such, P and Q are the inputs to the algorithm and M is the output. We describe the algorithm briefly, the full definition is in Appendix D.

The inference rules are applied in the order in which they are shown, with the rules at the top being preferred over the rules lower down. The inference rules ending with -L have symmetric versions that can also be applied (the full set of rules is shown in Figure 6).

The premise of each inference rule becomes a recursive call, finding a solution to a smaller abduction query. Some of the rules have side conditions of the form of $R \nvDash \mathsf{false}$, which is checked using the proof system of Berdine et al. [2005b, §4]. Given that each recursive call describes a progressively smaller symbolic heap, the algorithm either eventually reaches a case with no explicit resources (emp or true), in which a base rule applies, or gets stuck and returns no solutions. The inference rules are described below.

Base Rules. The first step is to attempt to apply a base rule to terminate the algorithm. Base-Emp applies when both branches describe empty heaps, as long as the pure assertions in each branch do not conflict. In Base-True-L, we match against the case wherein one of the branches has an arbitrary spatial assertion Σ' and the other contains the spatial assertion true, indicating that it can absorb more resources not explicitly mentioned, so we are able to move Σ' into the anti-frame.

Quantifier Elimination. The next step is to strip existentials from the inputs P and Q and add them back to the anti-frame M obtained from the recursive call. This is achieved using the Exists rule in Figure 3. In bi-abduction, existentials are not stripped from the assertion to the right of the entailment—doing so prevents the algorithm from finding solutions in some cases. For example, $\mathsf{ls}(e, e') * ? \vDash \exists X. e \mapsto X * ?$ has a solution ($e \neq e'$), but it does not have a solution with the existential removed since nothing can be added to $\mathsf{ls}(e, e')$ to force e to point to a *specific* X. Tri-abduction produces an anti-frame M from scratch, so we are not operating under such constraints, allowing us to strip existentials at an early step in order to simplify further analysis.

It is important to note that in the Exists rule, quantified variables in one assertion cannot overlap with the free variables of the other. This ensures that no free variables in P or Q end up existentially quantified in the anti-frame M. Without the side condition, the rule is unsound; suppose we want to tri-abduce $\exists X. X = Y$ with $X = 1$, then Exists gives us the anti-frame $\exists X. X = Y \land X = 1$, which is too weak since $\exists X. X = 1 \nvDash X = 1$. In practice, our symbolic execution algorithm always generates fresh logical variables, so we will not have collisions with our usage of the Exists rule.

Resource Matching. If a base rule does not apply, then we attempt to match resources from both branches, and then call the algorithm recursively on smaller symbolic heaps with some resources moved into the returned anti-frame. Ls-Start-L applies when both branches contain the same resource e_1; however, one includes e_1 as the head of a list segment and the other refers to e_1 using a points-to predicate. Here, the points-to predicate must be the head of the list, so we move it into M and recurse on the tail of the list. The Match rule applies when both branches use e_1 in a points-to predicate, therefore the values pointed to must be equal too. Ls-End-L applies when both branches have list segments starting at the same address, so one segment must be a prefix of the other.

As in Calcagno et al. [2009], we do not consider cases where pointers are aliased. For example, if the two branches are $x \mapsto 1$ and $y \mapsto 1$, then it is possible that $x = y$. Precluding this solution helps limit the number of options we consider. Calcagno et al. [2009, Example 3] remark that this loss of precision is not detrimental in practice.

Resource Adding. Adding resources that are only present on one side is the last resort, since it involves checking a potentially expensive side condition of the form $\Pi \land \Sigma * B(e_1, e_2) \nvDash \mathsf{false}$. The Missing-L rule handles the case wherein one branch refers to resources not present in the other. This is different from the Base-True-L rule, since it handles cases where *both* branches refer to resources not explicitly present in the other. For example, Missing-L can solve $x \mapsto X * \mathsf{true} \lhd [?] \rhd y \mapsto Y * \mathsf{true}$ even though the Base-True rules do not apply. If one side of the judgement contains a list segment, but the other side does *not* contain the spatial assertion true, then there is a possible solution where the list segment is empty. Emp-Ls-L handles such cases by forcing the list segment to be empty.

As we mentioned at the beginning of the section, the tri-abduction algorithm follows a similar structure to that of bi-abduction [Calcagno et al. 2011, Algorithm 3].

$$\mathsf{triab}(P, Q) = \{(M, F_1, F_2) \mid M \in \mathsf{abduce\text{-}par}(P * \mathsf{true}, Q * \mathsf{true}),\ M \vdash P * F_1,\ M \vdash Q * F_2\}$$

We first abduce a set of anti-frames using Algorithm 1 such that $M \vDash P * \mathsf{true}$ and $M \vDash Q * \mathsf{true}$ for each M. Adding the spatial assertion true absorbs extra resources; if P and Q have different memory footprints, then there is no M such that $P \lhd [M] \rhd Q$, but adding true to both sides of the entailments allows M to refer to resources present in only one branch. Next, we use the frame inference procedure from Berdine et al. [2005b, §5] to find F_1 and F_2 such that $M \vDash P * F_1$ and $M \vDash Q * F_2$. Applying frame inference is necessary because M may mention resources present in P, but not Q (and vice versa). The set of solutions is valid according to the following correctness result, which follows from the soundness of the proof system in Berdine et al. [2005b, §5].

Theorem 5.1 (Tri-abduction). *If* $(M, F_1, F_2) \in \mathsf{triab}(P, Q)$, *then* $M \vDash P * F_1$ *and* $M \vDash Q * F_2$

$$
\begin{aligned}
&\mathsf{seq}(\ \top\ , S, \vec{x}) = \{(\mathsf{emp}, \top)\}\\
&\mathsf{seq}(\varphi_1 \bowtie \varphi_2, S, \vec{x}) =\\
&\quad \big\{(M, \psi_1' \bowtie \psi_2') \mid (M_1, \psi_1) \in \mathsf{seq}(\varphi_1, S, \vec{x})\\
&\qquad\qquad (M_2, \psi_2) \in \mathsf{seq}(\varphi_2, S, \vec{x})\\
&\qquad\qquad (M, \psi_1', \psi_2') \in \mathsf{triab}'(M_1, M_2, \psi_1, \psi_2, \vec{x})\big\}\\
&\mathsf{seq}(\ (\varphi)_a\ , S, \vec{x}) = \{(M, (\psi)_a) \mid (M, \psi) \in \mathsf{seq}(\varphi, S, \vec{x})\}\\
&\mathsf{seq}(\ \mathsf{ok} : P\ , S, \vec{x}) =\\
&\quad \big\{(M, \psi') \mid (Q, \psi) \in S, (M, \psi') \in \mathsf{biab}'(P, Q, \psi, \vec{x})\big\}\\
&\mathsf{seq}(\ \mathsf{er} : Q\ , S, \vec{x}) = \{(\mathsf{emp}, \mathsf{er} : Q)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{biab}'(\exists\vec{Z}.\Delta, Q, \psi, \vec{x}) =\\
&\quad \big\{(M', (\psi \circledast \exists\vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}])\\
&\quad \mid (M, F) \in \mathsf{biab}(\Delta, Q)\\
&\qquad (\vec{e}, \vec{Y}, M') = \mathsf{rename}(\Delta, M, Q, \{\psi\}, \vec{X}, \vec{x})\big\}\\
&\mathsf{triab}'(P_1, P_2, \psi_1, \psi_2, \vec{x}) =\\
&\quad \big\{(M', (\psi_1 \circledast \exists\vec{X}.F_1[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}],\\
&\qquad\quad (\psi_2 \circledast \exists\vec{X}.F_2[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}])\\
&\quad \mid (M, F_1, F_2) \in \mathsf{triab}(P_1, P_2)\\
&\qquad (\vec{e}, \vec{Y}, M') = \mathsf{rename}(\mathsf{emp}, M, \{\psi_1, \psi_2\}, \emptyset, \vec{x})\big\}
\end{aligned}
$$

Fig. 4. Sequencing procedure. The vector $\vec{X}$ is assumed to be fresh and the same size as $\vec{x}$, and $\bowtie \in \{\vee, \oplus_a\}$.

5.2 Symbolic Execution Algorithm

All variants of the algorithm are presented as symbolic executions, which compute an abstract semantics for a program—denoted $[\![C]\!]^\sharp\,(T)$—represented as a set of pairs of pre- and postconditions. The precondition is a single symbolic heap P, whereas the postcondition is an outcome assertion φ. The parameter T is a lookup table that gets updated with summaries for procedures as the analysis moves through the codebase. Intuitively, we think about specifications as starting in a single state and producing a collection of outcomes, as the execution may branch due to computational choices. Abductor operates in a similar fashion, but the postcondition is a disjunction rather than an outcome conjunction. The intended semantics is captured by the following soundness result.

Theorem 5.2 (Symbolic Execution Soundness). *If* $(P, \varphi) \in [\![C]\!]^\sharp\,(T)$, *then* $\vDash \langle \mathsf{ok} : P\rangle\ C\ \langle\varphi\rangle$

The strategy for the analysis is to accumulate a set of outcomes while moving forward through the program. At each step, every outcome in the current summary must be sequenced with a summary for the next command using bi-abduction. This is achieved using the seq procedure, defined in Figure 4, which takes in an outcome assertion φ, a set of summaries for the next command C, and $\vec{x}$, the variables modified by C. It computes a set of missing anti-frames M and postconditions ψ such that $\langle\varphi \circledast M\rangle\ C\ \langle\psi\rangle$ is a valid specification for C.

Sequencing after $\top$ and $(\mathsf{er} : Q)$ has no effect, since $\top$ carries no information about the current branch, and $(\mathsf{er} : Q)$ means the program has crashed. Sequencing after $(\varphi)_a$ simply sequences φ and then reapplies the weight. Sequential composition is implemented in the $(\mathsf{ok} : P)$ case, where bi-abduction is used to reconcile the current outcome with each summary for the next command. The biab′ procedure is similar to AbduceAndAdapt from Calcagno et al. [2011, Fig. 4], in which a renaming step is applied to ensure that the anti-frame M is not phrased in terms of any program variables, therefore meeting the side condition of the frame rule. The details of renaming and why it is required for correctness are explained in Appendix E.

Our algorithm differs from Abductor in the cases with multiple program branches. This is where we use tri-abduction to obtain a precondition that is guaranteed to be valid for all program paths, allowing us to analyze the program in a single pass (unlike Abductor, which must re-evaluate the program using each candidate precondition). After sequencing each of the outcomes in the precondition φ_1 and φ_2 with the next command, we use triab′ to obtain the single renamed anti-frame M that is safe for both branches. The soundness property for seq is stated below.

Lemma 5.3 (Seq). *If* $(M, \psi) \in \mathsf{seq}(\varphi, S, \vec{x})$, $\vec{x} = \mathsf{mod}(C)$, *and* $\vDash \langle\mathsf{ok} : P\rangle\ C\ \langle\vartheta\rangle$ *for all* $(P, \vartheta) \in S$, *then* $\vDash \langle\varphi \circledast M\rangle\ C\ \langle\psi\rangle$.

Symbolic Execution Algorithm. The core symbolic execution algorithm, shown in Figure 5, computes a *local* symbolic semantics which can be augmented using the frame rule to obtain summaries

$C \in \mathsf{Cmd}$	$[\![C]\!]^\sharp\,(T)$
skip	$\{(\mathsf{emp}, \mathsf{ok} : \mathsf{emp})\}$
$C_1 \fatsemi C_2$	$\{(P * M, \psi) \mid (P, \varphi) \in [\![C_1]\!]^\sharp\,(T), (M, \psi) \in \mathsf{seq}(\varphi, [\![C_2]\!]^\sharp\,(T), \mathsf{mod}(C_2))\}$
$C_1 + C_2$	$\{(M, \psi_1' \oplus \psi_2') \mid (M_1, \psi_1) \in [\![C_1]\!]^\sharp\,(T), (M_2, \psi_2) \in [\![C_2]\!]^\sharp\,(T),$ $(M, \psi_1', \psi_2') \in \mathsf{triab}'(M_1, M_2, \psi_1, \psi_2, \mathsf{mod}(C_1, C_2))\}$
$\mathsf{assume}\ b$	$\{(b \land \mathsf{emp}, \mathsf{ok} : b \land \mathsf{emp}), (\neg b \land \mathsf{emp}, (\top)_{\mathbb{0}})\}$
$\mathsf{assume}\ a$	$\{(\mathsf{emp}, (\mathsf{ok} : \mathsf{emp})_a\}$
$\mathsf{while}\ b\ \mathsf{do}\ C$	$\mathsf{lfp}\ S. \{(\neg b \land \mathsf{emp}, \mathsf{ok} : \neg b \land \mathsf{emp})\} \cup$ $\{(M_1 * M_2 \land b, \psi)$ $\mid (M_1, \varphi) \in \mathsf{seq}(\mathsf{ok} : b \land \mathsf{emp}, [\![C]\!]^\sharp\,(T), \mathsf{mod}(C)), (M_2, \psi) \in \mathsf{seq}(\varphi, S, \mathsf{mod}(C))\}$

$c \in \mathsf{Act}$	$[\![c]\!]^\sharp\,(T)$
$x := e$	$\{(x = X \land \mathsf{emp}, \mathsf{ok} : x = e[X/x] \land \mathsf{emp})\}$
$x := \mathsf{alloc}()$	$\{(x = X \land \mathsf{emp}, \mathsf{ok} : \exists Y. x \mapsto Y)\}$
$\mathsf{free}(e)$	$\{(e \mapsto X, \mathsf{ok} : e \not\mapsto), (e \not\mapsto, \mathsf{er} : e \not\mapsto), (\mathsf{emp} \land e = \mathsf{null}, \mathsf{er} : \mathsf{emp} \land e = \mathsf{null})\}$
$[e_1] \leftarrow e_2$	$\{(e_1 \mapsto X, \mathsf{ok} : e_1 \mapsto e_2), (e_1 \not\mapsto, \mathsf{er} : e_1 \not\mapsto), (\mathsf{emp} \land e_1 = \mathsf{null}, \mathsf{er} : \mathsf{emp} \land e_1 = \mathsf{null})\}$
$x \leftarrow [e]$	$\{(x = X \land e \mapsto Y, \mathsf{ok} : x = Y \land e[X/x] \mapsto Y), (e \not\mapsto, \mathsf{er} : e \not\mapsto), (\mathsf{emp} \land e = \mathsf{null}, \mathsf{er} : \mathsf{emp} \land e = \mathsf{null})\}$
$\mathsf{error}()$	$\{(\mathsf{emp}, \mathsf{er} : \mathsf{emp})\}$
$f(\vec{e})$	$\{(P \land \vec{x} = \vec{X}, \varphi) \mid (P, \varphi) \in \mathsf{seq}(\mathsf{ok} : \vec{x} = \vec{e}[\vec{X}/\vec{x}] \land \mathsf{emp}, T(f(\vec{x})), \mathsf{mod}(f))\})\}$

Fig. 5. Symbolic execution of commands and actions, all logical variables $X, Y \in \mathsf{LVar}$ are assumed to be fresh, $a \in A$ is a program weight, and $b ::= e_1 = e_2 \mid e_1 \neq e_2$ is a simple test.

in larger heaps. For example, the semantics for skip is simply the triple $\langle \mathsf{ok} : \mathsf{emp} \rangle$ skip $\langle \mathsf{ok} : \mathsf{emp} \rangle$, but this implies that running the program in any heap will yield that same heap in the end. Executing $C_1 \fatsemi C_2$ uses seq; summaries for C_1 are sequenced with all the summaries for C_2. Choices $C_1 + C_2$ are analyzed by computing summaries for each path and reconciling them with tri-abduction.

We split assume e into two cases: if e is a simple test (equality or inequality), then we generate two specifications with the precondition stating that e is true and the postcondition being unchanged, or e being false and the outcome being eliminated. This is similar to the *"assume-as-assert"* behavior of Abductor and produces precise specifications without a priori knowledge of the logical conditions that will occur. If e is a weight literal a, when we simply weight the current outcome by it. All other expressions are not supported due to limitations of the bi-abduction solver [Calcagno et al. 2009]. Similarly, while loops use a least fixed point to unroll the loop, and produce one summary for each possible number of iterations. We will see more options for analyzing loops later on.

The abstract semantics of atomic actions mostly follow the small axioms of O'Hearn et al. [2001], with failure cases inspired by Incorrectness Separation Logic [Raad et al. 2020]. Each memory operation has three specifications: one in which the pointer is allocated and the operation accordingly succeeds, and two failure cases where the pointer is not allocated or null. Procedure calls rely on pre-computed summaries in a lookup table T, which is a parameter to $[\![C]\!]^\sharp$.

Simulating Pulse. As recounted by O'Hearn [2020]; Raad et al. [2020]; Le et al. [2022], the scalability of IL-based analyses stems from their ability to *drop disjuncts.* Analyzers such as Abductor accumulate a disjunction of possible end states at each program point. For incorrectness, it is not necessary to remember *all* of these possible states; *any* of them leading to an error constitutes a bug. Since IL allows *strengthening* of postconditions, those disjuncts can be soundly dropped.

We take a slightly different view, which nonetheless enables us to drop *paths* in the same way. We differentiate between program choices that result from logical conditions (*i.e.,* if and while statements) vs computational effects (*i.e.,* nondeterministic or probabilistic choice). In the former cases, we generate multiple summaries in order to precisely keep track of which initial states will result in which outcomes. In the latter case, we use an outcome conjunction rather than a disjunction to join the outcomes. While we cannot drop outcomes per se, we can replace them by $\top$, ensuring that they will not be explored any further according to the definition of seq (Figure 4). This can be implemented by altering the definition of choice as follows.

$$[\![C_1 + C_2]\!]^\sharp(T) = \{(P, \varphi \oplus \top) \mid (P, \varphi) \in [\![C_1]\!]^\sharp(T)\} \cup \{(P, \varphi \oplus \top) \mid (P, \varphi) \in [\![C_2]\!]^\sharp(T)\}$$

Given this modification, the algorithm remains sound with respect to the same semantics (*i.e.,* Theorem 5.2), but it no longer fits with the spirit of correctness reasoning, since some of the program outcomes are left unspecified. We will see in Section 6.1 how it can be used for efficient bug-finding.

Now, we have a situation where each element of $[\![C]\!]^\sharp(T)$ stands alone as a sound summary for the program C, and represents a particular trace. This corresponds to taking a particular logical branch in if statements, unfolding while loops for a particular number of iterations, and particular (nondeterministic or probabilistic) choices. So, eliminating elements from the set will only preclude possible summaries without affecting the correctness of the existing ones, meaning that we can drop paths by, *e.g.,* unrolling loops for a certain number of iterations and only taking one path when encountering a +. Le et al. [2022] refer to this as depth and width of the analysis, respectively.

Loop invariants and partial correctness. An alternative to bounded unrolling for (non)deterministic programs is to use loop invariants. We can alter the rule for while loops to the following.

$$[\![\text{while } e \text{ do } C]\!]^\sharp(T) = \{(I, (\mathsf{ok} : I \land \neg e) \lor (\top)_{\mathbb{0}}) \mid (I \land e, \mathsf{ok} : I) \in [\![C]\!]^\sharp(T)\}$$

The truth of the invariant I is preserved by the loop body, therefore it must remain true *if* the loop exits. The possibility of nontermination is expressed by the disjunction with $(\top)_{\mathbb{0}}$.

Finding loop invariants is generally undecidable, however techniques from abstract interpretation [Cousot and Cousot 1977] can be used to find invariants by framing the problem as a fixed point computation over a finite domain, thereby guaranteeing convergence. This is the approach taken in Abductor [Calcagno et al. 2011], which uses the same symbolic heaps, but without outcome conjunctions. In nondeterministic programs, we can convert outcome conjunctions into disjunctions since $(\mathsf{ok} : P) \oplus (\mathsf{ok} : Q) \Rightarrow (\mathsf{ok} : P \lor Q)$ and therefore we can use the same technique.

Nondeterministic Allocation. Memory bugs can arise in C from failing to check whether the address returned by malloc is non-null. This is often modeled using nondeterminism, wherein the semantics of malloc returns either a valid pointer or null, nondeterministically. Our language is generic over effects, so we do not have a nondeterministic malloc operation, but we can add $x := \mathsf{malloc}()$ as syntactic sugar for $(x := \mathsf{alloc}()) + (x := \mathsf{null})$, and derive the following semantics:

$$[\![x := \mathsf{malloc}()]\!]^\sharp(T) = \{(x = X \land \mathsf{emp}, (\mathsf{ok} : x = \mathsf{null} \land \mathsf{emp}) \oplus (\mathsf{ok} : \exists Y. x \mapsto Y))\}$$

Reusing Summaries. Though partial correctness specifications are incompatible with bug-finding, and under-approximate specifications are incompatible with verification, there is still overlap in summaries that can be used for both. Many procedures in a given codebase do not include loops or branching, so their summaries are equally valid for both correctness and incorrectness, and also in programs with different interpretations of choice. In other cases, when a procedure does have multiple outcomes, it is easy to convert a correctness specification into several individual incorrectness ones, since the following implication is sound. We will see this in action in Section 6.1.

$$\langle \mathsf{ok} : P \rangle\, C\, \langle \psi_1 \oplus \psi_2 \rangle \qquad \Longrightarrow \qquad \langle \mathsf{ok} : P \rangle\, C\, \langle \psi_1 \oplus \top \rangle \quad \text{and} \quad \langle \mathsf{ok} : P \rangle\, C\, \langle \psi_2 \oplus \top \rangle$$

6 CASE STUDIES

We will now demonstrate how the symbolic execution algorithms work by examining two case studies, which show the applicability in both nondeterministic and probabilistic execution models.

