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 (e.g., randomization) and, orthogonally, static analysers must handle incorrectness too. We present Outcome Separation Logic (OSL), a program logic that is sound for both correctness and incorrectness reasoning with varying effects. OSL has a frame rule—just like separation logic—but uses different underlying assumptions that lift restrictions imposed by SL, which precluded reasoning about incorrectness and effects.

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 to 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 2019] 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 separation logic relies on the following three assumptions, limiting its applicability [Yang and O’Hearn 2002].

**Nondeterminism.** Separation logic requires programs to be nondeterministic. As such, the promise of local reasoning and, by extension, scalable program analysis cannot immediately extend to alternative execution models such as probabilistic computation.

**Must properties.** Separation logic can only express properties that must occur, not ones that may occur, making it inept for incorrectness since many bugs only occur some of the time.

**Safe preconditions.** Separation logic preconditions ensure that all program paths do not fault, so analyses must examine the entire program even if a bug occurs only in certain traces.
In this paper, we show that local reasoning is sound under different assumptions, which do not force any particular evaluation model and are compatible with incorrectness reasoning. To that end, we introduce Outcome Separation Logic (OSL), a single program logic for locally reasoning about both correctness and incorrectness with varied computational effects.

OSL builds on Outcome Logic (OL), which supports correctness and incorrectness reasoning with effects [Zilberstein et al. 2023], but not local reasoning. Using OSL as a logical foundation, we present bi-abduction-style symbolic execution algorithms that infer both correctness and incorrectness specifications using a shared set of procedure summaries. Our contributions are as follows.

**Outcome Separation Logic.** While Zilberstein et al. [2023] embedded separation logic in OL, we go further by making heap assertions a first-class element of OL. This culminates in a frame rule that resembles that of standard separation logic, but relies on different underlying assumptions for its soundness, allowing us to lift the three aforementioned restrictions that make SL unsuitable for reasoning about effects and incorrectness. The OSL semantics is based on an algebraic representation of choice; we show how this can encode deterministic, nondeterministic, and probabilistic programs.

**Tri-abduction for parallel composition.** Bi-abduction enables scalable reasoning for sequential programs by reconciling the postcondition of one precomputed spec with the precondition of the next. However, programs with effects are not purely sequential, but rather have control flow branching that arises from, e.g., nondeterministic or probabilistic choice. Calcagno et al. [2009, 2011] handle nondeterminism by generating candidate preconditions for each program trace, and then re-evaluating the program to check whether each one is valid for all paths. This approach has two downsides: it misses some valid preconditions and involves extra work to re-check the ones it does find. To solve this problem, we introduce the parallel composition analogue of bi-abduction, which we call tri-abduction because it infers three assertions rather than two (a precondition, and a leftover frame for each of the two effectful branches).

**Symbolic execution algorithms.** Building on the previous two contributions, we present symbolic execution algorithms to analyze C-like pointer programs. The core algorithm finds all the reachable outcomes, and is therefore capable of reasoning about both correctness and incorrectness.

We also define a single-path variant, inferring specifications in which the postcondition is just one of the (possibly many) outcomes. It is similar to bug-finding algorithms based on Incorrectness Logic (Pulse [Raad et al. 2020] and Pulse-X [Le et al. 2022]) in its ability to drop paths for increased scalability, but with the added benefit that it can soundly re-use procedure summaries generated by the correctness algorithm so as to not re-compute summaries for the same procedure.

We begin in Section 2 by outlining how the assumptions of separation logic prevent reasoning about arbitrary effects and incorrectness, and how we lift those assumptions in OSL. 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 tri-abduction, which is inspired by bi-abduction but is used for parallel rather than sequential composition. Tri-abduction does not replace bi-abduction, we use both together in Section 6 to define symbolic execution algorithms. In Section 7 we examine two case studies to show the applicability of these algorithms and finally we conclude in Sections 8 and 9 by discussing related work and next steps.

2 KEY IDEAS: LOCAL REASONING FOR MORE TYPES OF PROGRAMS

We begin by examining how the local reasoning principles of separation logic, along with bi-abductive inference, underly 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.

\[
\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\} \quad \text{SEQUENCE}}{\{P\} C_1 ; C_2 \{R\}}
\]

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 (spacial) compositionality via the FRAME rule, which adds information about unused program resources \(F\) to the pre- and postcondition of a completed proof.

\[
\frac{\{P\} C \{Q\} \quad \{P \ast F\} C \{Q \ast F\} \quad \text{FRAME}}{\{P \ast F\} \ast F C \{Q \ast F\} \ast F \}
\]

But framing does not immediately provide a way to compose specifications. Given \(\{P_1\} C_1 \{Q_1\}\) and \(\{P_2\} C_2 \{Q_2\}\), it is not clear what—if anything—we can add to make \(Q_1\) match \(P_2\). This is where bi-abduction comes in—a technique that finds a missing resource \(M\) and a leftover frame \(F\) to make the entailment \(Q_1 \ast M \models P_2 \ast F\) hold. With the help of bi-abduction, we get a more usable sequence rule that stitches together two precomputed summaries without reexamining either piece of code.

\[
\frac{\{P_1\} C_1 \{Q_1\} \quad Q_1 \ast M \models P_2 \ast F \quad \{P_2\} C_2 \{Q_2\} \quad \text{BI-ABDUCTIVE SEQUENCE}}{\{P_1 \ast M\} C_1 ; C_2 \{Q_2 \ast F\}}
\]

While bi-abduction has enabled industrial strength static analyzers to scale to massive codebases, it does not support programs with effects such as probabilistic choice. In the remainder of this section, we will examine why this is the case, and then explain how our logic extends bi-abductive analysis to more programs (e.g., ones with computational effects) and program properties (e.g., reachability and incorrectness) not supported by current bi-abductive analyzers.

### 2.1 Interlude: Reasoning about Effects and Incorrectness

Computational effects have intricate interactions with incorrectness reasoning. Identifying bugs in a pure program is not so hard, as demonstrated by the following example in which the program crashes because it attempts to write into a null pointer \(x\).

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

The situation becomes more complicated once effects are involved.\(^1\) 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; now \(x\) is obtained using malloc, which nondeterministically either returns a valid pointer or null. In Hoare Logic, the best we can do is specify this program using a disjunction.

\[
\{\text{ok} : \text{emp}\} \ x := \text{malloc}() ; [x] \leftarrow 1 \ \{(\text{ok} : x \mapsto 1) \lor (\text{er} : x = \text{null})\}
\]

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 2019]. Specifying the

\(^1\) Even nontermination is an effect, so looping programs must be handled delicately when it comes to incorrectness reasoning.
Noam Zilberstein, Angelina Saliling, and Alexandra Silva

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

\[ \text{ok : emp} \ x := \text{malloc()} \mathbin{;} [x] \leftarrow 1 \ ([(\text{ok} : x \mapsto 1) \lor (\text{er} : x = \text{null})] \]

IL has a frame rule [Raad et al. 2020] and can underly 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.

\[ \langle \text{ok : emp} \rangle \ x := \text{malloc()} \mathbin{;} [x] \leftarrow 1 \ (\langle \text{ok} : x \mapsto 1 \rangle \oplus \langle \text{er} : x = \text{null} \rangle) \]

The above triple stipulates 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 \( S_1 \models (\text{ok} : x \mapsto 1) \) and \( S_2 \not\models (\text{er} : x = \text{null}) \). 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 2019; 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.

\[ \langle \text{ok : emp} \rangle \ x := \text{malloc()} \mathbin{;} [x] \leftarrow 1 \ (\langle \text{er} : x = \text{null} \rangle \oplus \top) \]

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.

\[ \langle \text{ok : true} \rangle \ x := \text{ping}(192.0.2.1) \ (\langle \text{ok} : x = 0 \rangle \oplus_{99\%} (\text{er} : x = 1)) \]

Our goal in this paper is to augment Outcome Logic with a sound frame rule, and to use the resulting theory to build bi-abductive symbolic execution algorithms both for correctness (finding all reachable outcomes) and incorrectness (only exploring one program path at a time). We will next see why the frame rule of separation logic is not suitable for reasoning about arbitrary effects and incorrectness before exploring our new solution based on Outcome Separation Logic.

### 2.2 Overcoming the Restrictions of the Frame Rule

Local reasoning does not come for free; separation logic imposes several restrictions that preclude reasoning about computational effects, reachability, and dropping program paths. We now walk through those restrictions, and explain how we relax them in Outcome Separation Logic.