6.1 Nondeterministic Vector Reallocation

Our first case study involves a common error in C++ when using the `std::vector` library in which a call to push_back may reallocate the vector's underlying memory buffer, invalidating any pointers to that cell that existed before the call. This was also used as a motivating example for Incorrectness Separation Logic [Raad et al. 2020]. Following their lead, we model the vector as a single pointer and we treat reallocation as nondeterministic. The program is shown below.

$$
\begin{array}{l}
\mathsf{main}() : \\
\quad x \leftarrow [v] ⨟ \\
\quad \mathsf{push\_back}(v) ⨟ \\
\quad [x] \leftarrow 1
\end{array}
\qquad
\begin{array}{l}
\mathsf{push\_back}(v) : \\
\left(\begin{array}{l} y \leftarrow [v] ⨟ \\ \mathsf{free}(y) ⨟ \\ y := \mathsf{alloc}() ⨟ \\ [v] \leftarrow y \end{array}\right) \quad + \quad \mathsf{skip}
\end{array}
$$

Before we analyze the main procedure, we must store $[\![\mathsf{push\_back}(v)]\!]^\sharp(T)$ in the procedure table. Since push_back is a common library function, it makes sense to compute summaries describing all the outcomes, which will be reusable for both correctness and incorrectness analyses. The first step is to analyze the two nondeterministic branches, which are both simple sequential programs.

$$
\left[\!\!\left[\begin{array}{l} y \leftarrow [v] ⨟ \\ \mathsf{free}(y) ⨟ \\ y := \mathsf{alloc}() ⨟ \\ [v] \leftarrow y \end{array}\right]\!\!\right]^\sharp (T) = \left\{\begin{array}{ll} (v \mapsto A * A \mapsto -, & \mathsf{ok} : \exists B. v \mapsto B * B \mapsto - * A \not\mapsto) \\ (v \mapsto A * A \not\mapsto, & \mathsf{er} : v \mapsto A * A \not\mapsto \quad) \\ (v \not\mapsto, & \mathsf{er} : v \not\mapsto \quad) \\ & \cdots \end{array}\right\}
\qquad [\![\mathsf{skip}]\!]^\sharp(T) = \{(\mathsf{emp}, \mathsf{ok} : \mathsf{emp})\}
$$

Now, we can compose the two program branches using tri-abduction. Choosing the first summary for the first branch, we get the following tri-abduction solution.

$$
v \mapsto A * A \mapsto - * [\mathsf{emp}] \dashv [v \mapsto A * A \mapsto -] \vDash \mathsf{emp} * [v \mapsto A * A \mapsto -]
$$

So, by framing emp into the first branch and $v \mapsto A * A \mapsto -$ into the second branch, we get a summary for push_back as a whole. This can similarly be done for the other summaries of the first branch, yielding the lookup table below.

$$
T = \left\{\begin{array}{lll}
\langle \mathsf{ok} : v \mapsto A * A \mapsto - \rangle & \mathsf{push\_back}(v) & \langle (\mathsf{ok} : \exists B. v \mapsto B * B \mapsto - * A \not\mapsto) \oplus (\mathsf{ok} : v \mapsto A * A \mapsto -) \rangle \\
\langle \mathsf{ok} : v \mapsto A * A \not\mapsto \rangle & \mathsf{push\_back}(v) & \langle (\mathsf{er} : v \mapsto A * A \not\mapsto) \oplus (\mathsf{ok} : v \mapsto A * A \not\mapsto) \rangle \\
\langle \mathsf{ok} : v \not\mapsto \rangle & \mathsf{push\_back}(v) & \langle (\mathsf{er} : v \not\mapsto) \oplus (\mathsf{ok} : v \not\mapsto) \rangle \\
& \cdots &
\end{array}\right\}
$$

The first summary tells us that push_back may reallocate the underlying buffer, in which case the original pointer A will become deallocated. The next two summaries describe ways in which push_back itself can fail. We will focus on using the first summary to show how main will fail if the buffer gets reallocated. We analyze main in an under-approximate fashion in order to look for bugs. The first step is to compute summaries for the first two commands of main. The load on the first line has three summaries according to Figure 5, we select the first one in which v is allocated.

$$
\langle \mathsf{ok} : x = X \land v \mapsto Y \rangle \; x \leftarrow [v] \; \langle \mathsf{ok} : x = Y \land v \mapsto Y \rangle \in [\![x \leftarrow [v]]\!]^\sharp (T)
$$

The procedure call on the second line requires us to look up summaries in T. We select the first one, but we will use an under-approximate version of it so as to explore only one of the paths

$$
\langle \mathsf{ok} : v \mapsto A * A \mapsto - \rangle \; \mathsf{push\_back}(v) \; \langle (\mathsf{ok} : \exists B. v \mapsto B * B \mapsto - * A \not\mapsto) \oplus \top \rangle
$$

Now, we use seq to sequentially compose these summaries, which involves bi-abducing the post-condition of $x \leftarrow [v]$ with the precondition of push_back(v).

$$
x = Y \land v \mapsto Y * [A = Y * x \mapsto -] \vDash v \mapsto A * A \mapsto - * [\mathsf{emp}]
$$

So, after renaming, we get the following summary for the composed program:

$$
\langle \mathsf{ok} : v \mapsto x * x \mapsto - \rangle \; x \leftarrow [v] ⨟ \mathsf{push\_back}(v) \; \langle (\mathsf{ok} : \exists B. v \mapsto B * B \mapsto - * x \not\mapsto) \oplus \top \rangle
$$

Now, observe that the postcondition above is only compatible with one of the summaries in Figure 5 for the last line of the program. Since x is deallocated in the only specified outcome, the write into x must fail. Using bi-abduction again, we can construct the following description of the error.

$$\langle \mathsf{ok} : v \mapsto x * x \mapsto - \rangle\; x \leftarrow [v] \fatsemi \mathsf{push\_back}(v) \fatsemi [x] \leftarrow 1\; \langle (\mathsf{er} : \exists B. v \mapsto B * B \mapsto - * x \not\mapsto) \oplus \top \rangle$$

6.2 Consensus in Distributed Computing

In this case study, we use OSL to lower bound reliability rates of a distributed system. In the basic consensus algorithm, shown below, each of three processes broadcasts a value v_i by storing it in a pointer p_i, and consensus is reached if any two of these processes broadcasted the same value. To model unreliability of the network, the broadcast procedure fails with probability 1%. We would like to know how likely we are to reach consensus, given that two of the processes agree.

```
main() :
  p1 := alloc() ⨾ broadcast(v1, p1)⨾
  p2 := alloc() ⨾ broadcast(v2, p2)⨾
  p3 := alloc() ⨾ broadcast(v3, p3)⨾
  v := alloc()⨾
  decide(p1, p2, p3, v)

broadcast(v, p) :
  ([p] ← v) ⊕0.99 error()
```

```
decide(p1, p2, p3, v) :
  x1 ← [p1] ⨾ x2 ← [p2] ⨾ x3 ← [p3]⨾
  if x1 = x2 then
    [v] ← x1
  else if x1 = x3 then
    [v] ← x1
  else if x2 = x3 then
    [v] ← x2
  else skip
```

We begin by examining the summary table. The broadcast procedure has two outcomes corresponding to whether or not the communication went through. Though there are many summaries for decide, we show only the one in which the values sent on p_1 and p_2 are equal.

$$T = \left\{ \begin{array}{c} \langle \mathsf{ok} : p \mapsto - \land v = V \rangle\; \mathsf{broadcast}(v, p)\; \langle (\mathsf{ok} : p \mapsto V \land v = V) \oplus_{0.99} (\mathsf{er} : p \mapsto - \land v = V) \rangle \\ \langle \mathsf{ok} : p_1 \mapsto V_1 * p_2 \mapsto V_2 * p_3 \mapsto V_3 * v \mapsto - \land V_1 = V_2 \rangle\; \mathsf{decide}(p_1, p_2, p_3, v)\; \langle \mathsf{ok} : v \mapsto V_1 * \cdots \rangle \\ \cdots \end{array} \right\}$$

We again use the single-path algorithm to analyze main, but this time we are interested only in the successfully terminating cases. We get the following summaries for each of the first three lines.

$$\langle \mathsf{ok} : v_i = V_i \rangle\; p_i := \mathsf{alloc}() \fatsemi \mathsf{broadcast}(v_i, p_i)\; \langle (\mathsf{ok} : v_i = V_i \land p_i \mapsto V_i) \oplus_{0.99} \top \rangle$$

These three summaries can be combined—along with the simplification that $(\varphi \oplus_a \top) \oplus_b \top$ implies $\varphi \oplus_{a \cdot b} \top$—to obtain the following assertion just before the call to decide

$$(\mathsf{ok} : v_1 = V_1 \land v_2 = V_2 \land v_3 = V_3 \land p_1 \mapsto V_1 * p_2 \mapsto V_2 * p_3 \mapsto V_3) \oplus_{0.99^3} \top$$

Now, we can bi-abduce the first outcome above with the precondition for decide shown in T. This sends $V_1 = V_2$ backwards into the precondition, and we get an overall summary for main telling us that if $v_1 = v_2$, then the protocol will reach consensus ($v \mapsto V_1$) with probability *at least* 97%.

$$\langle \mathsf{ok} : v_1 = V_1 \land v_2 = V_2 \land v_3 = V_3 \land V_1 = V_2 \rangle\; \mathsf{main}()\; \langle (\mathsf{ok} : v \mapsto V_1 * \cdots) \oplus_{0.9703} \top \rangle$$

7 RELATED WORK

Separation Logic and the Frame Rule. While many variants of separation logic exist, OSL is the only one that supports alternative types of choice as well as both may and must properties. The soundness the frame rule in both partial and total correctness separation logic relies on nondeterminism and *must* properties [Yang and O'Hearn 2002; Calcagno et al. 2007], so they are suitable only for correctness in nondeterministic languages. Some work has been done to drop the nondeterminism requirement, for example Baktiev [2006] proved that the frame rule is sound in a deterministic language if heap assertions are unaffected by address permutation, however this requires languages without address arithmetic. Similarly, Tatsuta et al. [2009] created a deterministic separation logic, but the frame rule only applies to programs that do not allocate memory.

Incorrectness Separation Logic (ISL) has a frame rule that is compatible with may properties, but not must properties [Raad et al. 2020]. Unlike OSL, ISL has no restrictions on assertions about memory allocation; it is possible to prove that a particular address ℓ is returned. If, after applying the frame rule, ℓ is already in use, then the postcondition of the ISL triple becomes false, making the triple vacuous (whereas in regular separation logic and OSL, a false *pre*condition makes triples vacuous). However, whereas ISL has slightly more power to express may properties, ISL cannot express must (*i.e.,* correctness) properties and is also specialized only to nondeterminism.

Exact Separation Logic (ESL) [Maksimović et al. 2023] combines the semantics of SL and ISL to create a logic that is suitable for both correctness and incorrectness, while inheriting the limitations of both. That is, ESL can only express properties that are *both* may and must properties, meaning that it cannot under-approximate by dropping paths or over-approximate using loop invariants.

Higher-order separation logics [Birkedal and Yang 2007; Birkedal et al. 2008] including Iris [Jung et al. 2015] bake the frame rule into the definition of triples themselves, making the soundness of the frame rule trivial without any additional assumptions. More precisely, triples $\{P\}\ C\ \{Q\}$ are valid iff for any frame F, if $\sigma \vDash P * F$, then $\tau \vDash Q * F$ for all $\tau \in [\![C]\!](\sigma)$. As a tradeoff, frame baking introduces additional proof obligations whenever constructing a triple (*i.e.,* it must be shown that every inference rule is *frame preserving*). We could have used this approach in OSL, but there is no free lunch; it would have involved moving the inductive cases of Lemma C.7 into Theorem 5.2.

We chose not to do this, as we preferred to keep the proof of the frame rule self contained. The choice is mostly orthogonal to the power of the logic, and somewhat philosophical: we proved that programs are local *actions* whereas the frame baking approach shows that you can only *express* local properties about commands. The exception to this is memory allocation, which is not a local action, so we baked the expressivity limitation into the triple semantics. We felt this was a good tradeoff, since we baked in a weaker property, which was just strong enough to complete the proof.

Probabilistic Separation Logics. While some work has been done to incorporate probabilistic reasoning into separation logic, these logics differ from OSL in the scope and applicability of the frame rule, and have not been shown to be compatible with bi-abduction. Quantitative Separation Logic [Batz et al. 2019] and its concurrent counterpart [Fesefeldt et al. 2022] use weakest pre-expectation [Morgan et al. 1996] style predicate transformers to derive expected values in probabilistic pointer programs. They rely on demonic nondeterminism for allocation—the expected value is lower bounded over all possible allocated addresses—and the frame rule gives a lower bound, whereas OSL is used for propositional reasoning, with each outcome having a likelihood.

Polaris [Tassarotti and Harper 2019] incorporates probabilistic reasoning into Iris [Jung et al. 2015] and is also used to bound expected values using refinements inspired by probabilistic relational Hoare Logic [Barthe et al. 2015]. Polaris is limited to programs that terminate in finitely many steps and the program logic itself is only used to relate probabilistic programs to each other, whereas the quantitative reasoning about expected values must be done externally to the program logic.

Probabilistic Separation Logic (PSL) [Barthe et al. 2019] and subsequent works [Bao et al. 2021, 2022; Li et al. 2023] use an alternative model of separation to characterize probabilistic independence and related probability theoretic properties. Doing so provides a compositional way to reason about probabilistic programs, though this work is orthogonal to our own as it does not deal with heaps.

Pulse and Incorrectness Separation Logic. As recounted by Raad et al. [2020, §5], Pulse uses under-approximation in four ways in order to achieve scalability:

(1) Pulse takes advantage of the IL semantics in order to explore only one path at a time when the program execution branches, and to unroll loops for a bounded number of iterations.

(2) Pulse elects to not consider cases in which memory is re-allocated.
(3) Pulse uses under-approximate specifications for some library functions.
(4) Pulse's bi-abductive inference assumes that pointers are not aliased unless explicitly stated.

We have shown how (1) is achieved using OSL in the single path algorithm, (2) and (4) are standard assumptions in bi-abduction [Calcagno et al. 2009, 2011] (which we also use), and (3) is a corollary to (1), since the ability to drop paths opens the possibility for under-approximate procedure summaries.

Pulse does not support inductive predicates (*e.g.,* list segments), so it uses a simplified bi-abduction procedure capable of handling more types of pure assertions. This results in *exact* bi-abduction solutions; the inferred M and F satisfy $P * M \dashv\vDash Q * F$. As a result, Pulse does not use consequences to—since it is based on IL—strengthen the postcondition, meaning that the resulting specs can be interpreted as both ISL and OSL triples, and suggesting that our algorithm accurately models Pulse.

Unified Metatheory with Effects. Our algebraic program semantics is similar to Weighted Programming [Batz et al. 2022]. Whereas we use an algebraic interpretation of choice to represent multiple types of (executable) program semantics, the goal of weighted programming is to specify mathematical models and find solutions to optimization problems via static analysis. Cîrstea [2013, 2014] also used partial semirings to represent properties of program branching in coalgebraic logics.

Delaware et al. [2013a,b] developed 3MT, a unified framework in the Coq proof assistant for mechanizing metatheoretic proofs—such as type soundness–about languages with monadic effects. Our motivations are similar, but with the goal of developing program logics.

Unifying correctness and incorrectness. Exact Separation Logic (ESL) combines the semantics of SL and ISL to support correctness and incorrectness within a single program logic [Maksimović et al. 2023]. ESL is implemented in the Gillian static analyzer [Fragoso Santos et al. 2020] and demonstrates that inductive predicates are compatible with the IL semantics, going beyond Pulse or Pulse-X. Similarly, Local Completeness Logic (LCL) is based on the semantics of IL, but uses an over-approximate abstract domain to ensure that the under-approximation is never too far away from the strongest post so as to preclude recovering a correctness spec too [Bruni et al. 2021, 2023].

Though the goals of these two developments are similar to our own, we take a different approach; in keeping with the tradition of O'Hearn [2020]; Raad et al. [2020]; Le et al. [2022], we opt to design *separate* algorithms for correctness and incorrectness, recognizing that fundamental properties of bug-finding allow us to trade off a complete view of all program outcomes for increased efficiency. Still, we are able to provide a unification of the metatheory and share summaries for some procedures. Crucially, OSL permits dropping paths just like IL, which is not possible in either ESL or LCL. Hyper Hoare Logic takes a similar approach to OL, in which both correctness and incorrectness hyper-properties can be proven for a nondeterministic language [Dardinier and Müller 2023].

8 CONCLUSION

Infer—based on separation logic and bi-abduction—is capable of analyzing industrial scale codebases, substantiating the idea that compositionality translates to real-world scalability [Calcagno et al. 2015]. But the deployment of Infer also surfaced that proving the *absence* of bugs is somewhat of a red herring—software has bugs and sound logical theories are needed to find them [Le et al. 2022].

Incorrectness Logic has shown that it is not only *possible* to formulate a theory for bug-finding, but it is in fact advantageous from a program analysis view; static analyzers can take certain liberties in searching for bugs that are not valid for correctness verification, such as dropping program paths for added efficiency. The downside is that the IL semantics is incompatible with correctness analysis, therefore separate implementations and procedure summaries must be used.

In OSL, we seek to get the best of both worlds. As Raad et al. [2020, §6] put it, "aiming for under-approximate results rather than exact ones gives additional flexibility to the analysis designer, just as aiming for over-approximate rather than exact results does for correctness tools." The fact that OSL supports over-approximation in the traditional sense as well as under-approximation in the sense of Pulse invites the reuse of tools between the two, while still enabling specialized techniques when needed (*i.e.,* loop invariants for correctness, dropping paths for incorrectness). In addition, OSL extends bi-abduction to programs with effects such as randomization for the first time.

OSL is not a simple extension of separation logic; it is designed from the ground up with new assumptions, since the properties that make the standard SL frame rule sound (nondeterministic allocation, must properties, and fault avoidance) are not suitable for reasoning about incorrectness and effects. The addition of tri-abduction to our symbolic execution algorithms also means that we can analyze more programs with control flow branching compared to Abductor. The power and flexibility of OSL makes it a strong foundation for analyzing pointer-programs with effects.

REFERENCES

Murat Baktiev. 2006. *Permutation Semantics of Separation Logic.* Master's thesis. Saarland University. https://www.ps.uni-saarland.de/Publications/documents/baktiev2006.pdf

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A Bunched Logic for Conditional Independence. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 13, 14 pages. https://doi.org/10.1109/LICS52264.2021.9470712

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A Separation Logic for Negative Dependence. *Proc. ACM Program. Lang.* 6, POPL, Article 57 (jan 2022), 29 pages. https://doi.org/10.1145/3498719

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 387–401. https://doi.org/10.1007/978-3-662-48899-7_27

Gilles Barthe, Justin Hsu, and Kevin Liao. 2019. A Probabilistic Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 55 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371123

Kevin Batz, Adrian Gallus, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Tobias Winkler. 2022. Weighted Programming: A Programming Paradigm for Specifying Mathematical Models. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 66 (apr 2022), 30 pages. https://doi.org/10.1145/3527310

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 34 (Jan 2019), 29 pages. https://doi.org/10.1145/3290347

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. A Decidable Fragment of Separation Logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, Kamal Lodaya and Meena Mahajan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005b. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–68. https://doi.org/10.1007/11575467_5

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. 2008. A Simple Model of Separation Logic for Higher-Order Store. In *Automata, Languages and Programming*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–360. https://doi.org/10.1007/978-3-540-70583-3_29

Lars Birkedal and Hongseok Yang. 2007. Relational Parametricity and Separation Logic. In *Foundations of Software Science and Computational Structures*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–107. https://doi.org/10.1007/978-3-540-71389-0_8

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS52264.2021.9470608

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* 70, 2, Article 15 (mar 2023), 45 pages. https://doi.org/10.1145/3582267

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/1480881.1480917

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 366–378. https://doi.org/10.1109/LICS.2007.30

Corina Cîrstea. 2013. From Branching to Linear Time, Coalgebraically. In *Proceedings Workshop on Fixed Points in Computer Science, FICS 2013, Turino, Italy, September 1st, 2013 (EPTCS, Vol. 126)*, David Baelde and Arnaud Carayol (Eds.). 11–27. https://doi.org/10.4204/EPTCS.126.2

Corina Cîrstea. 2014. A Coalgebraic Approach to Linear-Time Logics. In *Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 426–440. https://doi.org/10.1007/978-3-642-54830-7_28

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Thibault Dardinier and Peter Müller. 2023. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (extended version). https://doi.org/10.48550/ARXIV.2301.10037

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013a. Meta-Theory à La Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/2429069.2429094

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. 2013b. Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 319–330. https://doi.org/10.1145/2500365.2500587

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug 1975), 453–457. https://doi.org/10.1145/360933.360975

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. 2022. Towards Concurrent Quantitative Separation Logic. In *33rd International Conference on Concurrency Theory (CONCUR 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 243)*, Bartek Klin, Sławomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:24. https://doi.org/10.4230/LIPIcs.CONCUR.2022.25

José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. https://doi.org/10.1145/3385412.3386014

Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–85. https://doi.org/10.1007/BFb0092872

Jonathan S. Golan. 2003. *Semirings and Affine Equations over Them.* Springer Dordrecht. https://doi.org/10.1007/978-94-017-0383-3

Peter H. Gumm. 2009. Copower functors. *Theoretical Computer Science* 410, 12 (2009), 1129–1142. https://doi.org/10.1016/j.tcs.2008.09.057

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Georg Karner. 2004. Continuous monoids and semirings. *Theoretical Computer Science* 318, 3 (2004), 355–372. https://doi.org/10.1016/j.tcs.2004.01.020

Bartek Klin. 2009. *Structural Operational Semantics for Weighted Transition Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–139. https://doi.org/10.1007/978-3-642-04164-8_7

Alexander Kurz and Jiří Velebil. 2016. Relation lifting, a survey. *Journal of Logical and Algebraic Methods in Programming* 85, 4 (2016), 475–499. https://doi.org/10.1016/j.jlamp.2015.08.002 Relational and algebraic methods in computer science.

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (Apr 2022), 27 pages. https://doi.org/10.1145/3527325

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: a Modal Separation Logic for Conditional Probability. arXiv:2304.01339 [cs.PL]

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:27. https://doi.org/10.4230/LIPIcs.ECOOP.2023.19

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.* 18, 3 (may 1996), 325–353. https://doi.org/10.1145/229542.229547

Lawrence S. Moss. 1999. Coalgebraic logic. *Annals of Pure and Applied Logic* 96, 1 (1999), 277–317. https://doi.org/10.1016/S0168-0072(98)00042-6

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn. 2020. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Jan. 2020), 32 pages. https://doi.org/10.1145/3371078

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, Berlin, Heidelberg, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent Incorrectness Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 34 (Jan 2022), 29 pages. https://doi.org/10.1145/3498695

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Florian Sextl, Adam Rogalewicz, Tomáš Vojnar, and Florian Zuleger. 2023. Sound One-Phase Shape Analysis with Biabduction. arXiv:2307.06346 [cs.LO]

Joseph Tassarotti and Robert Harper. 2019. A Separation Logic for Concurrent Randomized Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 64 (Jan 2019), 30 pages. https://doi.org/10.1145/3290377

Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. 2009. Completeness of Pointer Program Verification by Separation Logic. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. 179–188. https://doi.org/10.1109/SEFM.2009.33

Hongseok Yang and Peter O'Hearn. 2002. A Semantic Basis for Local Reasoning. In *Foundations of Software Science and Computation Structures*, Mogens Nielsen and Uffe Engberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 402–416. https://doi.org/10.1007/3-540-45931-6_28

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (Apr 2023), 29 pages. https://doi.org/10.1145/3586045

A PROGRAM SEMANTICS

We begin by providing definitions for natural orders, completeness and Scott continuity, which are mentioned in Definition 3.3.

Definition A.1 (Natural Ordering). The *natural order* of a semiring $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$ is defined to be $a \leq b$ iff $\exists a' \in A. a + a' = b$. A semiring is *naturally ordered* if $\leq$ is a partial order. Note that $\leq$ is reflexive and transitive by the semiring laws, so it remains only to show that it is anti-symmetric. That is, if $a \leq b$ and $b \leq a$, then $a = b$.

Definition A.2 (Complete Partial Semiring). A partial semiring $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$ is complete if there is a sum operator $\sum_{i\in I} x_i$ such that the following properties hold:

(1) If $I = \{i_1, \ldots, i_n\}$ is finite, then $\sum_{i\in I} x_i = x_{i_1} + \cdots + x_{i_n}$
(2) If $\sum_{i\in I} x_i$ is defined, then $b \cdot \sum_{i\in I} x_i = \sum_{i\in I} b \cdot x_i$ and $(\sum_{i\in I} x_i) \cdot b = \sum_{i\in I} x_i \cdot b$ for any $b \in A$
(3) Let $(J_k)_{k\in K}$ be any family of nonempty disjoint subsets of I, so $I = \bigcup_{k\in K} J_k$ and $J_k \cap J_\ell = \emptyset$ if $k \neq \ell$. Then, $\sum_{k\in K} \sum_{j\in J_k} x_j = \sum_{i\in I} x_i$.

This definition is adapted from Golan [2003, Chapter 3].

Definition A.3 (Scott Continuity). Consider a semiring $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$ with partial order $\leq$. A function (or partial function) $f : A \to A$ is Scott continuous if for any directed set $D \subseteq A$ (where all pairs of elements in D have a supremum), $\sup_{a\in D} f(a) = f(\sup D)$. A semiring is Scott continuous if $\sum$ and $\cdot$ are Scott continuous in both arguments [Karner 2004].

In addition, we recall the definition of normalizable, which is stated as property (3) in Definition 3.3.

Definition A.4 (Normalizable). A semiring $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$ is normalizable if for any well defined sum $\sum_{i\in I} a_i$, there exists $(b_i)_{i\in I}$ such that $\sum_{i\in I} b_i = \mathbb{1}$ and $a_i = (\sum_{i\in I} a_i) \cdot b_i$ for every $i \in I$.

Normalizability is needed to show that relations lifted by the $\mathcal{W}_\mathcal{A}$ functor have several properties, for example that lifting is well behaved with respect to sums and scalar multiplication (Lemmas B.4 and B.5). We also show that normalization implies the weaker row-column property below:

Definition A.5 (Row-Column Property). A monoid $\langle A, +, \mathbb{0}\rangle$ has the row-column property if for any two sequences of elements $(a_i)_{i\in I}$ and $(b_j)_{j\in J}$, if $\sum_{i\in I} a_i = \sum_{j\in J} b_j$, then there exist $(u_k)_{k\in I\times J}$ such that $\sum_{j\in J} u_{(i,j)} = a_i$ for all $i \in I$ and $\sum_{i\in I} u_{(i,j)} = b_j$ for all $j \in J$.

Lemma A.6. *Let $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$ be a normalizable semiring (Definition A.4), then $\langle A, +, \mathbb{0}\rangle$ has the row-column property (Definition A.5).*

Proof. Take any sequences $(a_i)_{i\in I}$ and $(b_j)_{j\in J}$ such that $\sum_{i\in I} a_i = \sum_{j\in J} b_j$ and let $x = \sum_{i\in I} a_i = \sum_{j\in J} b_j$. Now, since the semiring is normalizable, there must be $(a'_i)_{i\in I}$ and $(b'_j)_{j\in J}$ such that $\sum_{i\in I} a'_i = \sum_{j\in J} b'_j = \mathbb{1}$ and $a_i = x \cdot a'_i$ for all $i \in I$ and $b_j = x \cdot b'_j$ for all $j \in J$. Let $u_{(i,j)} = x \cdot a'_i \cdot b'_j$. Now we have:

$$\sum_{i\in I} u_{(i,j)} = \sum_{i\in I} x \cdot a'_i \cdot b'_j = x \cdot (\sum_{i\in I} a'_i) \cdot b'_j = x \cdot \mathbb{1} \cdot b'_j = x \cdot b'_j = b_j$$

And

$$\sum_{j\in J} u_{(i,j)} = \sum_{j\in J} x \cdot a'_i \cdot b'_j = x \cdot a'_i \cdot (\sum_{j\in J} b'_j) = x \cdot a'_i \cdot \mathbb{1} = x \cdot a'_i = a_i$$

□

In fact, it is known that if some monoid A has the row-column property, then the functor $\mathcal{F}_A$ of finitely supported maps into A preserves weak pullbacks [Gumm 2009; Moss 1999; Klin 2009]. It has also been shown that relations lifted by some functor preserve composition iff the functor preserves weak pullbacks [Kurz and Velebil 2016]. Combining these two results, we get that relations lifted by $\mathcal{F}_A$ preserve composition if A has the row-column property.

In our case, the maps have countable support rather than finite, so the aforementioned results do not immediately apply, however they do provide evidence that the row-column property is a reasonable requirement. However, we require the stronger normalization property since lifted relations also must also be well behaved with respect to scalar multiplication (Lemma B.5).

Remark 3. In Definition 3.3 we require that $\sup(A) = \mathbb{1}$, which limits the models to be contractive maps (the weight of the computation can only decrease as the program executes). This rules out, for example, real-valued multisets where the weights are elements of the semiring $\langle \mathbb{R}^\infty, +, \cdot, 0, 1\rangle$. Still, there are more models than the ones we have presented including the tropical semiring $\langle \mathbb{R}_0^\infty, \min, +, \infty, 0\rangle$, which can be used to encode optimization problems [Batz et al. 2022].

The fact that $\sup(A) = \mathbb{1}$ is used in order to define Rep as a lifted relation (Appendix C.2), and it is also used in the proof of Lemma C.5. It may be possible to relax this constraint, however the more general version is not needed for the models we explore in this paper.

A.1 Proofs

Lemma A.7 (Scaled Sums). *If $\sum_{i\in I} x_i$ is defined, then $\sum_{i \in I} x_i \cdot y_i$ is defined for any $(y_i)_{i\in I}$.*

Proof. Since $\sup(A)$ exists, then $y_i \le \sup(A)$ for each $i \in I$. By the definition of $\le$, there exist $(y_i')_{i\in I}$ such that $y_i + y_i' = \sup(A)$. We also know that $(\sum_{i\in I} x_i) \cdot \sup(A)$ exists, since A is closed under multiplication. Now, we have:

$$(\sum_{i\in I} x_i) \cdot \sup(A) = \sum_{i\in I} x_i \cdot (y_i + y_i')$$

And by the semiring laws:

$$\begin{aligned} &= \sum_{i\in I} x_i \cdot y_i + x_i \cdot y_i' \\ &= \sum_{i\in I} x_i \cdot y_i + \sum_{i\in I} x_i \cdot y_i' \end{aligned}$$

So, clearly the subexpression $\sum_{i\in I} x_i \cdot y_i$ is defined. □

Lemma A.8 (Totality of Bind). *The* bind *function defined in Definition 3.7 is a total function (this is not immediate, since it uses partial addition).*

Proof. First, we note that $\sum_{a\in\mathsf{supp}(m)} m(a)$ must be defined by the definition of $\mathcal{W}$. By Lemma A.7, we know that $\sum_{a\in\mathsf{supp}(m)} m(a) \cdot f(a)(b)$ must be defined too, for any family $(f(a)(b))_{a\in\mathsf{supp}(m)}$. Now, since $\mathsf{bind}(m, f)(b) = \sum_{a\in\mathsf{supp}(m)} m(a) \cdot f(a)(b)$, then it must be a total function. The result is also countably supported, since m and each $f(a)$ are countably supported and $\mathsf{supp}(\mathsf{bind}(m, f)) = \cup_{a\in\mathsf{supp}(m)} \mathsf{supp}(f(a))$. Finally $|\mathsf{bind}(m, f)|$ exists since:

$$\begin{aligned} |\mathsf{bind}(m, f)| &= \sum_{b\in\mathsf{supp}(\mathsf{bind}(m,f))} \sum_{a\in\mathsf{supp}(m)} m(a) \cdot f(a)(b) \\ &= \sum_{a\in\mathsf{supp}(m)} m(a) \cdot \sum_{b\in\mathsf{supp}(\mathsf{bind}(m,f))} f(a)(b) \\ &= \sum_{a\in\mathsf{supp}(m)} m(a) \cdot \sum_{b\in\mathsf{supp}(f(a))} f(a)(b) = \sum_{a\in\mathsf{supp}(m)} m(a) \cdot |f(a)| \end{aligned}$$

And the sum on the last line exists by Lemma A.7.

□

Theorem A.9 (Fixed Point Existence). *The function $F_{\langle C,e,\mathsf{alloc}\rangle}$ defined in Figure 1 has a least fixed point.*

Proof. It will suffice to show that $F_{\langle C,e,\mathsf{alloc}\rangle}$ is Scott continuous, at which point, we can apply the Kleene fixed point theorem to conclude that the least fixed point exists. First, we define the pointwise order for $f_1, f_2 \colon \mathcal{S} \times \mathcal{H} \to \mathcal{W}(\mathrm{St})$ as $f_1 \sqsubseteq f_2$ iff $f_1(s,h) \sqsubseteq f_2(s,h)$ for all (s,h), where $f_1(s,h) \sqsubseteq f_2(s,h)$ iff there exists m such that $f_1(s,h) + m = f_2(s,h)$. Now, we will show that the monad bind is Scott continuous with respect to that order. Let D be a directed set.

$$\sup_{f\in D} \mathsf{bind}(m,f) = \sup_{f\in D} \sum_{s\in\mathsf{supp}(m)} m(s) \cdot \begin{cases} f(a) & \text{if } s = \mathbb{i}_{\mathsf{ok}}(a) \\ \mathsf{unit}_M(s) & \text{otherwise} \end{cases}$$

Now, by continuity of the semiring, suprema distribute over sums and products. It is relatively easy to see by induction that the supremum can move into every summand in the series.

$$\begin{aligned} &= \sum_{s\in\mathsf{supp}(m)} m(s) \cdot \begin{cases} \sup_{f\in D} f(a) & \text{if } s = \mathbb{i}_{\mathsf{ok}}(a) \\ \mathsf{unit}_M(s) & \text{otherwise} \end{cases} \\ &= \sum_{s\in\mathsf{supp}(m)} m(s) \cdot \begin{cases} (\sup D)(a) & \text{if } s = \mathbb{i}_{\mathsf{ok}}(a) \\ \mathsf{unit}_M(s) & \text{otherwise} \end{cases} \\ &= \mathsf{bind}(m, \sup D) \end{aligned}$$

Finally, we show that $F_{\langle C,e,\mathsf{alloc}\rangle}$ is Scott continuous with respect to the order defined above.

$$\begin{aligned} \sup_{f\in D} F_{\langle C,e,\mathsf{alloc}\rangle}(f) &= \lambda(s,h).\ \sup_{f\in D} F_{\langle C,e,\mathsf{alloc}\rangle}(f)(s,h) \\ &= \lambda(s,h).\ \sup_{f\in D} \begin{cases} \mathsf{bind}(\llbracket C\rrbracket_{\mathsf{alloc}}(s,h), f) & \text{if } \llbracket e\rrbracket(s) = \mathbb{1} \\ \mathsf{unit}(s,h) & \text{if } \llbracket e\rrbracket(s) = \mathbb{0} \end{cases} \\ &= \lambda(s,h). \begin{cases} \sup_{f\in D} \mathsf{bind}(\llbracket C\rrbracket_{\mathsf{alloc}}(s,h), f) & \text{if } \llbracket e\rrbracket(s) = \mathbb{1} \\ \sup_{f\in D} \mathsf{unit}(s,h) & \text{if } \llbracket e\rrbracket(s) = \mathbb{0} \end{cases} \\ &= \lambda(s,h). \begin{cases} \mathsf{bind}(\llbracket C\rrbracket_{\mathsf{alloc}}(s,h), \sup D) & \text{if } \llbracket e\rrbracket(s) = \mathbb{1} \\ \mathsf{unit}(s,h) & \text{if } \llbracket e\rrbracket(s) = \mathbb{0} \end{cases} \\ &= F_{\langle C,e,\mathsf{alloc}\rangle}(\sup D) \end{aligned}$$

□

Before proving that the semantics is total, we define the notion of compatible expressions.

Definition A.10 (Compatible Expressions). Given some outcome algebra $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$, two expressions e and e' are *compatible* if $\llbracket e\rrbracket(s) + \llbracket e'\rrbracket(s)$ is defined for all $s \in \mathcal{S}$.

Though compatibility is a semantic notion, there are straightforward syntactic checks to ensure that two expressions are compatible. For example, if e is a test—a Boolean-valued expression where $\llbracket e\rrbracket(s) \in \{\mathbb{0}, \mathbb{1}\}$ for all $s \in \mathcal{S}$—then e and $\neg e$ are compatible. In the probabilistic semiring, p and $1 - p$ are also compatible for any $p \in [0,1]$.

Theorem A.11 (Totality of Program Semantics). *Given an outcome algebra $\langle A, +, \cdot, \mathbb{0}, \mathbb{1}\rangle$, the semantics of a program $\llbracket C\rrbracket_{\mathsf{alloc}}(s,h)$ is defined as long as the following conditions are met:*

(1) *For each loop* while e do C' *appearing in C, the guard e must be a test. More precisely, $\llbracket e\rrbracket(s) \in \{\mathbb{0}, \mathbb{1}\}$ for every $s \in \mathcal{S}$.*

(2) *If the semiring addition is partial, then each use of + in C must be guarded by compatible expressions. That is, they must have the form* $(\mathsf{assume}\ e_1 \fatsemi C_1) + (\mathsf{assume}\ e_2 \fatsemi C_2)$ *where e_1 and e_2 are compatible according to Definition A.10.*

Proof. By induction on the structure of C. The cases for skip, assume, and all atoms are trivial.

- ▹ $C = C_1 \fatsemi C_2$. By the induction hypothesis, we assume $\llbracket C_1 \rrbracket_{\mathsf{alloc}}$ and $\llbracket C_2 \rrbracket_{\mathsf{alloc}}$ are total, and by Lemma A.8 we know that bind is total, therefore $\mathsf{bind}(\llbracket C_1 \rrbracket_{\mathsf{alloc}}(s,h), \llbracket C_2 \rrbracket_{\mathsf{alloc}})$ is total.
- ▹ $C = C_1 + C_2$. If the semiring addition is total, then this case follows trivially from the induction hypothesis. If not, then the program must have the form $(\mathsf{assume}\ e_1 \fatsemi C_1) + (\mathsf{assume}\ e_2 \fatsemi C_2)$ where e_1 and e_2 are compatible, so we have:

$$\begin{aligned}\llbracket C \rrbracket_{\mathsf{alloc}}(s,h) &= \llbracket (\mathsf{assume}\ e_1 \fatsemi C_1) + (\mathsf{assume}\ e_2 \fatsemi C_2) \rrbracket_{\mathsf{alloc}}(s,h) \\ &= \llbracket e_1 \rrbracket(s) \cdot \llbracket C_1 \rrbracket(s,h) + \llbracket e_2 \rrbracket(s) \cdot \llbracket C_2 \rrbracket(s,h)\end{aligned}$$

  By the induction hypothesis, $\llbracket C_1 \rrbracket(s,h)$ and $\llbracket C_2 \rrbracket(s,h)$ are defined. Since e_1 and e_2 are compatible, $\llbracket e_1 \rrbracket(s) + \llbracket e_2 \rrbracket(s)$ is defined too. So, the claim follows by Lemma A.7.
- ▹ $C = \mathsf{while}\ e\ \mathsf{do}\ C$. Follows from Theorem A.9.

□

B PROPERTIES OF LIFTED RELATIONS

Lemma B.1. *If $(m_1, m_2) \in \overline{R}$, then $|m_1| = |m_2|$.*

Proof. We know that there exists an m such that:

$$m_1 = \lambda x. \sum_{y \in \mathsf{supp}(m_2)} m(x,y) \qquad \text{and} \qquad m_2 = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m(x,y)$$

Now, we have:

$$\begin{aligned}|m_1| &= |\lambda x. \sum_{y \in \mathsf{supp}(m_2)} m(x,y)| \\ &= \sum_{x \in \mathsf{supp}(m_1)} \sum_{y \in \mathsf{supp}(m_2)} m(x,y) \\ &= \sum_{y \in \mathsf{supp}(m_2)} \sum_{x \in \mathsf{supp}(m_1)} m(x,y) \\ &= |\lambda y. \sum_{x \in \mathsf{supp}(m_1)} m(x,y)| = |m_2|\end{aligned}$$

□

Lemma B.2. $(\mathbb{0}, m) \in \overline{R}$ *iff* $m = \mathbb{0}$.

Proof.

($\Rightarrow$) We know there must be some m' such that $m = \lambda y. \sum_{x \in \mathsf{supp}(\mathbb{0})} m'(x,y)$, but since $\mathsf{supp}(\mathbb{0}) = \emptyset$, then $m = \mathbb{0}$.

($\Leftarrow$) Let $m' = \mathbb{0}$, so clearly $m' \in \mathcal{W}_{\mathcal{A}} R$ and we also have $\lambda x. \sum_{y \in \mathsf{supp}(\mathbb{0})}(\mathbb{0}(x,y)) = \mathbb{0}$ and $\lambda y. \sum_{x \in \mathsf{supp}(\mathbb{0})}(\mathbb{0}(x,y)) = \mathbb{0}$

□

Lemma B.3. *If $|m_1| + |m_2|$ is defined, then $m_1 + m_2$ is defined.*

Proof. Observe that:

$$|m_1| + |m_2| = \sum_{x \in \mathsf{supp}(m_1)} m_1(x) + \sum_{x \in \mathsf{supp}(m_2)} m_2(x)$$

By associativity:

$$= \sum_{x \in \mathsf{supp}(m_1) \cup \mathsf{supp}(m_2)} m_1(x) + m_2(x)$$

Therefore $m_1(x)+m_2(x)$ is defined for all x, and $|m_1+m_2|$ is defined as well, so m_1+m_2 is defined. □

Lemma B.4. *$(m_1 + m_2, m') \in \overline{R}$ iff there exist m_1' and m_2' such that $m' = m_1' + m_2'$ and $(m_1, m_1') \in \overline{R}$ and $(m_2, m_2') \in \overline{R}$.*

Proof.

($\Rightarrow$) We know that there is some m such that $m_1 + m_2 = \lambda x. \sum_{y \in \mathsf{supp}(m')} m(x, y)$ and $m' = \lambda y. \sum_{x \in \mathsf{supp}(m_1+m_2)} m(x, y)$. So, for each x, we have that $m_1(x)+m_2(x) = \sum_{y \in \mathsf{supp}(m')} m(x, y)$. Using Lemma A.6, we know there must be $(x_k)_{k \in \{1,2\} \times \mathsf{supp}(m')}$ such that $m_i(x) = \sum_{y \in \mathsf{supp}(m')} x_{(i,y)}$ for $i \in \{1, 2\}$ and $m(x, y) = x_{(1,y)} + x_{(2,y)}$ for each $y \in \mathsf{supp}(m')$. Now let $m_1''(x, y) = x_{(1,y)}$ and $m_2''(x, y) = x_{(2,y)}$ and let $m_1' = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m_1''(x, y)$ and $m_2' = \lambda y. \sum_{x \in \mathsf{supp}(m_2)} m_2''(x, y)$. First, we establish that $m_1' + m_2' = m'$:

$$m_1' + m_2' = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m_1''(x, y) + \sum_{x \in \mathsf{supp}(m_2)} m_2''(x, y) = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} x_{(1,y)} + \sum_{x \in \mathsf{supp}(m_2)} x_{(2,y)}$$

If $x \notin \mathsf{supp}(m_1)$, then $x_{(1,y)} = \mathbb{0}$ and similarly for $x_{(2,y)}$ and $\mathsf{supp}(m_2)$, so we can combine the sums

$$= \lambda y. \sum_{x \in \mathsf{supp}(m_1+m_2)} x_{(1,y)} + x_{(2,y)} = \lambda y. \sum_{x \in \mathsf{supp}(m_1+m_2)} m(x, y) = m'$$

Now, observe that:

$$\lambda x. \sum_{y \in \mathsf{supp}(m_i')} m_i''(x, y) = \lambda x. \sum_{y \in \mathsf{supp}(m_i')} x_{(i,y)}$$

Now, for any $y \notin \mathsf{supp}(m_i')$, it must be the case that $m_i'(y) = \mathbb{0}$ and so $\sum_{x \in \mathsf{supp}(m_i)} x_{(i,y)} = \mathbb{0}$, so each $x_{(i,y)} = \mathbb{0}$. This means we can expand the sum to be over $\mathsf{supp}(m_1' + m_2') = \mathsf{supp}(m')$.

$$= \lambda x. \sum_{y \in \mathsf{supp}(m')} x_{(i,y)} = m_i$$

We also know that $m_i' = \lambda y. \sum_{x \in \mathsf{supp}(m_i)} m_i''(x, y)$ by definition, so $(m_i, m_i') \in \overline{R}$.

($\Leftarrow$) We know that $(m_1, m_1') \in \overline{R}$ and $(m_2, m_2') \in \overline{R}$, so there are m_1'' and m_2'' such that $m_i = \lambda x. \sum_{y \in \mathsf{supp}(m_i')} m_i''(x, y)$ and $m_i' = \lambda y. \sum_{x \in \mathsf{supp}(m_i)} m_i''(x, y)$ for $i \in \{1, 2\}$. Let $m'' = m_1''+m_2''$ (we can conclude that this is defined using Lemmas B.1 and B.3). Now we have the following:

$$\lambda x. \sum_{y \in \mathsf{supp}(m')} m''(x, y) = \lambda x. \sum_{y \in \mathsf{supp}(m')} m_1''(x, y) + m_2''(x, y) = \lambda x. \sum_{y \in \mathsf{supp}(m_1')} m_1''(x, y) + \sum_{y \in \mathsf{supp}(m_2')} m_2''(x, y) = m_1 + m_2$$

$$\lambda y. \sum_{x \in \mathsf{supp}(m_1+m_2)} m''(x, y) = \lambda y. \sum_{x \in \mathsf{supp}(m_1+m_2)} m_1''(x, y) + m_2''(x, y) = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m_1''(x, y) + \sum_{x \in \mathsf{supp}(m_2)} m_2''(x, y) = m_1' + m_2' = m'$$

So, $(m_1 + m_2, m') \in \overline{R}$.

□

Lemma B.5. *$(a \cdot m_1, m_2) \in \overline{R}$ iff there exists m_2' such that $m_2 = a \cdot m_2'$ and $(m_1, m_2') \in \overline{R}$*

Proof.

($\Rightarrow$) By the definition of relation lifting, there is an m such that $a \cdot m_1 = \lambda x. \sum_{y \in \mathsf{supp}(m_2)} m(x,y)$ and $m_2 = \lambda y. \sum_{x \in \mathsf{supp}(a \cdot m_1)} m(x,y)$. This means that for all x:

$$a \cdot m_1(x) = \sum_{y \in \mathsf{supp}(m_2)} m(x,y)$$

By Definition A.4, we can obtain $(b_{(x,y)})_{y \in \mathsf{supp}(m_2)}$ such that $\sum_{y \in \mathsf{supp}(m_2)} b_{(x,y)} = \mathbb{1}$ and $m(x,y) = (\sum_{z \in \mathsf{supp}(m_2)} m(x,z)) \cdot b_{(x,y)}$ for all $y \in \mathsf{supp}(m_2)$. Now, define $m''(x,y) = m_1(x) \cdot b_{(x,y)}$ and $m_2'(y) = \sum_{x \in \mathsf{supp}(m_1)} m''(x,y)$. We now show that $m_2 = a \cdot m_2'$:

$$\begin{aligned} a \cdot m_2' &= \lambda y. a \cdot \sum_{x \in \mathsf{supp}(m_1)} m''(x,y) \\ &= \lambda y. a \cdot \sum_{x \in \mathsf{supp}(m_1)} m_1(x) \cdot b_{(x,y)} \\ &= \lambda y. \sum_{x \in \mathsf{supp}(m_1)} a \cdot m_1(x) \cdot b_{(x,y)} \\ &= \lambda y. \sum_{x \in \mathsf{supp}(m_1)} (\sum_{z \in \mathsf{supp}(m_2)} m(x,z)) \cdot b_{(x,y)} \\ &= \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m(x,y) = m_2 \end{aligned}$$

We also have:

$$\lambda x. \sum_{y \in \mathsf{supp}(m_2')} m''(x,y) = \lambda x. \sum_{y \in \mathsf{supp}(m_2')} m_1(x) \cdot b_{(x,y)}$$

Since we already showed that $m_2 = a \cdot m_2'$, then it must be the case that $\mathsf{supp}(m_2) = \mathsf{supp}(m_2')$.

$$= \lambda x. m_1(x) \cdot \sum_{y \in \mathsf{supp}(m_2)} b_{(x,y)} = \lambda x. m_1(x) \cdot \mathbb{1} = m_1$$

And clearly $m_2' = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m''(x,y)$ by definition, so $(m_1, m_2') \in \overline{R}$.

($\Leftarrow$) By the definition of relation lifting, there is some m such that $m_1 = \lambda x. \sum_{y \in \mathsf{supp}(m_2')} m(x,y)$ and $m_2' = \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m(x,y)$. Now, let $m' = a \cdot m$, so this clearly means that $a \cdot m_1 = \lambda x. \sum_{y \in \mathsf{supp}(a \cdot m_2')} m'(x,y)$ and $a \cdot m_2' = \lambda y. \sum_{x \in \mathsf{supp}(a \cdot m_1)} m'(x,y)$, so $(a \cdot m_1, a \cdot m_2') \in \overline{R}$.