**Nondeterminism.** Separation logic requires that the semantics of memory allocation be nondeterministic. To see what goes wrong without nondeterminism, let us consider a semantics in which the memory allocator always deterministically picks the next available heap address. The following...
specification (left) is then valid; if \( x \) is allocated in the empty heap, then it must be given the first address, \( i.e., x = 1 \). But, using the frame rule we can infer an invalid specification (right).

\[
\begin{align*}
\{\text{emp}\} \ x := \text{alloc()} \ {\{x = 1 \land x \mapsto \_\}} & \quad \{\text{emp}\} \ x := \text{alloc()} \ {\{x = 1 \land x \mapsto \_ \}} \quad \text{\textit{Frame}} \\
\{y \mapsto 1\} \ x := \text{alloc()} \ {\{x = 1 \land x \mapsto \_ \\* y \mapsto 1\}} & \quad \{\text{emp}\} \ x := \text{alloc()} \ {\{x = 1 \land x \mapsto \_\}}
\end{align*}
\]

It is possible that \( y \) has address 1, in which case \( x \) (a freshly allocated pointer) cannot also be equal to 1. So, this application of the frame rule is clearly unsound.

In response, separation logic forces memory allocation to be nondeterministic [Yang and O’Hearn 2002]—in the above program, \( x \) could be assigned \emph{any} address that is not already allocated. Effectively, this means that the postcondition cannot say anything specific about the address of \( x \); we cannot conclude that \( x = 1 \), but rather we could only conclude that \( x = 1 \lor x = 2 \lor \cdots \), which remains true after framing in \( y \mapsto 1 \). Nondeterminism has a delicate interaction with the next restriction; we will show how both can be relaxed together.

**Must properties.** If we concede for a moment that memory allocation will be nondeterministic and focus just on the unification of verification and bug-finding, separation logic \emph{still} imposes undesirable constraints. The soundness of framing also relies on the fact that Hoare Logic postconditions are \emph{must} properties (they describe outcomes that must occur), and not \emph{may} properties (that sometimes occur). Outcome Logic can express may properties, as is required for incorrectness. In fact, the bug that we saw in Section 2.1 required a may property.

The problematic interaction between framing and reachability is displayed in the following example, where the premise explicitly states that \( x = 1 \) is a reachable outcome of allocating \( x \) in the empty heap (note that allocation is still assumed to be nondeterministic).

\[
\begin{align*}
\langle \text{ok} : \text{emp}\rangle \ x := \text{alloc()} \ \langle (\text{ok} : x = 1) \oplus (\text{ok} : x \neq 1) \rangle & \quad \langle \text{ok} : y \mapsto 1\rangle \ x := \text{alloc()} \ \langle (\text{ok} : x = 1 \land y \mapsto 1) \oplus (\text{ok} : x \neq 1 \land y \mapsto 1) \rangle \quad \text{\textit{Frame}}
\end{align*}
\]

This inference is invalid; there are states satisfying the precondition in which \( y \) has the address 1, in which case the outcome where \( x = 1 \) is no longer reachable.

In summary, it is the combined interaction between nondeterministic memory allocation and the semantics of Hoare Logic that makes framing sound. However, this delicate interplay is ensuring a more direct property about local reasoning: assertions cannot be too specific about the \emph{addresses} of pointers, since those addresses may change after framing. In fact, Yang and O’Hearn [2002] already postulated that these restrictions could be dropped if assertions were invariant under address renaming and Baktiev [2006] subsequently proved exactly that for a deterministic separation logic.

We take a similar approach in OSL by proving that the \emph{symbolic heaps} that are typically used in symbolic execution algorithms [Berdine et al. 2005a,b; Calcagno et al. 2009, 2011] are invariant under renaming (Section 4.2). More specifically, if \( (s, h) \models P \), then \( (\pi(s), \pi(h)) \models P \) where \( \pi \) permutes the heap addresses via some bijection. Our underlying program semantics is parametric on an allocator, showing that our frame rule is sound for \emph{any} allocation semantics. Since real world allocators are not truly nondeterministic, this model captures the semantics more accurately, while also paving the way for reasoning about programs with alternative evaluation models.

**Safe preconditions.** The final restriction dictated by the frame rule relates to safety: framing cannot affect whether or not a program will encounter a memory fault. The reason for this restriction is because in typical partial correctness logics, any postcondition is valid if the program encounters a fault. So, the triple \{\text{emp}\} [\{x\}] ← 1 \{\text{emp}\} is valid since the program is guaranteed to fault.