□

Lemma B.6. *If $(x,y) \in R$, then $(\mathsf{unit}(x), \mathsf{unit}(y)) \in \overline{R}$*

Proof. Let $m = \mathsf{unit}(x,y)$. We therefore have:

$$\lambda x'. \sum_{y' \in \mathsf{supp}(\mathsf{unit}(y))} m(x', y') = \lambda x'. \mathsf{unit}(x,y)(x', y) = \mathsf{unit}(x)$$

And similarly, $\lambda y'. \sum_{x' \in \mathsf{supp}(\mathsf{unit}(x))} m(x', y') = \mathsf{unit}(y)$, so $(\mathsf{unit}(x), \mathsf{unit}(y)) \in \overline{R}$. □

Lemma B.7. *For any relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$, if $(m_1, m_2) \in \overline{S \circ R}$, then $(m_1, m_2) \in \overline{S} \circ \overline{R}$.*

Proof. Suppose $(m_1, m_2) \in \overline{S \circ R}$, so there is some $m \in \mathcal{W}_{\mathcal{A}}(S \circ R)$ such that $m_1 = \lambda x. \sum_{z \in \mathsf{supp}(m_2)} m(x,z)$ and $m_2 = \lambda z. \sum_{x \in \mathsf{supp}(m_1)} m(x,z)$. This means that for each $(x,z) \in \mathsf{supp}(m)$, there is some y such that $(x, y) \in R$ and $(y, z) \in S$. Let $S' \subseteq S$ be some relation where we choose one y for each z, so that $|\{y \mid (y, z) \in S'\}| = 1$ for each z and $\{z \mid \exists y.(y, z) \in S'\} = \{z \mid \exists y.(y, z) \in S\}$. Now, let:

$$m'(y,z) = \begin{cases} m_2(z) & \text{if } (y,z) \in S' \\ \mathbb{0} & \text{if } (y,z) \notin S' \end{cases} \qquad m''(x,y) = \sum_{z \in \mathsf{supp}(m_2) \mid (y,z) \in S'} m(x,z) \quad m_3 = \lambda y. \sum_{z \in \mathsf{supp}(m_2)} m'(y,z)$$

And now, we have the following:

$$\lambda x. \sum_{y \in \mathsf{supp}(m_3)} m''(x,y) = \lambda x. \sum_{y \in \mathsf{supp}(m_3)} \sum_{z \in \mathsf{supp}(m_2) \mid (y,z) \in S'} m(x,z)$$

Since S' relates each z to exactly one $y \in \mathsf{supp}(m_3)$, this is equivalent to summing over all z

$$= \lambda x. \sum_{z \in \mathsf{supp}(m_2)} m(x,z) = m_1$$

$$\begin{aligned} \lambda y. \sum_{x \in \mathsf{supp}(m_1)} m''(x,y) &= \lambda y. \sum_{x \in \mathsf{supp}(m_1)} \sum_{z \in \mathsf{supp}(m_2) \mid (y,z) \in S'} m(x,z) \\ &= \lambda y. \sum_{z \in \mathsf{supp}(m_2) \mid (y,z) \in S'} \sum_{x \in \mathsf{supp}(m_1)} m(x,z) \\ &= \lambda y. \sum_{z \in \mathsf{supp}(m_2) \mid (y,z) \in S'} m_2(z) = \lambda y. \sum_{z \in \mathsf{supp}(m_2)} m'(y,z) = m_3 \end{aligned}$$

So, $(m_1, m_3) \in \overline{R}$. Also, by definition we know that $m_3 = \lambda y. \sum_{z \in \mathsf{supp}(m_2)} m'(y,z)$ and $m_2 = \lambda z. \sum_{y \in \mathsf{supp}(m_3)} m'(y,z)$ since $m'(y,z)$ is nonzero for exactly one $y \in \mathsf{supp}(m_3)$ and is equal to $m_2(z)$ at that point. This means that $(m_3, m_2) \in \overline{S}$, therefore $(m_1, m_2) \in \overline{S} \circ \overline{R}$.

□

Lemma B.8. *For any pair of directed chains $m_{(i,1)} \sqsubseteq m_{(i,2)} \sqsubseteq \cdots$ for $i \in \{1,2\}$, if $(m_{(1,n)}, m_{(2,n)}) \in \overline{R}$ for all $n \in \mathbb{N}$, then $(\sup_{n\in\mathbb{N}} m_{(1,n)}, \sup_{n\in\mathbb{N}} m_{(2,n)}) \in \overline{R}$.*

Proof. For any $n \in \mathbb{N}$, we know that there is an m_n such that $m_{(1,n)} = \lambda x. \sum_{y \in \mathsf{supp}(m_{(2,n)})} m_n(x,y)$ and $m_{(2,n)} = \lambda y. \sum_{x \in \mathsf{supp}(m_{(1,n)})} m_n(x,y)$. Since each $(m_{(i,n)})_{n\in\mathbb{N}}$ is a chain, $\sup_{n\in\mathbb{N}} m_{(i,n)}$ exists, and:

$$\sup_{n\in\mathbb{N}} m_{(1,n)} = \sup_{n\in\mathbb{N}} (\lambda x. \sum_{y \in \mathsf{supp}(m_{(2,n)})} m_n(x,y))$$

Since we use a pointwise order for functions, the sup of a function is equal to the sup at each point. Additionally, since the semiring is continuous, the sup distributes over the sum.

$$= \lambda x. \sum_{y \in \mathsf{supp}(\sup_{n\in\mathbb{N}} m_{(2,n)})} \sup_{n\in\mathbb{N}} m_n(x,y)$$

And by a similar argument, $\sup_{n\in\mathbb{N}} m_{(2,n)} = \lambda y. \sum_{x \in \mathsf{supp}(\sup_{n\in\mathbb{N}} m_{(1,n)})} \sup_{n\in\mathbb{N}} m_n(x,y)$, so:

$$(\sup_{n\in\mathbb{N}} m_{(1,n)}, \sup_{n\in\mathbb{N}} m_{(2,n)}) \in \overline{R}$$

□

C OUTCOME SEPARATION LOGIC

Lemma C.1 (Normalization). *For any $m \neq \mathbb{0}$, there exists m' such that $|m'| = \mathbb{1}$ and $m = |m| \cdot m'$.*

Proof. By property (3) of Definition 3.3, there must be $(b_s)_{s\in\mathsf{supp}(m)}$ such that $m(s) = (\sum_{t\in\mathsf{supp}(m)} m(t))\cdot b_s$ and $\sum_{s\in\mathsf{supp}(m)} b_s = \mathbb{1}$. Now, let m' be defined as follows:

$$m'(s) = \begin{cases} b_s & \text{if } s \in \mathsf{supp}(m) \\ \mathbb{0} & \text{if } s \notin \mathsf{supp}(m) \end{cases}$$

So, clearly $|m'| = \sum_{s\in\mathsf{supp}(m)} b_s = \mathbb{1}$. For every s, we also have $m(s) = (\sum_{t\in\mathsf{supp}(m)} m(t)) \cdot b_s = |m| \cdot b_s = |m| \cdot m'(s)$, so $m = |m| \cdot m'$. □

Lemma C.2 (Splitting). *If $|m'| \leq |m_1| + |m_2|$, then there exist m_1' and m_2' such that $|m_1'| \leq |m_1|$ and $|m_2'| \leq |m_2|$ and $m' = m_1' + m_2'$.*

Proof. Since $|m_1| + |m_2| \geq |m'|$, then there is some a such that:

$$|m_1| + |m_2| = |m'| + a = a + \sum_{s\in\mathsf{supp}(m')} m'(s)$$

So, by Lemma A.6, there exists $(u_k)_{k\in\{1,2\}\times(1+\mathsf{supp}(m'))}$ such that for all $i \in \{1, 2\}$ and $s \in \mathsf{supp}(m')$:

$$|m_i| = \sum_{s\in 1+\mathsf{supp}(m')} u_{(i,s)} \qquad \text{and} \qquad a = u_{(1,\star)} + u_{(2,\star)} \qquad \text{and} \qquad m'(s) = u_{(1,s)} + u_{(2,s)}$$

Now, let $m_i' = \lambda s.u_{(i,s)}$ for $i \in \{1, 2\}$. So, $m_1' + m_2' = \lambda s.u_{(1,s)} + u_{(2,s)} = m'$. Now, it just remains to show that $|m_i| \geq |m_i'|$:

$$|m_i| = \sum_{s\in 1+\mathsf{supp}(m')} u_{(i,s)} = u_{(i,\star)} + \sum_{s\in\mathsf{supp}(m')} m_i'(s) = u_{(i,\star)} + |m_i'| \geq |m_i'|$$

□

C.1 The Outcome Separating Conjunction

Lemma 4.3. *If $m \vDash \varphi$ and $(m, m') \in \overline{\mathsf{frame}(F)}$, then $m' \vDash \varphi \circledast F$*

Proof. By induction on the structure of φ.

- $\varphi = \top$. Since $\varphi \circledast F = \top$, then clearly $m' \vDash \varphi \circledast F$
- $\varphi = \varphi_1 \vee \varphi_2$. We know $m \vDash \varphi_1$ or $m \vDash \varphi_2$. Without loss of generality, suppose that $m \vDash \varphi_1$. By the induction hypothesis, we know that $m' \vDash \varphi_1 \circledast F$. We can therefore weaken this to conclude that $m' \vDash (\varphi_1 \vee \varphi_2) \circledast F$. The case where $m \vDash \varphi_2$ is symmetrical.
- $\varphi = \varphi_1 \oplus \varphi_2$. We know that $m_1 \vDash \varphi_1$ and $m_2 \vDash \varphi_2$ for some m_1 and m_2 such that $m = m_1 + m_2$. Now, since $(m_1 + m_2, m') \in \overline{\mathsf{frame}(F)}$, by Lemma B.4 there must be m_1' and m_2' such that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$ and $(m_2, m_2') \in \overline{\mathsf{frame}(F)}$ and $m' = m_1' + m_2'$. By the induction hypothesis, $m_1' \vDash \varphi_1 \circledast F$ and $m_2' \vDash \varphi_2 \circledast F$, so $m' \vDash (\varphi_1 \oplus \varphi_2) \circledast F$.
- $\varphi = (\varphi')_a$. We know that $a = \mathbb{0}$ and $m = \mathbb{0}$ or $m_1 \vDash \varphi'$ for some m_1 such that $m = a \cdot m_1$. In the first case, by Lemma B.2, $m' = \mathbb{0}$ and so clearly $m' \vDash (\varphi')_{\mathbb{0}} \circledast F$. In the second case, since $(a \cdot m_1, m') \in \overline{\mathsf{frame}(A)}$, by Lemma B.5 there must be m_1' such that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$ and $m' = a \cdot m_1'$. By the induction hypothesis, $m_1' \vDash \varphi' \circledast F$, so therefore $m' \vDash (\varphi')_a \circledast F$.
- $\varphi = \epsilon : P$. We know that $|m| = \mathbb{1}$ and every $\sigma \in \mathsf{supp}(m)$ has the form $\mathbb{1}_\epsilon(s, h)$ such that $(s, h) \in P$. Since $(m, m') \in \overline{\mathsf{frame}(F)}$, we know by Lemma B.1 that $|m'| = |m| = \mathbb{1}$. Additionally, for every element in $\mathsf{supp}(m')$, there must be an element in $\mathsf{supp}(m)$ related by $\mathsf{frame}(F)$, so each element of m' has the form $\mathbb{1}_\epsilon(s, h \uplus h')$ such that $(s, h) \in P$ and $(s, h') \in F$, and so clearly $(s, h \uplus h') \in P * F$, and therefore also $m' \vDash (\epsilon : P) \circledast F$.

□

Lemma 4.4. *If $m \vDash \varphi \circledast F$, then there exist m_1, m_1', and m_2 such that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$, $m = m_1' + m_2$ and $m_1 + m_2' \vDash \varphi$ for any m_2' such that $|m_2'| \leq |m_2|$.*

Proof. By induction on the structure of φ.

- $\varphi = \top$. Suppose $m \vDash \top \circledast F$. Let $m_1 = m_1' = \mathbb{0}$ and $m_2 = m$. Clearly $(\mathbb{0}, \mathbb{0}) \in \overline{\mathsf{frame}(F)}$ and $m_1' + m_2 = \mathbb{0} + m = m$. Now, taking any m_2', it is obvious that $\mathbb{0} + m_2' \vDash \top$.
- $\varphi = \varphi_1 \vee \varphi_2$. We know $m \vDash \varphi_1 \circledast F$ or $m \vDash \varphi_2 \circledast F$. Without loss of generality, suppose that $m \vDash \varphi_1$. By the induction hypothesis, there are m_1, m_1', and m_2 such that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$ and $m = m_1' + m_2$ and $m_1 + m_2' \vDash \varphi_1$ for any m_2' such that $|m_2'| \leq |m_2|$. Now, take any such m_2', we know that $m_1 + m_2' \vDash \varphi_1$. We can weaken this to conclude that $m_1 + m_2' \vDash \varphi_1 \vee \varphi_2$
- $\varphi = \varphi_1 \oplus \varphi_2$. We know that there are m_1 and m_2 such that $m_1 \vDash \varphi_1 \circledast F$ and $m_2 \vDash \varphi_2 \circledast F$ and $m = m_1 + m_2$. By the induction hypotheses, we get that there are $u_1, u_1', u_2, v_1, v_1'$, and v_2 such that $m_1 = u_1' + u_2$ and $m_2' = v_1' + v_2$ and $(u_1, u_1') \in \overline{\mathsf{frame}(F)}$ and $(v_1, v_1') \in \overline{\mathsf{frame}(F)}$ and $u_1 + u_2' \vDash \varphi_1$ and $v_1 + v_2' \vDash \varphi_2$ whenever $|u_2'| \leq |u_2|$ and $|v_2'| \leq |v_2|$.
  Now, let $m_1 = u_1 + v_1$ and $m_1' = u_1' + v_1'$ and $m_2 = u_2 + v_2$. By Lemma B.4, we get that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$. We also have that:
  $$m_1' + m_2 = (u_1' + v_1') + (u_2 + v_2) = (u_1' + u_2) + (v_1' + v_2) = m_1 + m_2 = m$$
  Now, take any m_2' such that $|m_2'| \leq |m_2| = |u_2| + |v_2|$. By Lemma C.2 we know that there are u_2' and v_2' such that $|u_2'| \leq |u_2|$ and $|v_2'| \leq |v_2|$ and $m_2' = u_2' + v_2'$. This means that $u_1 + u_2' \vDash \varphi_1$ and $v_1 + v_2' \vDash \varphi_2$. We also have:
  $$(u_1 + u_2') + (v_1 + v_2') = (u_1 + v_1) + (u_2' + v_2') = m_1 + m_2'$$
  So, $m_1 + m_2' \vDash \varphi_1 \oplus \varphi_2$.
- $\varphi = (\varphi')_a$. We know that there is m' such that $m' \vDash \varphi'$ and $m = a \cdot m'$. By the induction hypothesis, we get that there are u_1, u_1', and u_2 such that $(u_1, u_1') \in \overline{\mathsf{frame}(F)}$, $m' = u_1' + u_2$, and $u_1 + u_2' \vDash \varphi'$ whenever $|u_2'| \leq |u_2|$.
  Now, let $m_1 = a \cdot u_1$, $m_1' = a \cdot u_1'$, and $m_2 = a \cdot u_2$. By Lemma B.5, we get that $(m_1, m_1') \in \overline{\mathsf{frame}(F)}$. We also have that:
  $$m_1' + m_2 = a \cdot u_1' + a \cdot u_2 = a \cdot (u_1' + u_2) = a \cdot m' = m$$
  Now, take any m_2' such that $|m_2'| \leq |m_2| = a \cdot |u_2|$. If $m_2' = \mathbb{0}$, then $m_1 + m_2' = m_1 = a \cdot u_1$, so $m_1 + m_2' \vDash (\varphi')_a$. If not, then by Lemma C.1 there is m_2'' such that $|m_2''| = \mathbb{1}$ and $m_2' = |m_2'| \cdot m_2''$. By the definition of $\leq$, there must be a' such that $|m_2'| + a' = a \cdot |u_2|$ and by Definition A.4, there must be a b such that $|m_2'| = (|m_2'| + a') \cdot b = a \cdot |u_2| \cdot b$. Now, let $u_2' = |u_2| \cdot b \cdot m_2''$, so $|u_2'| = |u_2| \cdot b \cdot \mathbb{1} \leq |u_2|$, and therefore $u_1 + u_2' \vDash \varphi'$ and also $a \cdot (u_1 + u_2') \vDash (\varphi')_a$. Finally, we have that:
  $$a \cdot (u_1 + u_2') = a \cdot u_1 + a \cdot |u_2| \cdot b \cdot m_2'' = m_1 + |m_2'| \cdot m_2'' = m_1 + m_2'$$
  So, we get that that $m_1 + m_2' \vDash (\varphi')_a$.
- $\varphi = \epsilon : P$. We know that $m \vDash \epsilon : P * F$, so $|m| = \mathbb{1}$ and every element of $\mathrm{supp}(m)$ has the form $\mathbb{i}_\epsilon(s, h \uplus h')$ such that $(s, h) \in P$ and $(s, h') \in F$. Now, let $m_2 = \mathbb{0}$, $m_1' = m$, so clearly $m = m_1' + m_2$. To define m_1, first we fix a relation $S \subseteq \mathsf{frame}(F)$ such that for any h'' such that $(s, h'') \in P * F$, there is a unique h and h' such that $h'' = h \uplus h'$ and $(s, h) \in P$ and $(\mathbb{i}_\epsilon(s, h), \mathbb{i}_\epsilon(s, h \uplus h')) \in S$. Now, m_1 is defined as follows:
  $$m_1(\sigma) = \sum_{\tau \mid (\sigma, \tau) \in S} m(\tau)$$

Now, we must show that $(m_1, m) \in \overline{\mathsf{frame}(F)}$. To do so, first let:

$$m'(\sigma, \tau) = \begin{cases} m(\tau) & \text{if } (\sigma,\tau) \in S \\ \mathbb{0} & \text{otherwise} \end{cases}$$

Now, we have:

$$\lambda\sigma. \sum_{\tau \in \mathsf{supp}(m)} m'(\sigma,\tau) = \lambda\sigma. \sum_{\tau \mid (\sigma,\tau)\in S} m(\tau) = m_1$$

$$\lambda\tau. \sum_{\sigma \in \mathsf{supp}(m_1)} m'(\sigma,\tau) = \lambda\tau. \sum_{\sigma \in \mathsf{supp}(m_1)} \begin{cases} m(\tau) & \text{if } (\sigma,\tau) \in S \\ \mathbb{0} & \text{otherwise} \end{cases}$$

By the definition of S, there is exactly one σ for each τ such that $(\sigma, \tau) \in S$, so:

$$= \lambda\tau.m(\tau) = m$$

So, $(m_1, m) \in \overline{\mathsf{frame}(F)}$, and this also means that $|m_1| = |m| = \mathbb{1}$. Also, by construction, each element of $\mathsf{supp}(m_1)$ has the form $\mathbb{i}_\epsilon(s, h)$ where $(s, h) \in P$, so $m_1 \vDash \epsilon : P$. Since $m_2 = \mathbb{0}$, then it follows that $m_1 + m_2' \vDash \epsilon : P$ for any m_2' such that $|m_2'| \leq |m_2| = \mathbb{0}$.

□

Now, for the proof of the frame property, it will be necessary to relate states based not only on whether they can be obtained by augmenting the heap, but we must also ensure that after augmenting the heap, it is possible to observe the same allocation behavior. To that end, we introduce a slightly modified frame′ relation.

$$\begin{aligned}
&\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}') = \\
&\quad \left\{ (\mathbb{i}_\epsilon(s[X \mapsto n], h), \mathbb{i}_\epsilon(s, h \uplus h'')) \;\middle|\; \begin{array}{l} (s, h'') \vDash F \\ \forall (s_0, h_0).\ s_0(x) = n \Rightarrow \\ \quad \mathsf{alloc}'(s_0, h_0) = \mathsf{alloc}(s_0[X \mapsto s(X)], h_0 \uplus h'') \end{array} \right\} \cup \\
&\quad \{(\mathsf{undef}, \mathsf{undef})\}
\end{aligned}$$

Lemma C.3. *If $m \vDash \varphi$, $X \notin \mathsf{fv}(\varphi)$ and $(m, m') \in \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')}$, then $m' \vDash \varphi \circledast F$*

Proof. Let $R = \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$. The proof is by induction on the structure of φ.

- ▹ $\varphi = \top$. Since $\varphi \circledast F = \top$, then clearly $m' \vDash \varphi \circledast F$
- ▹ $\varphi = \varphi_1 \vee \varphi_2$. We know $m \vDash \varphi_1$ or $m \vDash \varphi_2$. Without loss of generality, suppose that $m \vDash \varphi_1$. By the induction hypothesis, we know that $m' \vDash \varphi_1 \circledast F$. We can therefore weaken this to conclude that $m' \vDash (\varphi_1 \vee \varphi_2) \circledast F$. The case where $m \vDash \varphi_2$ is symmetrical.
- ▹ $\varphi = \varphi_1 \oplus \varphi_2$. We know that $m_1 \vDash \varphi_1$ and $m_2 \vDash \varphi_2$ for some m_1 and m_2 such that $m = m_1 + m_2$. Now, since $(m_1 + m_2, m') \in \overline{R}$, by Lemma B.4 there must be m_1' and m_2' such that $(m_1, m_1') \in \overline{R}$ and $(m_2, m_2') \in \overline{R}$ and $m' = m_1' + m_2'$. By the induction hypothesis, $m_1' \vDash \varphi_1 \circledast F$ and $m_2' \vDash \varphi_2 \circledast F$, so $m' \vDash (\varphi_1 \oplus \varphi_2) \circledast F$.
- ▹ $\varphi = (\varphi')_a$. If $a = \mathbb{0}$, then $m = \mathbb{0}$ and by Lemma B.2 $m' = \mathbb{0}$, so $m' \vDash (\varphi')_{\mathbb{0}} \circledast F$. Otherwise, we know that $m_1 \vDash \varphi'$ for some m_1 such that $m = a \cdot m_1$. Since $(a \cdot m_1, m') \in \overline{R}$, by Lemma B.5 there must be an m_1' such that $(m_1, m_1') \in \overline{R}$ and $m' = a \cdot m_1'$. By the induction hypothesis, $m_1' \vDash \varphi' \circledast F$, so $m' \vDash (\varphi')_a \circledast F$.
- ▹ $\varphi = \epsilon : P$. We know that $|m| = \mathbb{1}$ and every $\sigma \in \mathsf{supp}(m)$ has the form $\mathbb{i}_\epsilon(s[X \mapsto n], h)$ such that $(s[X \mapsto n], h) \in P$, and since $X \notin \mathsf{fv}(P)$, then it must also be that $(s, h) \in P$. Since $(m, m') \in \overline{R}$, we know by Lemma B.1 that $|m'| = |m| = \mathbb{1}$. Additionally, for every element in $\mathsf{supp}(m')$, there must be an element in $\mathsf{supp}(m)$ related by R, so each element of m' has the

form $\mathbb{i}_\epsilon(s, h \uplus h')$ such that $(s, h') \in F$, and so clearly $(s, h \uplus h') \in P * F$, and therefore also $m' \vDash (\epsilon : P) \circledast F$.

□

Lemma C.4. *For any $X \notin \mathsf{fv}(\varphi)$, and* $\mathsf{alloc} \in \mathsf{Alloc}$*, if* $(m_1, m_2) \in \overline{\mathsf{frame}(F)}$ *and* $m_1 + u \vDash \varphi$*, then there exists an* m_1' *and* alloc' *such that* $(m_1', m_2) \in \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')}$ *and* $m_1' + u \vDash \varphi$*.*

Proof. Since $(m_1, m_2) \in \overline{\mathsf{frame}(F)}$, there is some m such that:

$$m_1 = \lambda\sigma. \sum_{\tau \in \mathsf{supp}(m_2)} m(\sigma, \tau) \quad \text{and} \quad m_2 = \lambda\tau. \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau)$$

Since m must by countably supported, we can enumerate the elements of $\mathsf{supp}(m)$, assigning each a unique natural number n. Let $f \colon \mathsf{supp}(m) \to \{n \in \mathbb{N} \mid 1 \le n \le |\mathsf{supp}(m)|\}$ be such a bijection, and note that $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathsf{Val}$, so the codomain of f contains valid program values. Now, let m' be defined as follows:

$$m'(\mathbb{i}_\epsilon(s, h), \tau) = \begin{cases} m(\mathbb{i}_\epsilon(s', h), \tau) & \text{if } \exists s'. f^{-1}(s(X)) = (\mathbb{i}_\epsilon(s', h), \tau) \land s = s'[X \mapsto s(X)] \\ \mathbb{0} & \text{otherwise} \end{cases}$$
$$m'(\mathsf{undef}, \tau) = m(\mathsf{undef}, \tau)$$

Intuitively m' is obtained by taking each $(\mathbb{i}_\epsilon(s, h), \tau)$ in the support of m and updating s so that X has value $f(\mathbb{i}_\epsilon(s, h), \tau)$. Now, we define alloc' as follows:

$$\mathsf{alloc}'(s, h) = \begin{cases} \mathsf{alloc}(s[X \mapsto s'(x)], h \uplus h'') & \text{if } \exists s', h', h''.\ f^{-1}(s(X)) = (\mathbb{i}_\epsilon(s', h'), \mathbb{i}_\epsilon(s', h' \uplus h'')) \\ \mathsf{alloc}(s, h) & \text{otherwise} \end{cases}$$

First, we argue that $\mathsf{alloc}' \in \mathsf{Alloc}$. It is obvious that $\mathsf{alloc}'(s, h)$ cannot return anything in the domain of h, since its definition relies on alloc, run on an even larger heap. In the definition of alloc', there are two cases. In the first case, $|\mathsf{alloc}'(s, h)| = |\mathsf{alloc}(s[X \mapsto s'(x)], h \uplus h'')| = \mathbb{1}$ and in the second case, $|\mathsf{alloc}'(s, h)| = |\mathsf{alloc}(s, h)| = \mathbb{1}$.

We now show that $\mathsf{supp}(m') \subseteq \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$. Take any $(\sigma, \tau) \in \mathsf{supp}(m')$. If $\sigma = \mathsf{undef}$, then it must be the case that $\tau = \mathsf{undef}$ too, since $m'(\mathsf{undef}, \tau) = m(\mathsf{undef}, \tau)$ and $\mathsf{supp}(m) \subseteq \mathsf{frame}(F)$, which only relates undef to itself, therefore $(\sigma, \tau) = (\mathsf{undef}, \mathsf{undef}) \in \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$.

If instead $\sigma = \mathbb{i}_\epsilon(s, h)$, then there must be some s' such that $f^{-1}(s(X)) = (\mathbb{i}_\epsilon(s', h), \tau)$ and $s = s'[X \mapsto s(X)]$. By the definition of f, that means that $(\mathbb{i}_\epsilon(s', h), \tau) \in \mathsf{supp}(m)$, and so $\tau = \mathbb{i}_\epsilon(s', h \uplus h'')$ for some h'' such that $(s', h'') \in F$. In addition, by the definition of alloc', for any s_0 and h_0 such that $s_0(X) = s(X)$, $\mathsf{alloc}'(s_0, h_0) = \mathsf{alloc}(s_0[X \mapsto s'(X)], h_0 \uplus h'')$, therefore:

$$(\sigma, \tau) = (\mathbb{i}_\epsilon(s'[X \mapsto s(X)], h), \mathbb{i}_\epsilon(s', h \uplus h'')) \in \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$$

Now, let $m_1' = \lambda\sigma. \sum_{\tau \in \mathsf{supp}(m_2)} m'(\sigma, \tau)$ and observe that for all τ:

$$\begin{aligned} \sum_{\sigma \in \mathsf{supp}(m_1')} m'(\sigma, \tau) &= m'(\mathsf{undef}, \tau) + \sum_{\mathbb{i}_\epsilon(s,h) \in \mathsf{supp}(m_1')} m'(\mathbb{i}_\epsilon(s, h), \tau) \\ &= m(\mathsf{undef}, \tau) + \sum_{\mathbb{i}_\epsilon(s,h) \in \mathsf{supp}(m_1')} \sum_{v \in \mathsf{Val}} m(\mathbb{i}_\epsilon(s[X \mapsto v], h), \tau) \\ &= \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) = m_2(\tau) \end{aligned}$$

So, we conclude that $(m_1', m_2) \in \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')}$. Finally, for any $\mathbb{i}_\epsilon(s,h)$:

$$\begin{aligned} m_1(\mathbb{i}_\epsilon(s,h)) &= \sum_{\tau \in \mathsf{supp}(m_2)} m(\mathbb{i}_\epsilon(s,h), \tau) \\ &= \sum_{\tau \in \mathsf{supp}(m_2)} m'(\mathbb{i}_\epsilon(s[X \mapsto f(\mathbb{i}_\epsilon(s,h))], h), \tau) \\ &= m_1'(\mathbb{i}_\epsilon(s[X \mapsto f(\mathbb{i}_\epsilon(s,h))], h)) \end{aligned}$$

So m_1 and m_1' differ only in the values of X, which does not appear in φ by assumption, so $m_1' \vDash \varphi$. ◻

C.2 Replacement of Unsafe States

First, we provide the definitions of the Rep relation and the prune operation. Rep relates undef to any other state and $\lightning$, indicating that after framing, an undef state can become any other state, or can diverge ($\lightning$). All ok and er states are only related to themselves. The $\mathsf{prune}\colon \mathcal{W}(\mathsf{St} \cup \{\lightning\}) \to \mathcal{W}(\mathsf{St})$ function removes $\lightning$ from some program configuration m.

$$\mathsf{Rep} = \{(\mathsf{undef}, \sigma) \mid \sigma \in \mathsf{St} \cup \{\lightning\}\} \cup \{(\sigma,\sigma) \mid \sigma \in \mathsf{St}\} \qquad \mathsf{prune}(m)(\sigma) = \begin{cases} m(\sigma) & \text{if } \sigma \neq \lightning \\ \mathbb{0} & \text{if } \sigma = \lightning \end{cases}$$

Now, by lifting the Rep relation, we can prove that replacing undefined states in some program configuration does not affect the validity of outcome assertions. Intuitively, this is true because undef can only be satisfied by $\top$, which is also satisfied by anything else.

Lemma 4.5 (Replacement). *If $m \vDash \varphi$ and $(m, m') \in \overline{\mathsf{Rep}}$, then $\mathsf{prune}(m') \vDash \varphi$*

Proof. By induction on the assertion φ.

- $\varphi = \top$. $\mathsf{prune}(m') \vDash \varphi$ since everything satisfies $\top$.
- $\varphi = \varphi_1 \vee \varphi_2$. Without loss of generality, suppose $m \vDash \varphi_1$. By the induction hypothesis, $\mathsf{prune}(m') \vDash \varphi_1$. We can weaken this to conclude that $\mathsf{prune}(m') \vDash \varphi_1 \vee \varphi_2$.
- $\varphi = \varphi_1 \oplus \varphi_2$. We know that $m_1 \vDash \varphi_1$ and $m_2 \vDash \varphi_2$ for some m_1 and m_2 such that $m_1 + m_2 = m$. Since $(m_1 + m_2, m') \in \overline{\mathsf{Rep}}$, there must be some m_1' and m_2' such that $(m_1, m_1') \in \overline{\mathsf{Rep}}$ and $(m_2, m_2') \in \overline{\mathsf{Rep}}$ and $m' = m_1' + m_2'$. By the induction hypothesis, $\mathsf{prune}(m_1') \vDash \varphi_1$ and $\mathsf{prune}(m_2') \vDash \varphi_2$. It is easy to see that $\mathsf{prune}(m_1') + \mathsf{prune}(m_2') = \mathsf{prune}(m_1' + m_2') = \mathsf{prune}(m')$, therefore $\mathsf{prune}(m') \vDash \varphi_1 \oplus_a \varphi_2$.
- $\varphi = (\varphi')_a$. If $a = \mathbb{0}$, then $m = \mathbb{0}$, and by Lemma B.2 $m' = \mathbb{0}$ too. So $\mathsf{prune}(m') = \mathbb{0}$ and $\mathsf{prune}(m') \vDash (\varphi')_{\mathbb{0}}$. Otherwise, we know that there is an m_1 such that $m_1 \vDash \varphi'$ and $m = a \cdot m_1$. Since $(a \cdot m_1, m') \in \overline{\mathsf{Rep}}$, by Lemma B.5 we know there must be m_1' such that $m' = a \cdot m_1'$ and $(m_1, m_1') \in \overline{\mathsf{Rep}}$. By the induction hypothesis, $\mathsf{prune}(m_1') \vDash \varphi'$, so—since $\mathsf{prune}(m') = \mathsf{prune}(a \cdot m_1') = a \cdot \mathsf{prune}(m_1')$—we get that $\mathsf{prune}(m') \vDash (\varphi')_a \circledast F$.
- $\varphi = (\epsilon : P)$. By definition, $\mathsf{undef} \notin \mathsf{supp}(m)$, so $m' = m$ (since Rep only relates defined states to themselves), which satisfies $\epsilon : P$ by assumption.

◻

C.3 The Frame Rule

Lemma C.5 (Sequencing). *For any $f, g\colon \mathcal{S} \times \mathcal{H} \to \mathcal{W}_{\mathcal{A}}(\mathsf{St})$ and relation $R \subseteq \mathsf{St} \times \mathsf{St} \cup \{\lightning\}$, if $(m_1, m_2) \in \overline{R}$ and:*

$$\forall (\mathbb{i}_{\mathsf{ok}}(s_1,h_1), \mathbb{i}_{\mathsf{ok}}(s_2,h_2)) \in R. \quad \exists m. \quad g(s_2,h_2) = \mathsf{prune}(m) \quad \textit{and} \quad (f(s_1,h_1), m) \in \overline{R}$$

Then, there exists m_2' such that $\mathsf{bind}(\mathsf{prune}(m_2), g) = \mathsf{prune}(m_2')$ and $(\mathsf{bind}(m_1, f), m_2') \in \overline{R}$.

Proof. First, let $f', g' \colon \mathrm{St} \to \mathcal{W}_{\mathcal{A}}(\mathrm{St})$ be defined as follows:

$$f'(\sigma) = \begin{cases} f(s,h) & \text{if } \sigma = \mathbb{i}_{\mathsf{ok}}(s,h) \\ \mathsf{unit}_{\mathcal{W}}(\sigma) & \text{otherwise} \end{cases} \qquad g'(\sigma) = \begin{cases} g(s,h) & \text{if } \sigma = \mathbb{i}_{\mathsf{ok}}(s,h) \\ \mathsf{unit}_{\mathcal{W}}(\sigma) & \text{otherwise} \end{cases}$$

Note that $\mathsf{bind}(m, f) = \mathsf{bind}_{\mathcal{W}}(m, f')$ and $\mathsf{bind}(m, g) = \mathsf{bind}_{\mathcal{W}}(m, g')$ for all m. Now, we argue that if $(\sigma, \tau) \in R$, then there exists $m_{\sigma,\tau}$ such that $g'(\tau) = \mathsf{prune}(m_{\sigma,\tau})$ and $(f'(\sigma), m_{\sigma,\tau}) \in \overline{R}$. We do so by case analysis.

- $\sigma = \mathbb{i}_{\mathsf{ok}}(s,h)$ and $\tau = \mathbb{i}_{\mathsf{ok}}(s',h')$. By definition, $g'(\tau) = g(s',h')$ and $f'(\sigma) = f(s,h)$, so the claim holds by assumption.
- $\sigma = \mathsf{undef}$. In this case, $f'(\sigma) = \mathsf{unit}_{\mathcal{W}}(\mathsf{undef})$, which means that $f'(\sigma)$ is related to all configurations of size $\mathbb{1}$ according to $\overline{R}$. Now, since $\sup(A) = \mathbb{1}$, it must be that $|g'(\tau)| \le \mathbb{1}$ and so there is some u such that $|g'(\tau)| + u = \mathbb{1}$. Now, let $m_{\sigma,\tau} = g'(\tau) + u \cdot \mathsf{unit}_{\mathcal{W}}(\lightning)$, so clearly $g'(\tau) = \mathsf{prune}(m_{\sigma,\tau})$ and $(f'(\sigma), m_{\sigma,\tau}) \in \overline{R}$.
- In this final case, τ cannot have the form $\mathbb{i}_{\mathsf{ok}}$, since only ok and undef states are related to ok states according to R, and we have already handled both of those cases. This means that σ must also not be an ok state, since ok states are only related to other ok states. Therefore, $f'(\sigma) = \mathsf{unit}_{\mathcal{W}}(\sigma)$ and $f'(\tau) = \mathsf{unit}_{\mathcal{W}}(\tau)$. By Lemma B.6, we conclude that $(\mathsf{unit}_{\mathcal{W}}(\sigma), \mathsf{unit}_{\mathcal{W}}(\tau)) \in \overline{R}$.

Given this, we know that for each σ and τ, there must be some $m_{\sigma,\tau}$ and $m'_{\sigma,\tau}$ such that $g'(\tau) = \mathsf{prune}(m_{\sigma,\tau})$ and:

$$f'(\sigma) = \lambda\sigma'.\sum_{\tau' \in \mathsf{supp}(m_{\sigma,\tau})} m'_{\sigma,\tau}(\sigma',\tau') \qquad m_{\sigma,\tau} = \lambda\tau'.\sum_{\sigma' \in \mathsf{supp}(f'(\sigma))} m'_{\sigma,\tau}(\sigma',\tau')$$

Since $(m_1, m_2) \in \overline{R}$, we know that there is an m such that $m_1 = \lambda\sigma.\sum_{\tau \in \mathsf{supp}(m_2)} m(\sigma,\tau)$ and $m_2 = \lambda\tau.\sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma,\tau)$ Now let:

$$m'(\sigma',\tau') = \sum_{\sigma \in \mathsf{supp}(m_1)} \sum_{\tau \in \mathsf{supp}(m_2)} m(\sigma,\tau) \cdot m'_{\sigma,\tau}(\sigma',\tau') \qquad m'_2(\tau') = \sum_{\sigma' \in \mathsf{supp}(\mathsf{bind}(m_1,f))} m'(\sigma',\tau')$$

Now, we show that:

$$\begin{aligned} \lambda\sigma'.\sum_{\tau' \in \mathsf{supp}(m'_2)} m'(\sigma',\tau') &= \lambda\sigma'.\sum_{\tau' \in \mathsf{supp}(m'_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} \sum_{\tau \in \mathsf{supp}(m_2)} m(\sigma,\tau) \cdot m'_{\sigma,\tau}(\sigma',\tau') \\ &= \lambda\sigma'.\sum_{\sigma \in \mathsf{supp}(m_1)} \sum_{\tau \in \mathsf{supp}(m_2)} m(\sigma,\tau) \cdot \sum_{\tau' \in \mathsf{supp}(m'_2)} m'_{\sigma,\tau}(\sigma',\tau') \end{aligned}$$

Now, $\mathsf{supp}(m'_2)$ is all those τ' such that $m'(\sigma',\tau') \ne \mathbb{0}$ for some $\sigma' \in \mathsf{supp}(\mathsf{bind}(m_1, f))$, which also means that $m'_{\sigma,\tau}(\sigma',\tau') \ne \mathbb{0}$. Since the outer sum is over $\sigma \in \mathsf{supp}(m_1)$, the terms we are summing over $(m'_{\sigma,\tau}(\sigma',\tau'))$ will be $\mathbb{0}$ unless $\sigma' \in \mathsf{supp}(f'(\sigma))$. So, the last sum is equivalent to summing over τ' such that there is a $\sigma' \in \mathsf{supp}(f'(\sigma))$ such that $m'_{\sigma,\tau}(\sigma',\tau') \ne \mathbb{0}$, which is exactly $\mathsf{supp}(m_{\sigma,\tau})$.

$$\begin{aligned} &= \lambda\sigma'.\sum_{\sigma \in \mathsf{supp}(m_1)} \sum_{\tau \in \mathsf{supp}(m_2)} m(\sigma,\tau) \cdot \Big(\sum_{\tau' \in \mathsf{supp}(m_{\sigma,\tau})} m'_{\sigma,\tau}(\sigma',\tau')\Big) \\ &= \lambda\sigma'.\sum_{\sigma \in \mathsf{supp}(m_1)} m_1(\sigma) \cdot f'(\sigma)(\sigma') = \mathsf{bind}(m_1, f) \end{aligned}$$

So $(\mathsf{bind}(m_1, f), m'_2) \in \overline{R}$. It now just remains to prove that $\mathsf{bind}(m_2, g) = \mathsf{prune}(m'_2)$:

$$\mathsf{prune}(m'_2) = \mathsf{prune}\Big(\lambda\tau'.\sum_{\sigma' \in \mathsf{supp}(\mathsf{bind}(m_1,f))} m'(\sigma',\tau')\Big)$$

$$= \mathsf{prune}(\lambda\tau'. \sum_{\sigma' \in \mathsf{supp}(\mathsf{bind}(m_1, f))} \sum_{\tau \in \mathsf{supp}(m_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) \cdot m'_{\sigma,\tau}(\sigma', \tau'))$$
$$= \mathsf{prune}(\lambda\tau'. \sum_{\tau \in \mathsf{supp}(m_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) \cdot \sum_{\sigma' \in \mathsf{supp}(\mathsf{bind}(m_1, f))} m'_{\sigma,\tau}(\sigma', \tau'))$$
$$= \mathsf{prune}(\lambda\tau'. \sum_{\tau \in \mathsf{supp}(m_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) \cdot \sum_{\sigma' \in \mathsf{supp}(f'(\sigma))} m'_{\sigma,\tau}(\sigma', \tau'))$$
$$= \mathsf{prune}(\lambda\tau'. \sum_{\tau \in \mathsf{supp}(m_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) \cdot m_{\sigma,\tau}(\tau'))$$
$$= \lambda\tau'. \sum_{\tau \in \mathsf{supp}(m_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) \cdot \mathsf{prune}(m_{\sigma,\tau}(\tau'))$$
$$= \lambda\tau'. \sum_{\tau \in \mathsf{supp}(m_2)} \sum_{\sigma \in \mathsf{supp}(m_1)} m(\sigma, \tau) \cdot g'(\tau)(\tau')$$
$$= \lambda\tau'. \sum_{\tau \in \mathsf{supp}(m_2)} m_2(\tau) \cdot g'(\tau)(\tau') = \mathsf{bind}(m_2, g)$$

□

Lemma C.6. *Let* $R = \mathsf{Rep} \circ \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$*. If* $(\mathbb{i}_{\mathsf{ok}}(s[X \mapsto n], h), \mathbb{i}_{\mathsf{ok}}(s, h \uplus h')) \in R$ *and* $(\mathbb{i}_{\mathsf{ok}}(s_1, h_1), \mathbb{i}_{\mathsf{ok}}(s_2, h_2)) \in R$ *whenever* $h(\llbracket e \rrbracket(s)) \in \mathsf{Val}$*, then*

$$(\mathsf{update}(s[X \mapsto n], h, \llbracket e \rrbracket(s[X \mapsto n]), s_1, h_1), \mathsf{update}(s, h \uplus h', \llbracket e \rrbracket(s), s_2, h_2)) \in \overline{R}$$

Proof. Let $\ell = \llbracket e \rrbracket(s) = \llbracket e \rrbracket(s[X \mapsto n])$ (since X cannot affect the program expression e). We complete the proof by case analysis:

- ▹ $h(\ell) \in \mathsf{Val}$. It must also be the case that $(h \uplus h')(\ell) \in \mathsf{Val}$, since h' is disjoint from h. So, it just remains to show that $(\mathsf{unit}(s_1, h_1), \mathsf{unit}(s_2, h_2)) \in \overline{R}$, which follows from Lemma B.6.
- ▹ $h(\ell) = \bot$. By a similar argument to the previous case, it must be that $(h \uplus h')(\ell) = \bot$ too. So, we just need to show that $(\mathsf{error}(s[X \mapsto n], h), \mathsf{error}(s, h \uplus h')) \in \overline{R}$. We know that $(\mathbb{i}_{\mathsf{er}}(s[X \mapsto n], h), \mathbb{i}_{\mathsf{er}}(s[X \mapsto n], h \uplus h')) \in R$ since R treats ok and er the same, so the claim follows by Lemma B.6.
- ▹ $\ell \notin \mathsf{dom}(h)$. So, $\mathsf{update}(s[X \mapsto n], h, \llbracket e \rrbracket(s), s_1, h_1) = \mathsf{unit}_{\mathcal{W}}(\mathsf{undef})$, and since R relates undef to all states, $(\mathsf{update}(s, h, \llbracket e \rrbracket(s), s_1, h_1), m) \in \overline{R}$ for all m, which means that $\mathsf{update}(s, h \uplus h', \llbracket e \rrbracket(s), s_2, h_2)$ is related trivially.

□

Lemma C.7 (The Frame Property). *Let* $R = \mathsf{Rep} \circ \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$*, so* $R \subseteq \mathsf{St} \times (\mathsf{St} \cup \{\lightning\})$*. For any program* C *such that* $\mathsf{mod}(C) \cap \mathsf{fv}(F) = \emptyset$*:*

$$\forall (\mathbb{i}_{\mathsf{ok}}(s, h), \mathbb{i}_{\mathsf{ok}}(s', h')) \in R. \quad \exists m. \quad \llbracket C \rrbracket_{\mathsf{alloc}}(s', h') = \mathsf{prune}(m) \quad \textit{and} \quad (\llbracket C \rrbracket_{\mathsf{alloc}'}(s, h), m) \in \overline{R}$$

Proof. By induction on the structure of the program C.

- ▹ $C = \mathsf{skip}$. In this case, $\llbracket C \rrbracket_{\mathsf{alloc}'}(s, h) = \mathsf{unit}(s, h)$ and $\llbracket C \rrbracket_{\mathsf{alloc}}(s', h') = \mathsf{unit}(s', h')$, so the claim follows from Lemma B.6.
- ▹ $C = C_1 \fatsemi C_2$. By definition, we know that $\llbracket C \rrbracket_{\mathsf{alloc}'}(s, h) = \mathsf{bind}(\llbracket C_1 \rrbracket_{\mathsf{alloc}'}(s, h), \llbracket C_2 \rrbracket_{\mathsf{alloc}'})$ and $\llbracket C \rrbracket_{\mathsf{alloc}}(s', h') = \mathsf{bind}(\llbracket C_1 \rrbracket_{\mathsf{alloc}}(s', h'), \llbracket C_2 \rrbracket_{\mathsf{alloc}})$. By the induction hypothesis, we know there is some m such that $\llbracket C_1 \rrbracket_{\mathsf{alloc}}(s', h') = \mathsf{prune}(m)$ and $(\llbracket C_1 \rrbracket_{\mathsf{alloc}'}(s, h), m) \in \overline{R}$. Therefore by the induction hypothesis and Lemma C.5, we can conclude that there is some m' such

that $\mathsf{bind}(\llbracket C_1 \rrbracket_{\mathsf{alloc}}(s', h'), \llbracket C_2 \rrbracket_{\mathsf{alloc}}) = \mathsf{prune}(m')$ and $(\mathsf{bind}(\llbracket C_1 \rrbracket_{\mathsf{alloc}'}(s, h), \llbracket C_1 \rrbracket_{\mathsf{alloc}'}), m') \in \overline{R}$, which completes the proof.

- $C = C_1 + C_2$. We know that $\llbracket C_1 + C_2 \rrbracket_{\mathsf{alloc}'}(s, h) = \llbracket C_1 \rrbracket_{\mathsf{alloc}'}(s, h) + \llbracket C_2 \rrbracket_{\mathsf{alloc}'}(s, h)$ and $\llbracket C_1 + C_2 \rrbracket_{\mathsf{alloc}}(s', h') = \llbracket C_1 \rrbracket_{\mathsf{alloc}}(s', h') + \llbracket C_2 \rrbracket_{\mathsf{alloc}}(s', h')$. By the induction hypothesis, for each $i \in \{1, 2\}$ we get that there is some m_i such that $\llbracket C_i \rrbracket_{\mathsf{alloc}}(s', h') = \mathsf{prune}(m_i)$ and $(\llbracket C_i \rrbracket_{\mathsf{alloc}'}(s, h), m_i) \in \overline{R}$. Using Lemma B.4 we can conclude that $(\llbracket C_1 + C_2 \rrbracket_{\mathsf{alloc}'}(s, h), m_1 + m_2) \in \overline{R}$. It now only remains to show that:

$$\begin{aligned}\mathsf{prune}(m_1 + m_2) &= \mathsf{prune}(m_1) + \mathsf{prune}(m_2)\\ &= \llbracket C_1 \rrbracket_{\mathsf{alloc}}(s', h') + \llbracket C_2 \rrbracket_{\mathsf{alloc}}(s', h')\\ &= \llbracket C_1 + C_2 \rrbracket_{\mathsf{alloc}}(s', h')\end{aligned}$$

We now move on to the cases involving state. Since $(\mathbb{i}_{\mathsf{ok}}(s, h), \mathbb{i}_{\mathsf{ok}}(s', h')) \in R$, then there must be some n and h'' such that $s = s'[X \mapsto n]$, $h' = h \uplus h''$, $(s', h'') \vDash F$, and for any s_0 and h_0, if $s_0(X) = n$, then $\mathsf{alloc}'(s_0, h_0) = \mathsf{alloc}(s_0[X \mapsto s'(X)], h_0 \uplus h'')$. Additionally, many of the cases have a single outcome, so by Lemma B.6 it suffices to show that $(\sigma, \tau) \in R$ where $\llbracket C \rrbracket_{\mathsf{alloc}'}(s, h) = \mathsf{unit}_{\mathcal{W}}(\sigma)$ and $\llbracket C \rrbracket_{\mathsf{alloc}}(s', h') = \mathsf{unit}_{\mathcal{W}}(\tau)$ in those cases.