Now, using the frame rule, we can add information about \( y \) to obtain \{\{y \mapsto 2\}\} [\{x\}] ← 1 \{\{y \mapsto 2\}\}, which is untrue in the case that \( y \) aliases \( x \). In response, separation logic requires the precondition to be \emph{safe} for all program paths; any state satisfying the precondition will not encounter a fault.
Safety of the precondition is undesirable for our purposes; it requires that we examine the entire program, which is at odds with dropping paths for reasoning about incorrectness. Fortunately, as we show in Section 4.2, OSL preconditions need not be safe since OSL is not a partial correctness logic. Following the previous example, \( (\text{ok} : \text{emp}) \times [x] \leftarrow 1 \) \( (\text{ok} : \text{emp}) \) is not a valid OSL specification since \( \text{ok} : \text{emp} \) is not a \textit{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., \( (\text{ok} : \text{emp}) \times [x] \leftarrow 1 \). Framing information into the latter triple is perfectly safe since the post will absorb \textit{any} outcome including undefined behavior, nontermination, and successful termination \( (\text{ok} : y \rightarrow 2) \times [x] \leftarrow 1 \).

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
We now describe how 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 [Calcagno et al. 2009] and Infer [Calcagno et al. 2015]), but with greater care taken to handle effects. The aforementioned tools can analyze nondeterministic programs, but often 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 mentioned a possible fix that involves 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 program choices, which is exponential in the worst case. We take a different approach, acknowledging that choice operations are fundamentally different from sequential composition and therefore require 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 program branches—the goal is to find a single anti-frame \( M \) and two leftover frames \( F_1 \) and \( F_2 \) such that \( M = P_1 \ast F_1 \) and \( M = P_2 \ast F_2 \), allowing us to compose the summaries for two program branches in a parallel fashion, as demonstrated below.\(^2\)

\[
\begin{array}{c}
(P_1) \ C_1 \ C_2 \ Q_1 \ \ \ \ \ P_1 \ast F_1 \ast M \ast P_2 \ast F_2 \\
(M) \ C_1 \ C_2 \ \ \ ((Q_1 \ast F_1) \oplus_a (Q_2 \ast F_2))
\end{array}
\]

TRI-ABDUCTIVE COMPOSITION

Note that the choice operator \( \oplus_a \) and the outcome conjunction \( \oplus_a \) are parameterized by a weight. In Section 3, we describe interpretations of these weights that allow us to algebraically represent both nondeterministic and probabilistic choice in a uniform way. Tri-abduction does not replace bi-abduction, but rather they work together in complementary ways—bi-abduction is used to compose commands in a sequence 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 \( (\text{ok} : P) \ C (\text{err} : Q) \oplus_a \top) \), with only a single outcome specified and the remaining ones covered by \( \top \). The soundness of the

\(^2\)We use the word \textit{parallel} to contrast the sequential nature of bi-abduction. In this case, we are not talking about concurrent branches, but rather branches that result from computational effects—namely nondeterministic and probabilistic choice.
single-path approach is motivated by the fact that $P \oplus_a Q \Rightarrow P \oplus_a T$; extraneous outcomes can be converted to $T$, 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 have now seen an overview of how OSL is built from the ground up to support local reasoning for correctness and incorrectness in the presence of effects, and how tri-abduction aids in building symbolic execution algorithms for programs with control flow branching. In the remainder of the paper, we formalize these concepts. In Sections 3 and 4, we define a program semantics and Outcome Separation Logic and prove the soundness of the frame rule. Tri-abduction is defined in Section 5 and symbolic execution in Section 6, before we examine case studies in Section 7.

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.

\[ C \in \text{Cmd} ::= \text{skip} \mid C_1 ; C_2 \mid C_1 +_a C_2 \mid \text{if} \ e \ \text{then} \ C_1 \ \text{else} \ C_2 \mid \text{while} \ e \ \text{do} \ C \mid c \]

\[ c \in \text{Instr} ::= x := e \mid x := \text{alloc}() \mid \text{free}(e) \mid [e_1] \leftarrow e_2 \mid x \leftarrow [e] \mid \text{error}() \mid f(\vec{e}) \]

\[ e \in \text{Exp} ::= e_1 = e_2 \mid \neg e \mid x \mid X \mid \kappa \]

Commands $C \in \text{Cmd}$ are similar to those of Dijkstra’s [1975] Guarded Command Language (GCL), containing skip, sequencing ($C_1 ; C_2$), and the usual control flow structures if and while. Unlike GCL, the choice operator $C_1 +_a C_2$ is parameterized by $a \in A$, which has certain algebraic properties described in Section 3.1. For example, in probabilistic programs $A = [0, 1]$ and is interpreted as a probability, so $C_1 +_p C_2$ runs $C_1$ with probability $p$ and $C_2$ with probability $1 - p$.