- $C = \mathsf{assume}\ e$. Since s and s' only differ in the value of X, which cannot affect the value of e, then $\llbracket e \rrbracket(s') = \llbracket e \rrbracket(s)$. This means that both programs weight the computation by the same amount, let this weight be $a = \llbracket e \rrbracket(s') = \llbracket e \rrbracket(s)$. So, we get that $\llbracket \mathsf{assume}\ e \rrbracket_{\mathsf{alloc}'}(s, h) = a \cdot \mathsf{unit}(s, h)$ and $\llbracket \mathsf{assume}\ e \rrbracket_{\mathsf{alloc}}(s', h') = a \cdot \mathsf{unit}(s', h')$. Since $(\mathbb{i}_{\mathsf{ok}}(s, h), \mathbb{i}_{\mathsf{ok}}(s', h')) \in R$, then by Lemma B.6 $(\mathsf{unit}(s, h), \mathsf{unit}(s', h')) \in \overline{R}$, so by Lemma B.5 $(a \cdot \mathsf{unit}(s, h), a \cdot \mathsf{unit}(s', h')) \in \overline{R}$.
- $C = \mathsf{while}\ e\ \mathsf{do}\ C'$. First, we will show that there exists m such that $F^n_{\langle C', e, \mathsf{alloc}\rangle}(\bot)(s', h') = \mathsf{prune}(m)$ and $(F^n_{\langle C', e, \mathsf{alloc}'\rangle}(\bot)(s, h), m) \in \overline{R}$ for any $(\mathbb{i}_{\mathsf{ok}}(s, h), \mathbb{i}_{\mathsf{ok}}(s', h')) \in R$. The proof is by induction on n. If $n = 0$, then the claim holds using Lemma B.2:

$$F^0_{\langle C', e, \mathsf{alloc}\rangle}(\bot)(s', h') = \bot(s', h') = \mathbb{0} = \bot(s, h) = F^0_{\langle C', e, \mathsf{alloc}'\rangle}(\bot)(s, h)$$

Now suppose the claim holds for n, we will show that it also holds for $n + 1$:

$$F^{n+1}_{\langle C', e, \mathsf{alloc}\rangle}(\bot)(s', h') = \begin{cases} \mathsf{bind}(\llbracket C' \rrbracket_{\mathsf{alloc}}(s', h'), F^n_{\langle C', e, \mathsf{alloc}\rangle}(\bot)) & \text{if } \llbracket e \rrbracket(s') = \mathbb{1}\\ \mathsf{unit}(s', h') & \text{if } \llbracket e \rrbracket(s') = \mathbb{0}\end{cases}$$

and

$$F^{n+1}_{\langle C', e, \mathsf{alloc}'\rangle}(\bot)(s, h) = \begin{cases} \mathsf{bind}(\llbracket C' \rrbracket_{\mathsf{alloc}'}(s, h), F^n_{\langle C', e, \mathsf{alloc}'\rangle}(\bot)) & \text{if } \llbracket e \rrbracket(s) = \mathbb{1}\\ \mathsf{unit}(s, h) & \text{if } \llbracket e \rrbracket(s) = \mathbb{0}\end{cases}$$

Note that as we showed in the previous case for assume, $\llbracket e \rrbracket(s) = \llbracket e \rrbracket(s')$, so both executions will take the same path. If $\llbracket e \rrbracket(s') = \llbracket e \rrbracket(s) = \mathbb{1}$, then the claim holds by Lemma C.5 and the induction hypothesis. In the second case, it holds by Lemma B.6.

Now, by the definition of prune, this also means that for any n, there exists a_n such that:

$$(F^n_{\langle C', e, \mathsf{alloc}'\rangle}(\bot)(s, h), F^n_{\langle C', e, \mathsf{alloc}\rangle}(\bot)(s', h') + a_n \cdot \mathsf{unit}(\lightning)) \in \overline{R}$$

Recall that $\lightning$ represents the nonterminating traces, and as we continue to unroll the loop, the weight of nontermination can only increase, so the a_n must increase monotonically and therefore $F^n_{\langle C', e, \mathsf{alloc}\rangle}(\bot)(s', h') + a_n \cdot \mathsf{unit}(\bot)$ is a chain, so by Lemma B.8 we know that:

$$(\sup_{n \in \mathbb{N}} F^n_{\langle C', e\rangle}(\bot)(s, h), \sup_{n \in \mathbb{N}} F^n_{\langle C', e\rangle}(\bot)(s', h') + a_n \cdot \mathsf{unit}(\lightning)) \in \overline{R}$$

And we can therefore conclude that there exists m such that $\llbracket \mathsf{while}\ e\ \mathsf{do}\ C' \rrbracket(s', h') = \mathsf{prune}(m)$ and $(\llbracket \mathsf{while}\ e\ \mathsf{do}\ C' \rrbracket(s, h), m) \in \overline{R}$.

▹ $C = (x := e)$. We know the following:

$$\begin{aligned}\llbracket C\rrbracket_{\mathsf{alloc}'}(s'[X\mapsto n], h) &= \mathsf{unit}(s'[X\mapsto n, x\mapsto \llbracket e\rrbracket(s'[X\mapsto n])], h)\\ &= \mathsf{unit}(s'[x\mapsto \llbracket e\rrbracket(s')][X\mapsto n], h)\\ \llbracket C\rrbracket_{\mathsf{alloc}}(s', h\uplus h'') &= \mathsf{unit}(s'[x\mapsto \llbracket e\rrbracket(s')], h\uplus h'')\end{aligned}$$

Since $x\in\mathsf{mod}(C)$, then $x\notin\mathsf{fv}(F)$, so updating x in s will not affect the truth of F, therefore $(s'[x\mapsto\llbracket e\rrbracket(s')], h\uplus h'')\vDash F$. In addition, since we did not modify the value of X, it is still true that $\mathsf{alloc}'(s_0,h_0) = \mathsf{alloc}(s_0[X\mapsto s'(X)], h_0\uplus h'')$ for any s_0, h_0 with $s_0(X) = n$. So, we know that:

$$(\mathbb{i}_{\mathsf{ok}}(s'[x\mapsto\llbracket e\rrbracket(s')][X\mapsto n], h), \mathbb{i}_{\mathsf{ok}}(s[x\mapsto\llbracket e\rrbracket(s')], h\uplus h''))\in\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$$

Putting this together along with the fact that Rep is reflexive finishes the proof.

▹ $C = (x := \mathsf{alloc}())$. We know the following:

$$\begin{aligned}\llbracket C\rrbracket_{\mathsf{alloc}'}(s'[X\mapsto n], h) &= \mathsf{bind}_{\mathcal{W}}(\mathsf{alloc}'(s'[X\mapsto n], h), \lambda(\ell,v).\mathsf{unit}(s'[X\mapsto n][x\mapsto\ell], h[\ell\mapsto v]))\\ &= \mathsf{bind}_{\mathcal{W}}(\mathsf{alloc}(s', h\uplus h''), \lambda(\ell,v).\mathsf{unit}(s'[x\mapsto\ell][X\mapsto n], h[\ell\mapsto v]))\\ \llbracket C\rrbracket_{\mathsf{alloc}}(s', h\uplus h'') &= \mathsf{bind}_{\mathcal{W}}(\mathsf{alloc}(s', h\uplus h''), \lambda(\ell,v).\mathsf{unit}(s'[x\mapsto\ell], (h\uplus h'')[\ell\mapsto v]))\\ &= \mathsf{bind}_{\mathcal{W}}(\mathsf{alloc}(s', h\uplus h''), \lambda(\ell,v).\mathsf{unit}(s'[x\mapsto\ell], h[\ell\mapsto v]\uplus h''))\end{aligned}$$

So it is clear that $(\llbracket C\rrbracket_{\mathsf{alloc}'}(s,h), \llbracket C\rrbracket_{\mathsf{alloc}}(s, h\uplus h'))\in\overline{R}$ since the two weighting functions are identical except that $\llbracket C\rrbracket_{\mathsf{alloc}}(s, h\uplus h'')$ has h'' added at every state and just as in the previous case, we did not update X, so the relationship between alloc and alloc′ holds as well.

▹ $C = \mathsf{free}(e)$. Using Lemma C.6, we only need to show that if $h(\llbracket e\rrbracket(s'))\in\mathsf{Val}$, then:

$$(\mathbb{i}_{\mathsf{ok}}(s'[X\mapsto n], h[\llbracket e\rrbracket(s)\mapsto\bot]), \mathbb{i}_{\mathsf{ok}}(s', (h\uplus h'')[\llbracket e\rrbracket(s')\mapsto\bot]))\in\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')\subseteq R$$

If $h(\llbracket e\rrbracket(s'))\in\mathsf{Val}$, then $\llbracket e\rrbracket(s')\in\mathsf{dom}(h)$, and since h'' is disjoint from h, then $\llbracket e\rrbracket(s')\notin\mathsf{dom}(h'')$, so $(h\uplus h'')[\llbracket e\rrbracket(s')\mapsto\bot] = h[\llbracket e\rrbracket(s')\mapsto\bot]\uplus h''$, and clearly:

$$(\mathbb{i}_{\mathsf{ok}}(s'[X\mapsto n], h[\llbracket e\rrbracket(s')\mapsto\bot]), \mathbb{i}_{\mathsf{ok}}(s', h[\llbracket e\rrbracket(s')\mapsto\bot]\uplus h''))\in\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$$

▹ $C = ([e_1]\leftarrow e_2)$. Using Lemma C.6, we only need to show that if $h(\llbracket e_1\rrbracket(s'))\in\mathsf{Val}$, then:

$$\begin{aligned}&(\mathbb{i}_{\mathsf{ok}}(s'[X\mapsto n], h[\llbracket e_1\rrbracket(s')\mapsto\llbracket e_2\rrbracket(s')]), \mathbb{i}_{\mathsf{ok}}(s', (h\uplus h'')[\llbracket e_1\rrbracket(s')\mapsto\llbracket e_2\rrbracket(s')]))\\ &\quad\in\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')\\ &\quad\subseteq R\end{aligned}$$

If $h(\llbracket e_1\rrbracket(s'))\in\mathsf{Val}$, then $\llbracket e_1\rrbracket(s')\in\mathsf{dom}(h)$, and since h'' is disjoint from h, then $\llbracket e_1\rrbracket(s')\notin\mathsf{dom}(h'')$, so $(h\uplus h'')[\llbracket e_1\rrbracket(s')\mapsto\llbracket e_2\rrbracket(s')] = h)[\llbracket e_1\rrbracket(s')\mapsto\llbracket e_2\rrbracket(s')]\uplus h''$. Now, clearly:

$$(\mathbb{i}_{\mathsf{ok}}(s'[X\mapsto n], h[\llbracket e_1\rrbracket(s')\mapsto\llbracket e_2\rrbracket(s')]), \mathbb{i}_{\mathsf{ok}}(s', h[\llbracket e_1\rrbracket(s')\mapsto\llbracket e_2\rrbracket(s')]\uplus h''))\in R$$

▹ $C = (x\leftarrow[e])$. Using Lemma C.6, we only need to show that if $h(\llbracket e\rrbracket(s'))\in\mathsf{Val}$, then:

$$(\mathbb{i}_{\mathsf{ok}}(s[X\mapsto n][x\mapsto h(\llbracket e\rrbracket(s')), h), \mathbb{i}_{\mathsf{ok}}(s'[x\mapsto(h\uplus h'')(\llbracket e\rrbracket(s')], h\uplus h''))\in R$$

If $h(\llbracket e\rrbracket(s'))\in\mathsf{Val}$, then $\llbracket e\rrbracket(s')\in\mathsf{dom}(h)$, and since h'' is disjoint from h, then $\llbracket e\rrbracket(s')\notin\mathsf{dom}(h'')$, so $s'[x\mapsto(h\uplus h'')(\llbracket e\rrbracket(s'))] = s'[x\mapsto h(\llbracket e\rrbracket(s'))]$.
So, clearly $(\mathbb{i}_{\mathsf{ok}}(s'[x\mapsto h(\llbracket e\rrbracket(s)][X\mapsto n]), h), \mathbb{i}_{\mathsf{ok}}(s'[x\mapsto h(\llbracket e\rrbracket(s'))]), h\uplus h'')\in R$.

▹ $C = \mathsf{error}()$. By Lemma B.6, it suffices to show that $(\mathbb{i}_{\mathsf{er}}(s'[X\mapsto n], h), \mathbb{i}_{\mathsf{er}}(s', h\uplus h''))\in R$, which follows from the assumptions since R treats ok and er states in the same way.

- $C = f(\vec{e})$. Let C' be the body of f. By the same argument used in the assignment case, $(\mathbb{1}_{\mathsf{ok}}(s'[X \mapsto n][\vec{x} \mapsto [\![\vec{e}]\!](s')], h), \mathbb{1}_{\mathsf{ok}}(s'[\vec{x} \mapsto [\![\vec{e}]\!](s')]), h \uplus h'')) \in R$. So, the claim follows from the induction hypothesis.

□

Theorem 4.6 (The Frame Rule). *If* $\vDash \langle\varphi\rangle\ C\ \langle\psi\rangle$ *and* $\mathsf{mod}(C) \cap \mathsf{fv}(F) = \emptyset$, *then* $\vDash \langle\varphi \circledast F\rangle\ C\ \langle\psi \circledast F\rangle$.

Proof. Let $R = \mathsf{Rep} \circ \mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')$. Suppose $m \vDash \varphi \circledast F$, take any alloc $\in$ Alloc, and pick some $X \notin \mathsf{fv}(\varphi, \psi, F)$. Then by Lemmas 4.4 and C.4 we know that there are m_1, m_1', m_2, and alloc′ such that $(m_1, m_1') \in \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')}$ and $m = m_1' + m_2$ and $m_1 + m_2' \vDash \varphi$ for any m_2' such that $|m_2'| \le |m_2|$. So that means that $m_1 + |m_2| \cdot \mathsf{unit}(\mathsf{undef}) \vDash \varphi$. We know that:

$$[\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m_1 + |m_2| \cdot \mathsf{unit}(\mathsf{undef})) = [\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m_1) + |m_2| \cdot \mathsf{unit}(\mathsf{undef})$$

So, therefore $[\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m_1) + |m_2| \cdot \mathsf{unit}(\mathsf{undef}) \vDash \psi$ since $\vDash \langle\varphi\rangle\ C\ \langle\psi\rangle$. Now, observe that $(m_1, m_1') \in \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')} \subseteq \overline{R}$ since Rep is reflexive. We also know that $(|m_2| \cdot \mathsf{unit}(\mathsf{undef}), m_2) \in \overline{R}$ since R permits undef states to be remapped to anything. Therefore, using Lemma B.4 we get that:

$$(m_1 + |m_2| \cdot \mathsf{unit}(\mathsf{undef}), m) = (m_1 + |m_2| \cdot \mathsf{unit}(\mathsf{undef}), m_1' + m_2) \in \overline{R}$$

So, using Lemmas C.5 and C.7, we know that there exists some m' such that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) = \mathsf{prune}(m')$ and:

$$([\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m_1 + |m_2| \cdot \mathsf{unit}(\mathsf{undef})), m') \in \overline{R} \subseteq \overline{\mathsf{Rep}} \circ \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')}$$

Where the last step is by Lemma B.7. All that remains now is to peel away the layers in the above expression. More concretely, we know that there is some m'' such that $([\![C]\!]^{\dagger}_{\mathsf{alloc}'}(m_1 + |m_2| \cdot \mathsf{unit}(\mathsf{undef})), m'') \in \overline{\mathsf{frame}'(F, X, \mathsf{alloc}, \mathsf{alloc}')}$ and $(m'', m') \in \overline{\mathsf{Rep}}$. By Lemma C.3 $m'' \vDash \psi \circledast F$ and by Lemma 4.5, $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash \psi \circledast F$.

□

D TRI-ABDUCTION

The full set of inference rules for the tri-abduction proof system is shown in Figure 6. The abduction algorithm is given in Algorithm 1.

Lemma D.1. *If* $P \lhd [M] \rhd Q$ *is derivable, then* $M \vDash P$ *and* $M \vDash Q$

Proof. The proof is by induction on the derivation of $P \lhd [M] \rhd Q$.

(1) **Base-Emp.** We need to show that $\Pi \land \Pi' \land \mathsf{emp} \vDash \Pi \land \mathsf{emp}$ and $\Pi \land \Pi' \land \mathsf{emp} \vDash \Pi' \land \mathsf{emp}$, both hold by the semantics of logical conjunction.
(2) **Base-True-L.** We need to show that $\Pi \land \Pi' \land \Sigma' \vDash \Pi \land \mathsf{true}$ and $\Pi \land \Pi' \land \Sigma' \vDash \Pi' \land \Sigma'$, both hold by the semantics of logical conjunction.
(3) **Base-True-R.** This case is symmetric to Case 2.
(4) **Exists.** Here, we know that $M \vDash \Delta$ and $M \vDash \Delta'$. We also know that $\vec{X}$ is not free in Δ' with $\vec{Y}$ removed and vice versa. Now let:
$$\vec{Z} = \vec{X} \cap \vec{Y} \qquad \vec{X}' = \vec{X} \setminus \vec{Z} \qquad \vec{Y}' = \vec{Y} \setminus \vec{Z}$$
That is, $\vec{Z}$ is the variables occurring in both $\vec{X}$ and $\vec{Y}$, $\vec{X}'$ is the variables only occurring in $\vec{X}$ and $\vec{Y}'$ is the variables only occurring in $\vec{Y}$. This means that $\vec{X}'$, $\vec{Y}'$, and $\vec{Z}$ are disjoint.
We first show that $\exists\vec{X}\vec{Y}.M \vDash \exists\vec{X}.\Delta$. Suppose $(s, h) \vDash \exists\vec{X}\vec{Y}.M$. We know that $(s', h) \vDash M$ where $s' = s[\vec{X}' \mapsto \vec{v}_1][\vec{Y}' \mapsto \vec{v}_2][\vec{Z} \mapsto v_3]$ for some $\vec{v}_1, \vec{v}_2$ and $\vec{v}_3$. Given that we know $M \vDash \Delta$, we now have $(s', h) \vDash \Delta$ and therefore $(s[Y' \mapsto \vec{v}_2], h) \vDash \exists\vec{X}.\Delta$ (since $\vec{X} = \vec{X}' \cup \vec{Z}$).

Algorithm 1 abduce-par(P,Q)

if either Base-Emp, Base-True-L, or Base-True-R apply **then**
  return anti-frame M as indicated by that rule
else
  for Each remaining row in Figure 6 from top to bottom **do**
    result = $\emptyset$
    for Each inference rule in the row of the form below **do**

$$\frac{P' \lhd [M'] \rhd Q' \qquad R}{P \lhd [M] \rhd Q}$$

      if The input parameters match P and Q in the inference rule and R is true **then**
        result = result $\cup \{M \mid M' \in \text{abduce-par}(P', Q')\}$
      end if
    end for
    if result $\neq \emptyset$ **then**
      return result
    end if
  end for
  return $\emptyset$
end if

Now, given that $\vec{Y} \cap (\mathsf{fv}(\Delta) \setminus \vec{X}) = \emptyset$, we know that none of the variables in $\vec{Y}'$ are free in Δ, and so we can remove them from the state to conclude that $(s, h) \vDash \exists\vec{X}.\Delta$.
It can also be shown that $\exists\vec{X}\vec{Y}.M \vDash \exists\vec{Y}.\Delta'$ by a symmetric argument.

(5) **Ls-Start-L.** Here, we know $M \vDash \Delta * \mathsf{ls}(e_3, e_2)$ and $M \vDash \Delta'$. We now want to show $M * e_1 \mapsto e_3 \vDash \Delta * \mathsf{ls}(e_1, e_2)$ and $M * e_1 \mapsto e_3 \vDash \Delta' * e_1 \mapsto e_3$.
Suppose that $(s, h) \vDash M * e_1 \mapsto e_3$, and so $(s, h_1) \vDash M$ and $(s, h_2) \vDash e_1 \mapsto e_3$ for some h_1 and h_2 such that $h = h_1 \uplus h_2$.
Since $M \vDash \Delta * \mathsf{ls}(e_3, e_2)$, we get that $(s, h_1) \vDash \Delta * \mathsf{ls}(e_3, e_2)$ and recombining, we get that $(s, h) \vDash \Delta * \mathsf{ls}(e_3, e_2) * e_1 \mapsto e_3$. Now, $\mathsf{ls}(e_3, e_2) * e_1 \mapsto e_3 \vDash \exists X.e_1 \mapsto X * \mathsf{ls}(X, e_2)$ and $\exists X.e_1 \mapsto X * \mathsf{ls}(X, e_2) \vDash \mathsf{ls}(e_1, e_2)$, so we get that $(s, h) \vDash \Delta * \mathsf{ls}(e_1, e_2)$ and therefore $M * e_1 \mapsto e_3 \vDash \Delta * \mathsf{ls}(e_1, e_2)$.
We also know that $M \vDash \Delta'$ which means that because $(s, h_1) \vDash M$, $(s, h_1) \vDash \Delta'$. This means that $(s, h_1 \uplus h_2) \vDash \Delta' * e_1 \mapsto e_3$ and $h = h_1 \uplus h_2$, so we now have $(s, h) \vDash \Delta' * e_1 \mapsto e_3$, therefore $M * e_1 \mapsto e_3 \vDash \Delta' * e_1 \mapsto e_3$

(6) **Ls-Start-R.** This case is symmetric to Case 5.

(7) **Match.** Here, we know that $M \vDash \Delta \land e_2 = e_3$ and $M \vDash \Delta' \land e_2 = e_3$. We want to show that $M * e_1 \mapsto e_2 \vDash \Delta * e_1 \mapsto e_2$ and $M * e_1 \mapsto e_2 \vDash \Delta' * e_1 \mapsto e_3$.
Let us first show $M * e_1 \mapsto e_2 \vDash \Delta * e_1 \mapsto e_2$. Because we know that $M \vDash \Delta$ and $e_1 \mapsto e_2 \vDash e_1 \mapsto e_2$, it follows that $M * e_1 \mapsto e_2 \vDash \Delta * e_1 \mapsto e_2$.
Let us now show $M * e_1 \mapsto e_2 \vDash \Delta' * e_1 \mapsto e_3$. We know $M \vDash \Delta' \land e_2 = e_3$, now suppose that $(s, h) \vDash M * e_1 \mapsto e_2$, so $(s, h) \vDash \Delta' \land e_2 = e_3$ as well. This means that $e_2 = e_3$ in state s, and therefore it must also be the case that $(s, h) \vDash e_1 \mapsto e_3$. Given what else we know, we conclude that $(s, h) \vDash \Delta' * e_1 \mapsto e_3$.

(8) **Ls-End-L.** Here, we know that $M \vDash \Delta * \mathsf{ls}(e_3, e_2)$ and $M \vDash \Delta'$. We want to show that $M * \mathsf{ls}(e_1, e_3) \vDash \Delta * \mathsf{ls}(e_1, e_2)$ and $M * \mathsf{ls}(e_1, e_3) \vDash \Delta' * \mathsf{ls}(e_1, e_3)$.

Base Cases

$$\frac{\Pi \wedge \Pi' \nvDash \text{false}}{\Pi \wedge \text{emp} \lhd [\Pi \wedge \Pi' \wedge \text{emp}] \rhd \Pi' \wedge \text{emp}}\ \textsc{Base-Emp}$$

$$\frac{\Pi \wedge \Pi' \wedge \Sigma' \nvDash \text{false}}{\Pi \wedge \text{true} \lhd [\Pi \wedge \Pi' \wedge \Sigma'] \rhd \Pi' \wedge \Sigma'}\ \textsc{Base-True-L} \qquad \frac{\Pi \wedge \Pi' \wedge \Sigma \nvDash \text{false}}{\Pi \wedge \Sigma \lhd [\Pi \wedge \Pi' \wedge \Sigma] \rhd \Pi' \wedge \text{true}}\ \textsc{Base-True-R}$$

Quantifier Elimination

$$\frac{\Delta \lhd [M] \rhd \Delta' \qquad \vec{X} \cap (\mathsf{fv}(\Delta') \setminus \vec{Y}) = \emptyset \qquad \vec{Y} \cap (\mathsf{fv}(\Delta) \setminus \vec{X}) = \emptyset}{\exists \vec{X}.\Delta \lhd [\exists \vec{X}\vec{Y}.M] \rhd \exists \vec{Y}.\Delta'}\ \textsc{Exists}$$

Resource Matching

$$\frac{\Delta * \mathsf{ls}(e_3, e_2) \lhd [M] \rhd \Delta'}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M * e_1 \mapsto e_3] \rhd \Delta' * e_1 \mapsto e_3}\ \textsc{Ls-Start-L} \qquad \frac{\Delta \lhd [M] \rhd \Delta' * \mathsf{ls}(e_3, e_2)}{\Delta * e_1 \mapsto e_3 \lhd [M * e_1 \mapsto e_3] \rhd \Delta' * \mathsf{ls}(e_1, e_2)}\ \textsc{Ls-Start-R}$$

$$\frac{\Delta \wedge e_2 = e_3 \lhd [M] \rhd \Delta' \wedge e_2 = e_3}{\Delta * e_1 \mapsto e_2 \lhd [M * e_1 \mapsto e_2] \rhd \Delta' * e_1 \mapsto e_3}\ \textsc{Match}$$

$$\frac{\Delta * \mathsf{ls}(e_3, e_2) \lhd [M] \rhd \Delta'}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M * \mathsf{ls}(e_1, e_3)] \rhd \Delta' * \mathsf{ls}(e_1, e_3)}\ \textsc{Ls-End-L} \qquad \frac{\Delta \lhd [M] \rhd \mathsf{ls}(e_2, e_3) * \Delta'}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M * \mathsf{ls}(e_1, e_2)] \rhd \Delta' * \mathsf{ls}(e_1, e_3)}\ \textsc{Ls-End-R}$$

Resource Adding

$$\frac{\Delta \lhd [M] \rhd \Pi' \wedge (\Sigma' * \text{true}) \qquad \Pi' \wedge \Sigma' * B(e_1, e_2) \nvDash \text{false}}{\Delta * B(e_1, e_2) \lhd [M * B(e_1, e_2)] \rhd \Pi' \wedge (\Sigma' * \text{true})}\ \textsc{Missing-L}$$

$$\frac{\Pi \wedge (\Sigma * \text{true}) \lhd [M] \rhd \Delta' \qquad \Pi \wedge \Sigma * B(e_1, e_2) \nvDash \text{false}}{\Pi \wedge (\Sigma * \text{true}) \lhd [M * B(e_1, e_2)] \rhd \Delta' * B(e_1, e_2)}\ \textsc{Missing-R}$$

$$\frac{\Delta \wedge e_1 = e_2 \lhd [M] \rhd \Delta' \wedge e_1 = e_2}{\Delta * \mathsf{ls}(e_1, e_2) \lhd [M] \rhd \Delta'}\ \textsc{Emp-Ls-L} \qquad \frac{\Delta \wedge e_1 = e_2 \lhd [M] \rhd \Delta' \wedge e_1 = e_2}{\Delta \lhd [M] \rhd \Delta' * \mathsf{ls}(e_1, e_2)}\ \textsc{Emp-Ls-R}$$

Fig. 6. Tri-abduction proof rules. Similarly to Calcagno et al. [2009], in the above we use $B(e_1, e_2)$ to represent either $\mathsf{ls}(e_1, e_2)$ or $e_1 \mapsto e_2$.

Let us first show $M * \mathsf{ls}(e_1, e_3) \vDash \Delta * \mathsf{ls}(e_1, e_2)$. First, we know that $M \vDash \Delta * \mathsf{ls}(e_3, e_2)$, and so $M * \mathsf{ls}(e_1, e_3) \vDash \Delta * \mathsf{ls}(e_3, e_2) * \mathsf{ls}(e_1, e_3)$. Clearly, it is also the case that $\mathsf{ls}(e_3, e_2) * \mathsf{ls}(e_1, e_3) \vDash \mathsf{ls}(e_1, e_2)$, and so combining these facts we get $M * \mathsf{ls}(e_1, e_3) \vDash \Delta * \mathsf{ls}(e_1, e_2)$.
Let us now show $M * \mathsf{ls}(e_1, e_3) \vDash \Delta' * \mathsf{ls}(e_1, e_3)$. We know that $M \vDash \Delta'$, so clearly $M * \mathsf{ls}(e_1, e_3) \vDash \Delta' * \mathsf{ls}(e_1, e_3)$.

(9) **Ls-End-R.** This case is symmetric to Case 8.

(10) **Missing-L.** Here, we know that $M \vDash \Delta$ and $M \vDash \Pi' \wedge (\Sigma' * \text{true})$. This means we also know that $M \vDash \Pi'$ and $M \vDash \Sigma' * \text{true}$ by semantic definition.
Let us first show that $M * B(e_1, e_2) \vDash \Delta * B(e_1, e_2)$. Suppose that $(s, h) \vDash M * B(e_1, e_2)$. We know that $(s, h_1) \vDash M$ and $(s, h_2) \vDash B(e_1, e_2)$ for some h_1 and h_2 such that $h = h_1 \uplus h_2$. Since $M \vDash \Delta$, then $(s, h_1) \vDash \Delta$. This means that $(s, h) \vDash \Delta * B(e_1, e_2)$, therefore $M * B(e_1, e_2) \vDash \Delta * B(e_1, e_2)$.
Let us now show that $M * B(e_1, e_2) \vDash \Pi' * (\Sigma' * \text{true})$. Here, we know that $M \vDash \Pi' * (\Sigma' * \text{true})$ and trivially $B(e_1, e_2) \vDash \text{true}$. This means $M * B(e_1, e_2) \vDash \Pi' * (\Sigma' * \text{true}) * \text{true}$; therefore, $M * B(e_1, e_2) \vDash \Pi' * (\Sigma' * \text{true})$.

(11) **Missing-R.** This case is symmetric to Case 10.

(12) **Emp-Ls-L.** We know that $M \vDash \Delta \wedge e_1 = e_2$ and $M \vDash \Delta' \wedge e_1 = e_2$. This means that $M \vDash \Delta'$, so the right side of the tri-abductive judgement is taken care of.
Let us now show $M \vDash \Delta * \mathsf{ls}(e_1, e_2)$. We first establish that $e_1 = e_2 \wedge \mathsf{emp} \vDash \mathsf{ls}(e_1, e_2)$ by definition, since $\mathsf{ls}(e_1, e_2) \iff (\mathsf{emp} \wedge e_1 = e_2) \vee \exists X. e_1 \mapsto X * \mathsf{ls}(X, e_2)$. Now, we know that $M \vDash \Delta \wedge e_1 = e_2$, which also means that $M \vDash (\Delta * \mathsf{emp}) \wedge e_1 = e_2$, or equivalently, $M \vDash (\Delta \wedge e_1 = e_2) * (\mathsf{emp} \wedge e_1 = e_2)$. Using $e_1 = e_2 \wedge \mathsf{emp} \vDash \mathsf{ls}(e_1, e_2)$, we get $M \vDash (\Delta \wedge e_1 = e_2) * \mathsf{ls}(e_1, e_2)$, and by weakening we get $M \vDash \Delta * \mathsf{ls}(e_1, e_2)$

(13) **Emp-Ls-R.** This case is symmetric to Case 12

□

Theorem 5.1 (Tri-abduction). *If* $(M, F_1, F_2) \in \mathsf{triab}(P, Q)$, *then* $M \vDash P * F_1$ *and* $M \vDash Q * F_2$

Proof. In our tri-abduction algorithm, we call abduce-par on $P * \mathsf{true}$ and $Q * \mathsf{true}$, so we know based on Lemma D.1 that if $P * \mathsf{true} \lhd [M] \rhd Q * \mathsf{true}$ is derivable, then $M \vDash P * \mathsf{true}$ and $M \vDash Q * \mathsf{true}$ since abduce-par operates by applying the inference in Figure 3. The procedure for finding F_1 and F_2 follows that of Berdine et al. [2005b, §5] and so $M \vDash P * F_1$ and $M \vDash Q * F_2$ by Berdine et al. [2005b, Theorem 7]. □

E SYMBOLIC EXECUTION

E.1 Renaming

We first define the renaming procedure in Algorithm 2, which is identical to that of Calcagno et al. [2011, Fig. 4] except that we additionally require $\vec{e}$ to be disjoint from $\vec{x}$. Renaming produces a new anti-frame M_0 which is similar to M except that it is guaranteed not to mention any program variables and so it trivially meets the side condition of the frame rule. It additionally provides a vector of expressions $\vec{e}$ to be substituted for the free variables in the postcondition $\vec{Y}$ so as to match M_0.

Algorithm 2 $\mathsf{rename}(\Delta, M, Q, \boldsymbol{Q}, \vec{X}, \vec{x})$

Let $\vec{Y}$ be the free logical variables of Q and all the assertions in $\boldsymbol{Q}$.
Pick $\vec{e}$ disjoint from $\vec{Y}$ and $\vec{x}$ such that $\Delta * M \vDash \vec{e} = \vec{Y}$.
Pick M' disjoint from $\vec{X}$, $\vec{Y}$, and Var such that $\Delta * M' \vDash \Delta * M[\vec{e}/\vec{Y}]$.
return $(\vec{e}, \vec{Y}, M')$

Now, we recall the definitions of the following two procedures:

$$\begin{aligned}&\mathsf{biab}'(\exists \vec{Z}.\Delta, Q, \psi, \vec{x}) = \\ &\quad \big\{(M', (\psi \circledast \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]) \\ &\quad \mid (M, F) \in \mathsf{biab}(\Delta, Q) \\ &\qquad (\vec{e}, \vec{Y}, M') = \mathsf{rename}(\Delta, M, Q, \{\psi\}, \vec{Z})\big\}\end{aligned}$$

$$\begin{aligned}&\mathsf{triab}'(P_1, P_2, \psi_1, \psi_2, \vec{x}) = \\ &\quad \big\{(M', (\psi_1 \circledast \exists \vec{X}.F_1[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}], \\ &\qquad (\psi_2 \circledast \exists \vec{X}.F_2[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]) \\ &\quad \mid (M, F_1, F_2) \in \mathsf{triab}(P_1, P_2) \\ &\qquad (\vec{e}, \vec{Y}, M') = \mathsf{rename}(\mathsf{emp}, M, \{\psi_1, \psi_2\}, \emptyset)\big\}\end{aligned}$$

The biab' procedure is similar to AbduceAndAdapt from Calcagno et al. [2011, Fig. 4]. Since the bi-abduction procedure does not support existentially quantified assertions on the left hand side, the existentials must be stripped and then re-added later (as is also done in Calcagno et al. [2011, Algorithm 4]). The renaming step ensures that the anti-frame M' is safe to use with the frame rule. We capture the motivation behind biab' using the following correctness lemma, stating that biab'

produces a suitable frame and anti-frame so as to adapt a specification $\vDash \langle \mathsf{ok} : Q \rangle\ C\ \langle \psi \rangle$ to use a different precondition P.

Lemma E.1. *For all* $(M, \psi') \in \mathsf{biab}'(P, Q, \psi, \vec{x})$, *if* $\vDash \langle \mathsf{ok} : Q \rangle\ C\ \langle \psi \rangle$ *and* $\vec{x} = \mathsf{mod}(C)$, *then*

$$\vDash \langle \mathsf{ok} : P * M \rangle\ C\ \langle \psi' \rangle$$

Proof. By definition, any element of $\mathsf{biab}'(P, Q, \psi, \vec{x})$ (where $P = \exists \vec{Z}.\Delta$) must have the form $(M', (\psi \circledast \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}])$ where $(\vec{e}, \vec{Y}, M') = \mathsf{rename}(\Delta, M, Q, \{\psi\}, \vec{X})$ and $(M, F) \in \mathsf{biab}(\Delta, Q)$. By the definition of rename, we know that:

$$\Delta * M' \vDash \Delta * M[\vec{e}/\vec{Y}]$$

Since M' is assumed to be disjoint from $\vec{Z}$, then $\exists \vec{Z}.M'$ iff M', so we can existentially quantify both sides to obtain:

$$P * M' \vDash \exists \vec{Z}.\Delta * M[\vec{e}/\vec{Y}] \tag{1}$$

In addition, $(M, F) \in \mathsf{biab}(\Delta, Q)$, so we also know that:

$$\Delta * M \vDash Q * F$$

In Figure 5, we assumed all the logical variables used are fresh, so Δ must be disjoint from $\vec{Y}$ (the free variables of Q and ψ), and therefore $\Delta[\vec{e}/\vec{Y}] = \Delta$, so substituting both sides, we get:

$$\Delta * M[\vec{e}/\vec{Y}] \vDash (Q * F)[\vec{e}/\vec{Y}]$$

We also weaken the right hand side by replacing $\vec{x}$ with fresh existentially quantified variables in F.

$$\Delta * M[\vec{e}/\vec{Y}] \vDash (Q * \exists \vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]$$

Now, we can existentially quantify both sides of the entailment. Since logical variables are fresh, $\vec{Z}$ is disjoint from Q.

$$\exists \vec{Z}.\Delta * M[\vec{e}/\vec{Y}] \vDash (Q * \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \tag{2}$$

And finally, we combine Equations (1) and (2) to get:

$$P * M' \vDash (Q * \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \tag{3}$$

Now, given that $\vDash \langle \mathsf{ok} : Q \rangle\ C\ \langle \psi \rangle$, we can use the frame rule to get:

$$\vDash \langle \mathsf{ok} : Q * \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}] \rangle\ C\ \langle \psi \circledast \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}] \rangle$$

This is clearly valid, since $\vec{x} = \mathsf{mod}(C)$ has been removed from the assertion that we are framing in, therefore satisfying $\mathsf{mod}(C) \cap \mathsf{fv}(\exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}]) = \emptyset$. We can also substitute $\vec{e}$ for $\vec{Y}$ in the pre- and postconditions since we assumed that $\vec{e}$ is disjoint from the program variables $\vec{x}$, and therefore $\vec{e}$ remains constant after executing C.

$$\vDash \langle \mathsf{ok} : (Q * \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle\ C\ \langle (\psi \circledast \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle$$

Finally, using the rule of consequence with Equation (3), we strengthen the precondition to get:

$$\vDash \langle \mathsf{ok} : P * M' \rangle\ C\ \langle (\psi \circledast \exists \vec{Z}\vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle$$

□

The triab′ procedure is similar, but it is fundamentally based on tri-abduction and is accordingly used for parallel composition instead of sequential composition. We include two separate proofs corresponding to the two ways in which tri-abduction is using during symbolic execution. The first (Lemma E.2) pertains to merging the anti-frames obtained by continuing to evaluate a single program C after the control flow has already branched whereas the second (Lemma E.3) deals with merging the preconditions from two different program program branches, C_1 and C_2.

Lemma E.2. *If* $(M, \psi_1', \psi_2') \in \mathsf{triab}'(M_1, M_2, \psi_1, \psi_2, \vec{x})$ *and* $\vDash \langle \varphi_1 \circledast M_1 \rangle\ C\ \langle \psi_1 \rangle$ *and* $\vDash \langle \varphi_2 \circledast M_2 \rangle\ C\ \langle \psi_2 \rangle$ *and* $\vec{x} = \mathsf{mod}(C)$*, then* $\vDash \langle \varphi_1 \circledast M \rangle\ C\ \langle \psi_1' \rangle$ *and* $\vDash \langle \varphi_2 \circledast M \rangle\ C\ \langle \psi_2' \rangle$.

Proof. By definition, any element of $\mathsf{triab}'(M_1, M_2, \psi_1, \psi_2, \vec{x})$ will have the form:

$$\left(M', (\psi_1 \circledast \exists \vec{X}.F_1[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}], (\psi_2 \circledast \exists \vec{X}.F_2[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]\right)$$

Where $(\vec{e}, \vec{Y}, M') = \mathsf{rename}(\mathsf{emp}, M, \{\psi_1, \psi_2\}, \emptyset, \vec{x})$ and $(M, F_1, F_2) \in \mathsf{triab}(P_1, P_2)$. From rename, we know that $M' \vDash M[\vec{e}/\vec{Y}]$ and from tri-abduction, we know that $M \vDash M_i * F_i$ for $i = 1, 2$, so $M' \vDash (M_i * F_i)[\vec{e}/\vec{Y}]$. We can weaken this by replacing $\vec{x}$ in F_i with fresh existentially quantified variables to obtain $M' \vDash (M_i * \exists \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]$. By assumption, we know that $\vDash \langle \varphi_i \circledast M_i \rangle\ C\ \langle \psi_i \rangle$ for $i = 1, 2$. So, using the frame rule, we get:

$$\vDash \langle \varphi_i \circledast (M_i * \exists \vec{X}.F_i[\vec{X}/\vec{x}]) \rangle\ C\ \langle \psi_i \circledast \exists \vec{X}.F_i[\vec{X}/\vec{x}] \rangle$$

This is valid since $\exists \vec{X}.F_i[\vec{X}/\vec{x}]$ is disjoint from $\vec{x}$ (the modified program variables) by construction. We also assumed in rename that $\vec{e}$ is disjoint from $\vec{x}$, so we can substitute $\vec{e}$ for $\vec{Y}$ to get:

$$\vDash \langle (\varphi_i \circledast (M_i * \exists \vec{X}.F_i[\vec{X}/\vec{x}]))[\vec{e}/\vec{Y}] \rangle\ C\ \langle (\psi_i \circledast \exists \vec{X}.F_i[\vec{X}/\vec{x}]i)[\vec{e}/\vec{Y}] \rangle$$

Note that $\varphi_i[\vec{e}/\vec{Y}] = \varphi_i$, since the logical variables $\vec{Y}$ are generated freshly, independent of φ_i, as was mentioned in Figure 5. So, using the rule of consequence we get:

$$\vDash \langle \varphi_i \circledast M \rangle\ C\ \langle (\psi_i \circledast \exists \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle$$

□

Lemma E.3. *If* $(M, \psi_1', \psi_2') \in \mathsf{triab}'(P_1, P_2, \psi_1, \psi_2, \vec{x})$ *and* $\vDash \langle \mathsf{ok} : P_1 \rangle\ C_1\ \langle \psi_1 \rangle$ *and* $\vDash \langle \mathsf{ok} : P_2 \rangle\ C_2\ \langle \psi_2 \rangle$ *and* $\vec{x} = \mathsf{mod}(C_1, C_2)$*, then* $\vDash \langle \mathsf{ok} : M \rangle\ C_1\ \langle \psi_1' \rangle$ *and* $\vDash \langle \mathsf{ok} : M \rangle\ C_2\ \langle \psi_2' \rangle$.

Proof. By definition, any element of $\mathsf{triab}'(P_1, P_2, \psi_1, \psi_2, \vec{x})$ will have the form:

$$\left(M', (\psi_1 \circledast \exists \vec{X}.F_1[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}], (\psi_2 \circledast \exists \vec{X}.F_2[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]\right)$$

Where $(\vec{e}, \vec{Y}, M') = \mathsf{rename}(\mathsf{emp}, M, \{\psi_1, \psi_2\}, \emptyset, \vec{x})$ and $(M, F_1, F_2) \in \mathsf{triab}(P_1, P_2)$. From rename, we know that $M' \vDash M[\vec{e}/\vec{Y}]$ and from tri-abduction, we know that $M \vDash P_i * F_i$ for $i = 1, 2$, so $M' \vDash (P_i * F_i)[\vec{e}/\vec{Y}]$. We can weaken this by replacing $\vec{x}$ in F_i with fresh existentially quantified variables to obtain $M' \vDash (P_i * \exists \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]$

By the frame rule, we know that $\vDash \langle \mathsf{ok} : P_i * \exists \vec{X}.F_i[\vec{X}/\vec{x}] \rangle\ C_i\ \langle \psi_i \circledast \vec{X}.F_i[\vec{X}/\vec{x}] \rangle$ since $\vec{X}.F_i[\vec{X}/\vec{x}]$ must be disjoint from the modified program variables $\vec{x}$. By substituting into both the pre and postconditions, we get $\vDash \langle (\mathsf{ok} : P_i * \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle\ C_i\ \langle (\psi_i \circledast \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle$ (this is valid since rename guarantees that $\vec{e}$ is disjoint from $\vec{x}$). Finally, we complete the proof by applying the rule of consequence with $M' \vDash (P_i * \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]$ to obtain:

$$\vDash \langle \mathsf{ok} : M' \rangle\ C_i\ \langle (\psi_i \circledast \vec{X}.F_i[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}] \rangle$$

□

E.2 Sequencing Proof

Lemma 5.3 (Seq). *If* $(M, \psi) \in \mathsf{seq}(\varphi, S, \vec{x})$, $\vec{x} = \mathsf{mod}(C)$, *and* $\vDash \langle \mathsf{ok} : P \rangle\ C\ \langle \vartheta \rangle$ *for all* $(P, \vartheta) \in S$, *then* $\vDash \langle \varphi \circledast M \rangle\ C\ \langle \psi \rangle$.

Proof. By induction on the structure of φ.

- $\varphi = \top$. We need to show that $\vDash \langle \top \circledast \mathsf{emp} \rangle\ C\ \langle \top \rangle$ holds. This triple is clearly valid since any triple with the postcondition $\top$ is trivially true.
- $\varphi = \varphi_1 \bowtie \varphi_2$ where $\bowtie \in \{\vee, \oplus\}$. We need to show:

$$\vDash \langle (\varphi_1 \bowtie \varphi_2) \circledast M \rangle\ C\ \langle \psi_1' \bowtie \psi_2' \rangle$$

  Where $(M, \psi_1', \psi_2') \in \mathsf{triab}'(M_1, M_2, \psi_1, \psi_2, \vec{x})$ and $(M_i, \psi_i) \in \mathsf{seq}(\varphi_i, S, \vec{x})$ for each $i \in \{1, 2\}$. By the induction hypothesis, we know that $\vDash \langle \varphi_i \circledast M_i \rangle\ C\ \langle \psi_i \rangle$, so by Lemma E.2 we get that $\vDash \langle \varphi_i \circledast M \rangle\ C\ \langle \psi_i' \rangle$. We now complete the proof separately for the two logical operators:
  - $\varphi = \varphi_1 \vee \varphi_2$. Suppose that $m \vDash (\varphi_1 \vee \varphi_2) \circledast M$, so $m \vDash \varphi_i \circledast M$ for some $i \in \{1, 2\}$. Since $\vDash \langle \varphi_i \circledast M \rangle\ C\ \langle \psi_i' \rangle$, we know that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash \psi_i'$, and we can weaken this to conclude that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash \psi_1' \vee \psi_2'$.
  - $\varphi = \varphi_1 \oplus \varphi_2$. Suppose that $m \vDash (\varphi_1 \oplus \varphi_2) \circledast M$, and so there are m_1 and m_2 such that $m = m_1 + m_2$ and $m_i \vDash \varphi_i \circledast M$ for each i. Since $\vDash \langle \varphi_i \circledast M \rangle\ C\ \langle \psi_i' \rangle$, we know that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m_i) \vDash \psi_i'$. We also know that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) = [\![C]\!]^{\dagger}_{\mathsf{alloc}}(m_1 + m_2) = [\![C]\!]^{\dagger}_{\mathsf{alloc}}(m_1) + [\![C]\!]^{\dagger}_{\mathsf{alloc}}(m_2)$, and so $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash \psi_1' \oplus \psi_2'$.
- $\varphi = (\varphi')_a$. We need to show that $\vDash \langle (\varphi')_a \circledast M \rangle\ C\ \langle (\psi)_a \rangle$ where $(M, \psi) \in \mathsf{seq}(\varphi', S, \vec{x})$. By the induction hypothesis, we know that $\vDash \langle \varphi' \circledast M \rangle\ C\ \langle \psi \rangle$. Now, suppose that $m \vDash (\varphi')_a \circledast M$. If $a = \mathbb{0}$, then $m = \mathbb{0}$ and therefore $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash (\psi)_{\mathbb{0}}$. If not, then there is some m' such that $m' \vDash \varphi' \circledast M$ and $m = a \cdot m'$. So, $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m') \vDash \psi'$. We also know that $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) = [\![C]\!]^{\dagger}_{\mathsf{alloc}}(a \cdot m') = a \cdot [\![C]\!]^{\dagger}_{\mathsf{alloc}}(m')$, so by the semantics of $(-)_a$, $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) \vDash (\psi)_a$.
- $\varphi = \mathsf{ok} : P$. We need to show that $\vDash \langle \mathsf{ok} : P * M \rangle\ C\ \langle \psi' \rangle$ where $(M, \psi') \in \mathsf{biab}'(P, Q, \psi, \vec{x})$ and $(Q, \psi) \in S$. By assumption, we know that $\vDash \langle \mathsf{ok} : Q \rangle\ C\ \langle \psi \rangle$. The remainder of the proof follows directly from Lemma E.1.
- $\varphi = \mathsf{er} : Q$. We need to show that $\vDash \langle \mathsf{er} : Q \rangle\ C\ \langle \mathsf{er} : Q \rangle$. This trivially holds since any m satisfying $\mathsf{er} : Q$ must consist only of $\mathbb{1}_{\mathsf{er}}(s, h)$ states, and so $[\![C]\!]^{\dagger}_{\mathsf{alloc}}(m) = m$.