Instructions $c \in \text{Instr}$ can assign to variables ($x := e$), allocate ($x := \text{alloc}()$) and deallocate ($\text{free}(e)$) memory, write ($[e_1] \leftarrow e_2$) and read ($x \leftarrow [e]$) pointers, crash (error($()$)), and call procedures ($f(\vec{e})$). Expressions have a limited syntax, containing equalities, negation, program variables $x \in \text{Var}$, logical variables $X \in \text{LVar}$, and constants $\kappa \in \text{Const}$ (e.g., integers and Booleans). This ensures that entailments containing expressions are decidable, which is nontrivial even after ruling out pointer arithmetic [Berdine et al. 2005a], but is necessary for bi-abduction [Calcagno et al. 2009]\(^3\).

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 $a \in A$, 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 $(A, +, \emptyset)$ consists of a carrier set $A$, an associative binary operator $+: A \times A \rightarrow A$, and an identity element $\emptyset \in A$ such that $a + \emptyset = \emptyset + a = a$ for all $a \in A$. Additionally, a monoid is partial if $+$ is partial ($+: A \times A \rightarrow A$) and it is commutative if $a + b = b + a$.

For example, $(\{0, 1\}, +, 0)$ 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 $(\{0, 1\}, \cdot, 1)$ is another monoid with with same carrier set, but it is total rather than partial. These two monoids can be combined to form a semiring, as follows.

\(^3\)Our expression syntax differs slightly from that of Berdine et al. [2005a], which included $\neq$ rather than logical negation. The two are nonetheless equally expressive since $e_1 \neq e_2$ is equivalent to $\neg(e_1 = e_2)$ and $\neg e$ is equivalent to $e = false$. 
A semiring is partial if it diverges. To encode this, we use an Outcome Algebra supported weighting functions and the monad operations are defined as follows:

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

1. \( \langle A, +, 0 \rangle \) is a commutative monoid.
2. \( \langle A, \cdot, 1 \rangle \) 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 \( (a + b) \cdot c = ac + bc \).
4. \( \emptyset \) is the annihilator of multiplication: \( a \cdot \emptyset = \emptyset \cdot a = \emptyset \).

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

Definition 3.3 (Natural Ordering). The natural order of a semiring \( \langle A, +, \cdot, 0, 1 \rangle \) is defined to be \( a \leq b \) if \( \exists a' \in A. a + a' = b \). A semiring is naturally ordered if \( \leq \) is a partial order.

For the probabilistic semiring, the natural order corresponds to real number comparison. We now define Outcome Algebras that give the interpretation of choice \( C_1 +_a C_2 \). 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 3.4 (Outcome Algebra). An Outcome Algebra is a structure \( \langle A, +, \cdot, 1, 0 \rangle \) in which \( \langle A, +, \cdot, 0, 1 \rangle \) is a complete, Scott continuous, naturally ordered, partial semiring, and:

1. \( \sim : A \to A \) is a partial unary operation such that if \( \overline{a} \) is defined, then \( a + \overline{a} = 1 \) and \( \overline{a} = a \).
2. \( \langle A, \leq \rangle \) is a complete partial order (cpo) and \( \sup(A) = 1 \).
3. If \( \sum_{i \in I} a_i \) is defined, there exist \( (b_i)_{i \in I} \) such that \( \sum_{i \in I} b_i = 1 \) 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 (3).

Outcome Algebras can encode the following three interpretations of choice:

Definition 3.5 (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. In addition, \( \cdot \) is typical integer multiplication, and \( \overline{a} = 1 - a \).

Definition 3.6 (Nondeterministic Outcome Algebra). The nondeterministic Outcome Algebra is \( \langle \{0, 1\}, \lor, \land, \overline{\cdot}, 0, 1 \rangle \). Similar to the previous case, the elements are Booleans indicating whether an outcome has occurred, but now the semiring addition is a logical disjunction, indicating that outcomes can be combined. We also define \( \overline{1} = 1 \), and \( \overline{0} \) is not defined.