□

E.3 Symbolic Execution Proofs

Lemma E.4. *Let:*

$$f(S) = \begin{array}{l} \{(\neg e \wedge \mathsf{emp}, \mathsf{ok} : \neg e \wedge \mathsf{emp})\} \cup \\ \{(M_1 * M_2 \wedge e, \psi) \mid (M_1, \varphi) \in \mathsf{seq}(\mathsf{ok} : e \wedge \mathsf{emp}, [\![C]\!]^{\sharp}(T), \mathsf{mod}(C)), (M_2, \psi) \in \mathsf{seq}(\varphi, S, \mathsf{mod}(C))\} \end{array}$$

For any $n \in \mathbb{N}$ *and* $(P, \varphi) \in f^n(\emptyset)$, $\vDash \langle \mathsf{ok} : P \rangle$ while e do $C\ \langle \varphi \rangle$.

Proof. By induction on n. Suppose $n = 0$, then $f^0(\emptyset) = \emptyset$, so the claim vacuously holds. Now, suppose the claim holds for n, we will show it holds for $n + 1$. First, observe that:

$$\begin{aligned} f^{n+1}(\emptyset) &= f(f^n(\emptyset)) \\ &= \{(\neg e \wedge \mathsf{emp}, \mathsf{ok} : \neg e \wedge \mathsf{emp})\} \cup \\ &\quad\ \{(M_1 * M_2 \wedge e, \psi) \mid (M_1, \varphi) \in \mathsf{seq}(\mathsf{ok} : e \wedge \mathsf{emp}, [\![C]\!]^{\sharp}(T), \mathsf{mod}(C)) \\ &\qquad\qquad\qquad\qquad\qquad (M_2, \psi) \in \mathsf{seq}(\varphi, f^n(\emptyset), \mathsf{mod}(C))\} \end{aligned}$$

So any $(P, \varphi) \in f^{n+1}(\emptyset)$ comes from one of the two sets in the above union. Suppose it is in the first, so we need to show that $\vDash \langle \mathsf{ok} : \neg e \wedge \mathsf{emp} \rangle$ while e do C $\langle \mathsf{ok} : \neg e \wedge \mathsf{emp} \rangle$. This is clearly true, since the loop does not execute in states where $\neg e$ holds and therefore the whole command is equivalent to skip.

Now suppose we are in the second case, so the element has the form $(M_1 * M_2 \wedge e, \psi)$ where $(M_1, \varphi) \in \mathsf{seq}(\mathsf{ok} : e \wedge \mathsf{emp}, \llbracket C \rrbracket^\sharp (T), \mathsf{mod}(C))$ and $(M_2, \psi) \in \mathsf{seq}(\varphi, f^n(\emptyset), \mathsf{mod}(C))$. By Lemma C.5, we know that $\vDash \langle \mathsf{ok} : M_1 \wedge e \rangle\ C\ \langle \varphi \rangle$ and by Lemma C.5 and the induction hypothesis, we get $\vDash \langle \varphi \circledast M_2 \rangle$ while e do C $\langle \psi \rangle$. Using the frame rule, we also get $\vDash \langle \mathsf{ok} : M_1 * M_2 \wedge e \rangle\ C\ \langle \varphi \circledast M_2 \rangle$, and so we can sequence the previous specifications to get $\vDash \langle \mathsf{ok} : M_1 * M_2 \wedge e \rangle\ C \fatsemi$ while e do C $\langle \psi \rangle$. Now, since the precondition stipulates that e is true, the loop must run for at least one iteration, so for any $m \vDash (\mathsf{ok} : M_1 * M_2 \wedge e)$, $\llbracket C \fatsemi \text{while } e \text{ do } C \rrbracket^\dagger_{\mathsf{alloc}}(m) = \llbracket \text{while } e \text{ do } C \rrbracket^\dagger_{\mathsf{alloc}}(m)$, and so $\vDash \langle \mathsf{ok} : M_1 * M_2 \wedge e \rangle$ while e do C $\langle \psi \rangle$. □

Lemma E.5. *If for every $(s, h) \vDash P$ and $\mathsf{alloc} \in \mathsf{Alloc}$, there exists s' and t' such that $\llbracket C \rrbracket_{\mathsf{alloc}}(s, h) = \mathsf{unit}_{\mathcal{W}}(\mathbb{i}_\epsilon(s', t'))$ and $(s', h') \vDash Q$, then $\vDash \langle \mathsf{ok} : P \rangle\ C\ \langle \epsilon : Q \rangle$*

Proof. Suppose that $m \vDash \mathsf{ok} : P$. That means that $|m| = \mathbb{1}$ and all elements of $\mathsf{supp}(m)$ have the form $\mathbb{i}_{\mathsf{ok}}(s, h)$ where $(s, h) \vDash P$. By assumption, we know that $\llbracket C \rrbracket_{\mathsf{alloc}}(s, h) = \mathsf{unit}_{\mathcal{W}}(\mathbb{i}_\epsilon(s', t'))$ such that $(s', t') \vDash Q$. This means that every element of $\llbracket C \rrbracket^\dagger_{\mathsf{alloc}}(m)$ must have the form $\mathbb{i}_\epsilon(s', t')$ where $(s', t') \vDash Q$ and also $|\llbracket C \rrbracket^\dagger_{\mathsf{alloc}}(m)| = \mathbb{1}$ since each (s, h) does not change the mass of the distribution, so $\llbracket C \rrbracket^\dagger_{\mathsf{alloc}}(m) \vDash \epsilon : Q$. □

Theorem 5.2 (Symbolic Execution Soundness). *If $(P, \varphi) \in \llbracket C \rrbracket^\sharp (T)$, then $\vDash \langle \mathsf{ok} : P \rangle\ C\ \langle \varphi \rangle$*

Proof. By induction on the structure of the program C.

- $C = \mathsf{skip}$. We need to show that $\vDash \langle \mathsf{ok} : \mathsf{emp} \rangle$ skip $\langle \mathsf{ok} : \mathsf{emp} \rangle$, which is trivially true.
- $C = C_1 \fatsemi C_2$. By definition, any element of $\llbracket C_1 \fatsemi C_2 \rrbracket^\sharp (T)$ must have the form $(P * M, \psi)$ where $(P, \varphi) \in \llbracket C_1 \rrbracket^\sharp (T)$ and $(M, \psi) \in \mathsf{seq}(\varphi, \llbracket C_2 \rrbracket^\sharp (T), \mathsf{mod}(C_2))$. By the induction hypothesis, we know that $\vDash \langle \mathsf{ok} : P \rangle\ C_1\ \langle \varphi \rangle$ and by Lemma C.5 we know that $\vDash \langle \varphi \circledast M \rangle\ C_2\ \langle \psi \rangle$. Using the frame rule, we get that $\vDash \langle \mathsf{ok} : P * M \rangle\ C_1\ \langle \varphi \circledast M \rangle$ (given the renaming step used in seq, M contains no program variables, so it must obey the side condition of the frame rule). Finally, we can join the two specifications to conclude that $\vDash \langle \mathsf{ok} : P * M \rangle\ C_1 \fatsemi C_2\ \langle \psi \rangle$.
- $C = C_1 + C_2$. Any element of $\llbracket C_1 + C_2 \rrbracket^\sharp (T)$ must have the form $(M, \psi_1' \oplus \psi_2')$ where $(M, \psi_1', \psi_2') \in \mathsf{triab}'(M_1, M_2, \psi_1, \psi_2, \mathsf{mod}(C_1, C_2))$ and $(M_1, \psi_1) \in \llbracket C_1 \rrbracket^\sharp (T)$ and $(M_2, \psi_2) \in \llbracket C_2 \rrbracket^\sharp (T)$. By the induction hypothesis, we know that $\vDash \langle \mathsf{ok} : M_i \rangle\ C_i\ \langle \psi_i \rangle$ for $i = 1, 2$. By Lemma E.3 we know that $\vDash \langle \mathsf{ok} : M \rangle\ C_i\ \langle \psi_i' \rangle$. Now, we show that $\vDash \langle M \rangle\ C_1 + C_2\ \langle \psi_1' \oplus \psi_2' \rangle$: suppose $m \vDash M$. By definition, $\llbracket C_1 + C_2 \rrbracket^\dagger_{\mathsf{alloc}}(m) = \llbracket C_1 \rrbracket^\dagger_{\mathsf{alloc}}(m) + \llbracket C_2 \rrbracket^\dagger_{\mathsf{alloc}}(m)$. Now, using what we obtained from Lemma E.3, we know that since $m \vDash M$, $\llbracket C_i \rrbracket^\dagger_{\mathsf{alloc}}(m) \vDash \psi_i'$ for each $i \in \{1, 2\}$. Combining these two, we get that $\llbracket C_1 \rrbracket^\dagger_{\mathsf{alloc}}(m) + \llbracket C_2 \rrbracket^\dagger_{\mathsf{alloc}}(m) \vDash \psi_1' \oplus \psi_2'$.
- $C = \mathsf{assume}\ b$. Any element of $\llbracket \mathsf{assume}\ b \rrbracket^\sharp (T)$ must have the form $(b \wedge \mathsf{emp}, \mathsf{ok} : b \wedge \mathsf{emp})$ or $(\neg b \wedge \mathsf{emp}, (\top)_{\mathbb{0}})$. In the first case, we need to show $\vDash \langle \mathsf{ok} : b \wedge \mathsf{emp} \rangle$ assume b $\langle \mathsf{ok} : b \wedge \mathsf{emp} \rangle$. Suppose $m \vDash \mathsf{ok} : b \wedge \mathsf{emp}$, then it's easy to see that $\llbracket \mathsf{assume}\ b \rrbracket^\dagger_{\mathsf{alloc}}(m) = m$, so the triple is valid. In the second case, we must show $\vDash \langle \mathsf{ok} : \neg b \wedge \mathsf{emp} \rangle$ assume b $\langle (\top)_{\mathbb{0}} \rangle$. Suppose $m \vDash \mathsf{ok} : \neg b \wedge \mathsf{emp}$, so $\llbracket \mathsf{assume}\ b \rrbracket^\dagger_{\mathsf{alloc}}(m) = \mathbb{0}$, and $\mathbb{0} \vDash (\top)_{\mathbb{0}}$.
- $C = \mathsf{assume}\ a$. Any element of $\llbracket \mathsf{assume}\ a \rrbracket^\sharp (T)$ must have the form $(\mathsf{emp}, (\mathsf{ok} : \mathsf{emp})_a)$, so we need to show $\vDash \langle \mathsf{ok} : \mathsf{emp} \rangle$ assume a $\langle (\mathsf{ok} : \mathsf{emp})_a \rangle$. Suppose $m \vDash \mathsf{ok} : \mathsf{emp}$, so we know that $\llbracket \mathsf{assume}\ a \rrbracket^\dagger_{\mathsf{alloc}}(m) = a \cdot m$, therefore $\llbracket \mathsf{assume}\ a \rrbracket^\dagger_{\mathsf{alloc}}(m) \vDash (\mathsf{ok} : \mathsf{emp})_a$.

- $C = \text{while } e \text{ do } C$. By the Kleene fixed point theorem, $[\![\text{while } e \text{ do } C]\!]^\sharp(T) = \bigcup_{n\in\mathbb{N}} f^n(\emptyset)$ where $f(S)$ is defined as in Lemma E.4. So, any $(P,\varphi) \in [\![\text{while } e \text{ do } C]\!]^\sharp(T)$ must also be an element of $f^n(\emptyset)$ for some n. We complete the proof by applying Lemma E.4.

The remaining cases are for primitive instructions, most of which are *pure*, meaning that each program state maps to a single outcome according to the program semantics. In these cases, it suffices to show that if $(P,\varphi) \in [\![c]\!]^\sharp(T)$, then $[\![c]\!]_{\mathsf{alloc}}(s,h) \vDash \varphi$ for all $(s,h) \vDash P$ by Lemma E.5.

- $C = (x := e)$. Suppose that $(s,h) \vDash \mathsf{ok} : x = X \land \mathsf{emp}$, so $s(x) = s(X)$ and $h = \emptyset$. Now, $[\![x := e]\!]_{\mathsf{alloc}}(s,h) = \mathsf{unit}(s[x \mapsto [\![e]\!](s)], h)$, so let $s' = s[x \mapsto [\![e]\!](s)]$. Clearly, $[\![e]\!](s) = [\![e[X/x]]\!](s)$ since $s(x) = s(X)$. It must also be the case that $[\![e[X/x]]\!](s) = [\![e[X/x]]\!](s')$ since s and s' differ only in the values of x, and x does not appear in $e[X/x]$. So, $s'(x) = [\![e]\!](s) = [\![e[X/x]]\!](s) = [\![e[X/x]]\!](s')$, and therefore $(s',h) \vDash \mathsf{ok} : x = e[X/x] \land \mathsf{emp}$. The remainder of the proof follows by Lemma E.5.
- $C = (x := \mathsf{alloc}())$. We need to show that $\vDash \langle \mathsf{ok} : \mathsf{emp} \land x = X \rangle\ x := \mathsf{alloc}()\ \langle \mathsf{ok} : \exists Y. x \mapsto Y \rangle$. Suppose that $m \vDash \mathsf{ok} : \mathsf{emp} \land x = X$, so each state in m has the form $\mathbb{i}_{\mathsf{ok}}(s,h)$ where $(s,h) \vDash \mathsf{emp} \land x = X$, so $s(x) = s(X)$ and $h = \emptyset$. We know that $[\![x := \mathsf{alloc}]\!]_{\mathsf{alloc}}(s,h) = \mathsf{bind}_{\mathcal{W}}(\mathsf{alloc}(s,h), \lambda(\ell,v).\mathsf{unit}(s[x \mapsto \ell], h[\ell \mapsto v]))$ where $\ell \notin \mathsf{dom}(h)$. Let $s' = s[x \mapsto \ell]$ and $h' = h[\ell \mapsto v]$. Clearly, $h'([\![x]\!](s')) = h'(\ell) = v$, so $(s',h') \vDash \exists Y. x \mapsto Y$. Since this is true for all end states, and since alloc does not alter the total mass of the distribution, then $[\![x := \mathsf{alloc}()]\!]^\dagger_{\mathsf{alloc}}(m) \vDash \mathsf{ok} : \exists Y. x \mapsto Y$.
- $C = \mathsf{free}(x)$. There are three cases since specifications for free can take on multiple forms. In the first case, we need to show that $\vDash \langle \mathsf{ok} : e \mapsto X \rangle\ \mathsf{free}(e)\ \langle \mathsf{ok} : e \not\mapsto \rangle$. Suppose $(s,h) \vDash e \mapsto X$, so $h([\![e]\!](s)) = s(X)$. We also know that $[\![\mathsf{free}(e)]\!]_{\mathsf{alloc}}(s,h) = \mathsf{unit}(s, h[[\![e]\!](s) \mapsto \bot])$, and since $(s, h[[\![e]\!](s) \mapsto \bot]) \vDash e \not\mapsto$, the claim follows by Lemma E.5.
In the other cases, we need to show that:
$$\vDash \langle \mathsf{ok} : e \not\mapsto \rangle\ \mathsf{free}(e)\ \langle \mathsf{er} : e \not\mapsto \rangle \quad \text{and} \quad \vDash \langle \mathsf{ok} : e = \mathsf{null} \rangle\ \mathsf{free}(e)\ \langle \mathsf{er} : e = \mathsf{null} \rangle$$
Suppose $(s,h) \vDash e \not\mapsto$ and so $h([\![e]\!](s)) = \bot$. Clearly, $[\![\mathsf{free}(e)]\!]_{\mathsf{alloc}}(s,h) = \mathsf{error}(s,h)$, so by Lemma E.5 the claim holds. The case where $e = \mathsf{null}$ is nearly identical.
- $C = [e_1] \leftarrow e_2$. There are three cases, first we must show that $\vDash \langle \mathsf{ok} : e_1 \mapsto X \rangle\ [e_1] \leftarrow e_2\ \langle \mathsf{ok} : e_1 \mapsto e_2 \rangle$. Suppose $(s,h) \vDash e_1 \mapsto X$, so $h([\![e_1]\!](s)) = s(X)$ and therefore that memory address is allocated since $s(X) \in \mathsf{Val}$. This means that $[\![[e_1] \leftarrow e_2]\!]_{\mathsf{alloc}}(s,h) = \mathsf{unit}(s, h[[\![e_1]\!](s) \mapsto [\![e_2]\!](s)])$ and clearly $(s, h[[\![e_1]\!](s) \mapsto [\![e_2]\!](s)]) \vDash e_1 \mapsto e_2$ by definition, so the claim holds by Lemma E.5.
In the remaining case, we must show that $\vDash \langle \mathsf{ok} : e_1 \not\mapsto \rangle\ [e_1] \leftarrow e_2\ \langle \mathsf{er} : e_1 \not\mapsto \rangle$ and $\vDash \langle \mathsf{ok} : e_1 = \mathsf{null} \rangle\ [e_1] \leftarrow e_2\ \langle \mathsf{er} : e_1 = \mathsf{null} \rangle$. The proof is similar to the second case for $\mathsf{free}(e)$.
- $C = x \leftarrow [e]$. There are three cases, first we must show that $\vDash \langle \mathsf{ok} : x = X \land e \mapsto Y \rangle\ x \leftarrow [e]\ \langle \mathsf{ok} : x = Y \land e[X/x] \mapsto Y \rangle$. Suppose that $(s,h) \vDash x = X \land e \mapsto Y$, so $s(x) = s(X)$ and $h([\![e]\!](s)) = s(Y)$. We also know that $[\![x \leftarrow [e]]\!]_{\mathsf{alloc}}(s,h) = \mathsf{unit}(s[x \mapsto h([\![e]\!](s))], h)$. Let $s' = s[x \mapsto h([\![e]\!](s))]$, as we showed in the $x := e$ case, $[\![e]\!](s) = [\![e[X/x]]\!](s) = [\![e[X/x]]\!](s')$. So, this means that $h([\![e[X/x]]\!](s')) = h([\![e]\!](s)) = s(Y)$ and $s'(x) = [\![e]\!](s) = s(Y)$, so clearly $(s',h) \vDash x = Y \land e[X/x] \mapsto Y$ and the claim follows from Lemma E.5.
In the remaining case, we need to show that $\vDash \langle \mathsf{ok} : e \not\mapsto \rangle\ x \leftarrow [e]\ \langle \mathsf{er} : e \not\mapsto \rangle$ and $\vDash \langle \mathsf{ok} : e = \mathsf{null} \rangle\ x \leftarrow [e]\ \langle \mathsf{er} : e = \mathsf{null} \rangle$. The proof is similar to the second case for $\mathsf{free}(e)$.
- $C = \mathsf{error}()$. We need to show that $\vDash \langle \mathsf{ok} : \mathsf{emp} \rangle\ \mathsf{error}()\ \langle \mathsf{er} : \mathsf{emp} \rangle$. Suppose $(s,h) \vDash \mathsf{emp}$. Clearly, $[\![\mathsf{error}()]\!]_{\mathsf{alloc}}(s,h) = \mathsf{error}(s,h)$, and so the claim holds by Lemma E.5.

- $C = f(\vec{e})$. Any element of $[\![f(\vec{e})]\!]^\sharp(T)$ must have the form $(P \land \vec{x} = \vec{X})$ where $(P, \varphi) \in \mathsf{seq}(\mathsf{ok} : \vec{x} = \vec{e}[X/x], T(f(\vec{x})), \mathsf{mod}(f))$. By Lemma C.5, we get:
$$\vDash \langle \mathsf{ok} : P \land \vec{x} = \vec{e}[X/x] \rangle\ f(\vec{x})\ \langle \varphi \rangle$$
Now, we need to show that $\vDash \langle \mathsf{ok} : P \land \vec{x} = \vec{X} \rangle\ f(\vec{e})\ \langle \varphi \rangle$. Suppose that $m \vDash \mathsf{ok} : P \land \vec{x} = \vec{X}$. Let m' be obtained by taking every state $\mathbb{i}_{\mathsf{ok}}(s, h) \in \mathsf{supp}(m)$ and modifying it to be $\mathbb{i}_{\mathsf{ok}}(s[\vec{x} \mapsto [\![\vec{e}]\!](s)], h)$. By a similar argument to the $x := e$ case, we know that $m' \vDash \mathsf{ok} : \vec{x} = \vec{e}[\vec{X}/\vec{x}]$. We know P is disjoint from the program variables by the definition of seq, so $m' \vDash \mathsf{ok} : P$ as well, since $m \vDash \mathsf{ok} : P$ and the only difference between m and m' is updates to the program variables. Let C be the body of f and note that $[\![f(\vec{x})]\!]^\dagger_{\mathsf{alloc}}(m') = [\![C]\!]^\dagger_{\mathsf{alloc}}(m')$ since the initial modification of the program state is just updating variable values to themselves. We know from $\vDash \langle \mathsf{ok} : P \land \vec{x} = \vec{e}[X/x] \rangle\ f(\vec{x})\ \langle \varphi \rangle$ that $[\![C]\!]^\dagger_{\mathsf{alloc}}(m') \vDash \varphi$ and by definition, $[\![f(\vec{e})]\!]^\dagger_{\mathsf{alloc}}(m) = [\![C]\!]^\dagger_{\mathsf{alloc}}(m')$, so $[\![f(\vec{e})]\!]^\dagger_{\mathsf{alloc}}(m) \vDash \varphi$.

In addition, we show that the two refinements for single-path computation and loops invariants are sound too:

- Single Path. Any element of $[\![C_1 + C_2]\!]^\sharp(T)$ has one of two forms. In the first case, we need to show that $\vDash \langle \mathsf{ok} : P \rangle\ C_1 + C_2\ \langle \varphi \oplus \top \rangle$ given that $\vDash \langle \mathsf{ok} : P \rangle\ C_1\ \langle \varphi \rangle$. Suppose that $m \vDash \mathsf{ok} : P$. By our assumption, we know that $[\![C_1]\!]^\dagger_{\mathsf{alloc}}(m) \vDash \varphi$. Now, $[\![C_1 + C_2]\!]^\dagger_{\mathsf{alloc}}(m) = [\![C_1]\!]^\dagger_{\mathsf{alloc}}(m) + [\![C_2]\!]^\dagger_{\mathsf{alloc}}(m)$ and clearly $[\![C_2]\!]^\dagger_{\mathsf{alloc}}(m) \vDash \top$, so $[\![C_1 + C_2]\!]^\dagger_{\mathsf{alloc}}(m) \vDash \varphi \oplus \top$. The second case is symmetrical, using the fact that $+$ is commutative.
- Loop invariants. We need to show that $\vDash \langle \mathsf{ok} : I \rangle$ while e do C $\langle (\mathsf{ok} : I \land \neg e) \lor (\top)_{\mathbb{0}} \rangle$ given that $\vDash \langle \mathsf{ok} : I \land e \rangle\ C\ \langle \mathsf{ok} : I \rangle$. Note that this case is only valid for deterministic or nondeterministic programs (not probabilistic ones). Suppose $m \vDash \mathsf{ok} : I$, so every state in $\mathsf{supp}(m)$ has the form $\mathbb{i}_{\mathsf{ok}}(s, h)$ where $(s, h) \vDash I$. By assumption, we know that every execution of the loop body will preserve the truth of I, so either all the states in $[\![\text{while } e \text{ do } C]\!]_{\mathsf{alloc}}(s, h)$ must satisfy I and $\neg e$, or there are no terminating states. In other words, $[\![\text{while } e \text{ do } C]\!]_{\mathsf{alloc}}(s, h) \vDash (\mathsf{ok} : I \land \neg e) \lor (\top)_{\mathbb{0}}$. In the deterministic case, we are done since there can only be a single start state. In the nondeterministic case, each start state (s, h) leads to a set of end states satisfying $(\mathsf{ok} : I \land \neg e) \lor (\top)_{\mathbb{0}}$, then the union of all these states will also satisfy $(\mathsf{ok} : I \land \neg e) \lor (\top)_{\mathbb{0}}$.

□