Definition 3.7 (Probabilistic Outcome Algebra). Let \( \langle \{0, 1\}, +, \cdot, 0, 1 \rangle \) be an outcome algebra where \( + \) is real-valued addition (and undefined if \( a + b > 1 \)), \( \cdot \) is real-valued multiplication, and \( \overline{a} = 1 - a \). 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 3.8 (Outcome Monad). Given an Outcome Algebra \( \mathcal{A} = \langle A, +, \cdot, 0, 1 \rangle \), we define a monad \( \langle \mathcal{W}_\mathcal{A}, \text{unit}, \text{bind} \rangle \), where \( \mathcal{W}_\mathcal{A}S = \{ m : S \to A \mid \sum_{s \in \text{supp}(m)} m(s) \text{ is defined} \} \) is the set of countably supported weighting functions and the monad operations are defined as follows:

\[
\text{unit}(s)(t) = \begin{cases} 
1 & \text{if } s = t \\
0 & \text{if } s \neq t 
\end{cases} \quad \text{bind}(m, f)(t) = \sum_{s \in \text{supp}(m)} m(s) \cdot f(s)(t)
\]
We also let \( \text{supp}(m) = \{ s \mid m(s) \neq \emptyset \} \) and \( |m| = \sum_{s \in \text{supp}(m)} m(s) \). This monad is very 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 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 \( a \in \mathcal{A} \) to each program state \( s \in S \). Definitions 3.5 to 3.7 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 3.5, the constraint that \( \sum_{s \in \text{supp}(m)} m(s) \) 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: \( m_1 + m_2 = \lambda s. (m_1(s) + m_2(s)) \), \( \emptyset = \lambda s. 0 \), and \( a \cdot m = \lambda s. (a \cdot m(s)) \). When the nondeterministic algebra (Definition 3.6) is lifted in this way, the result is isomorphic to the powerset monad with \( m_1 + m_2 = m_1 \cup m_2 \) and \( \emptyset = \emptyset \).

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:

\[
\begin{align*}
\text{\texttt{1}}_{\text{ok}} &: S \rightarrow S + E + 1 \\
\text{\texttt{1}}_{\text{er}} &: E \rightarrow S + E + 1 \\
\text{\texttt{1}}_{\text{undef}} &: 1 \rightarrow S + E + 1 \\
\text{\texttt{undef}} &= \text{\texttt{1}}_{\text{undef}}(\ast)
\end{align*}
\]

Borrowing the notation of Incorrectness Logic [O’Hearn 2019], we use \( \text{\texttt{ok}} \) and \( \text{\texttt{er}} \) to denote states in which the program terminated successfully or crashed, respectively. We will also write \( \text{\texttt{e}} \) to refer to one of the above injections, where \( e \in \{\text{\texttt{ok}}, \text{\texttt{er}}\} \).

**Definition 3.9 (Error Monad Transformer).** Let \( \langle \mathcal{W}_\mathcal{A}, \text{bind}_{\mathcal{W}}, \text{unit}_{\mathcal{W}} \rangle \) be the outcome monad described in Definition 3.8 and let \( E \) be some set of error states. We define a new monad \( \langle \mathcal{W}_\mathcal{A}(\sim + E + 1), \text{bind}, \text{unit} \rangle \) where the monad operations are defined as follows:

\[
\begin{align*}
\text{\texttt{unit}}(s) &= \text{\texttt{unit}}_{\mathcal{W}}(\text{\texttt{1}}_{\text{ok}}(s)) \\
\text{\texttt{bind}}(m, f) &= \text{\texttt{bind}}_{\mathcal{W}}\left(m, \lambda \sigma. \begin{cases} f(s) & \text{if } \sigma = \text{\texttt{1}}_{\text{ok}}(s) \\ \text{\texttt{unit}}_{\mathcal{W}}(\sigma) & \text{otherwise} \end{cases} \right)
\end{align*}
\]

### 3.2 Denotational Semantics

For every command \( C \) we define a function \( \llbracket C \rrbracket : S \times \mathcal{H} \rightarrow \mathcal{W}_\mathcal{A}\text{St} \), as shown in Figure 1. The set of program states is \( \text{St} = S \times \mathcal{H} + S \times \mathcal{H} + 1 \), where \( S = \{ s : \text{Var} \cup \text{LVar} \rightarrow \text{Val} \} \) are variable stores and \( \mathcal{H} = \{ h : \text{Addr} \rightarrow \text{Val} \cup \{\bot\} \} \) 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 (\( \ell \notin \text{dom}(h) \)) vs cases where a pointer is explicitly deallocated (\( h(\ell) = \bot \)). The set of addresses \( \text{Addr} \) is opaque, but we assume \( \text{null} \in \text{Const} \subseteq \text{Val} \) and \( \text{Addr} \cap \text{Const} = \emptyset \).

In addition to being parametric on an outcome algebra \( \mathcal{A} = \langle A, +, \cdot, \ast, 0, 1 \rangle \), the semantics is also parametric on an allocator \( \text{alloc} : S \times \mathcal{H} \rightarrow \mathcal{W}_\mathcal{A}\text{Addr} \) such that \( |\text{alloc}(s, h)| = 1 \) and \( \text{supp}(\text{alloc}(s, h)) \) is disjoint from all the addresses found in \( s \) and \( h \).\(^4\) A global procedure table \( \text{P} : \text{Proc} \rightarrow \text{Cmd} \times \text{Var} \) that returns a command and vector of variable names (the arguments) given a procedure name \( f \in \text{Proc} \) is used to interpret procedure calls. We assume that all procedures used in programs are defined and pass the correct number of arguments.

\(^4\) A deterministic allocator \( \text{alloc}(s, h) = \min(\text{Addr} \setminus (\text{dom}(h) \cup \text{im}(s) \cup \text{im}(h))) \) that always picks the first unused address is valid in all OSL instances. The image of a function is defined as follows \( \text{im}(f) = \{ f(x) \mid x \in \text{dom}(f) \} \).
The monad operations (Definition 3.9) are used to give semantics to skip and $\downarrow$ in the standard way, if statements select the appropriate branch according to the value of the Boolean guard, and while loops are defined using a least fixed point.

Choice $C_1 +_a C_2$ is defined as a weighted sum whose meaning depends on the algebraic interpretation. Choice is not so useful in the deterministic model (Definition 3.5) since $C_1 +_1 C_2$ is equivalent to $C_1$ and $C_1 +_0 C_2$ is equivalent to $C_2$. In the nondeterministic model (Definition 3.6), since $0$ is undefined, we write $C_1 + C_2$ instead of $C_1 +_1 C_2$, which will evaluate to the set of all the outcomes of both $C_1$ and $C_2$. Finally, in the probabilistic model (Definition 3.7), $C_1 +_a C_2$ evaluates to a probability distribution where $C_1$ is chosen with probability $a$ and $C_2$ is chosen with probability $1 - a$.

We define two operations before giving the semantics of instructions: error$(s, h)$ constructs an error state and update$(s, h, \ell, s', h')$ returns $(s', h')$ if the address $\ell$ is allocated in $h$, it returns an error if $\ell$ is deallocated, and is undefined if $\ell \not\in \text{dom}(h)$. Assignment is defined in the usual way by updating the program store; memory allocation uses the alloc operation to obtain a fresh address (or collection thereof) and initializes the value to 0; 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 $\vec{e}$.

Despite using the partial sum operation and a least fixed point, the semantics of programs is total. We prove this in Appendix A. We will additionally occasionally use the monadic extension of the semantics, which is defined as $[C]^\downarrow(m) = \text{bind}(m, [C])$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{Denotational semantics of program commands, parametric on an outcome algebra $\mathcal{A} = (A, +, \cdot, 0, 1)$, an allocator function alloc : $S \times \mathcal{H} \rightarrow \mathcal{W}_\mathcal{A}\text{Addr}$, and a procedure table $P : \text{Proc} \rightarrow \text{Cmd} \times \text{Var}$.}
\end{figure}
We now proceed to define Outcome Separation Logic (OSL), present a frame rule, and prove its soundness. First, we define an assertion logic that will be used as the pre- and postconditions of outcome triples. These assertions are based on the outcome assertions of Zilberstein et al. [2023], using the symbolic heaps of Berdine et al. [2005a,b] as basic predicates.

**Symbolic Heaps.** The syntax for symbolic heaps is shown below and the semantics is standard as defined by Berdine et al. [2005b, §2].

\[
P ::= \exists X.\Delta \quad \text{(Symbolic Heaps)} \quad \Pi ::= \text{true} \mid \Pi_1 \land \Pi_2 \mid e \quad \text{(Pure Assertions)}
\]

\[\Delta ::= \Pi \land \Sigma \quad \text{(Quantifier-Free)} \quad \Sigma ::= \text{true} \mid \text{emp} \mid \Sigma_1 * \Sigma_2 \mid e_1 \iff e_2 \mid \text{ls}(e_1, e_2) \quad \text{(Spacial Assertions)}\]

A symbolic heap \(P\) consists of existentially quantified logical variables, along with a pure part \(\Pi\) and a spacial part \(\Sigma\). A pure assertion is a conjunction of Boolean valued expressions \(e \in \text{Exp}\) (equalities and inequalities), whereas a spacial assertion is a sequence of heap assertions joined by **separating** conjunctions. The separating conjunction requires that the heap can be split into two disjoint components to satisfy the two assertions separately.

\[(s, h) \models \Sigma_1 * \Sigma_2 \iff \exists h_1, h_2. \ h = h_1 \uplus h_2 \ \text{and} \ (s, h_1) \models \Sigma_1 \ \text{and} \ (s, h_2) \models \Sigma_2\]

The points-to predicate \(e_1 \iff e_2\) specifies a singleton heap in which the address \(e_1\) points to the value \(e_2\). Negative heap assertions \(e \not\iff\) are syntactic sugar for \(e \iff \bot\). These assertions were introduced in Incorrectness Separation Logic to express that a pointer is invalidated in order to prove that a program crashes due to a memory error [Raad et al. 2020]. Finally, inductive list segment predicates \(\text{ls}(e_1, e_2)\), state that there is a sequence of pointers starting with \(e_1\) and ending with \(e_2\). Formally, it is the least solution of \(\text{ls}(e_1, e_2) \iff (e_1 = e_2 \land \text{emp}) \lor (\exists X. e_1 \iff X \land \text{ls}(X, e_2))\). We also provide overloaded definitions of the * and \(\land\) operators.

\[(\exists X.\Pi \land \Sigma) * (\exists Y.\Pi' \land \Sigma') \triangleq \exists X,Y.((\Pi \land \Pi') \land (\Sigma * \Sigma')) \quad P \land \Pi \triangleq P * (\Pi \land \text{emp})\]

Though symbolic heaps have limited expressivity—particularly for pure assertions—they have a complete decision procedure [Berdine et al. 2005a], which is necessary for bi-abductive analysis algorithms. These same symbolic heaps are used by Calcagno et al. [2009, 2011].

**Outcome Assertions.** OSL assertions are based on the outcome assertions of Zilberstein et al. [2023]. They use symbolic heaps as basic predicates and rely on an outcome algebra for their interpretation. The syntax for these assertions is below and their semantics is in Figure 2.

\[
\varphi ::= \top \mid \top^\oplus \mid \varphi \lor \psi \mid \varphi \oplus \psi \mid e \in P \quad \epsilon ::= \text{ok} \mid \text{er}
\]

Outcome assertions include some familiar constructs such as \(\top\), which is always true, and disjunctions \(\varphi \lor \psi\). Additionally, we have \(\top^\oplus\), which asserts that there are no outcomes.

The weighted outcome conjunction \(\varphi \oplus \psi\) splits the program configuration \(m\) into two pieces \(m_1\) and \(m_2\) whose weighted sum is equal to \(m\) (see Figure 2, note that this connective requires \(\overline{a}\) to be defined). Using the nondeterministic interpretation (Definition 3.6), the only valid choice of \(a\) is 1, thus we will omit the \(a\) in this case and simply write \(\varphi \oplus \psi\), which means that \(\varphi\) and \(\psi\) are outcomes.

![Figure 2. Semantics of outcome assertions given an outcome algebra \(\langle A, +, \cdot, \top, 0, 1\rangle\).](image-url)
that can arise due to nondeterminism. In the probabilistic interpretation, for any \( p \in [0, 1] \), the assertion \( \varphi \oplus_p \psi \) states that \( \varphi \) occurs with probability \( p \) and \( \psi \) occurs with probability \( 1 - p \).

Finally, basic assertions (\( \text{ok} : P \)) and (\( \text{er} : Q \)) require that all states in \( \text{supp}(m) \) terminated successfully and satisfy \( P \) or crashed and satisfy \( Q \), respectively, where \( P \) and \( Q \) are symbolic heaps. The requirement that |\( m \)| = 1 ensures that the set of outcomes is nonempty in the (non)deterministic cases (Definitions 3.5 and 3.6) and that the total mass is 1 in the probabilistic case (Definition 3.7).

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 \{\( P \) \} \( f(\bar{e}) \) \{\( \varphi \) \} \{\( \psi \) \} and the disjunctive post indicates a series of possible outcomes. In our case, we will exchange those disjunctions for outcome conjunctions in the cases where the outcomes arise due to computational effects. We include \( T \) in order to drop outcomes using the assertion \( \varphi \ominus_a T \) (as discussed in Section 2). Finally, we include the standard disjunction to express joins of outcomes that occur due to logical choice, and also to express partial correctness; while \( e : P \) guarantees reachability, \( (e : P) \lor T^\ominus \) also permits nontermination.

**Remark 1 (Disproving Outcome Assertions).** Given the goal of creating symbolic execution algorithms for both correctness and incorrectness, it makes sense to ask whether it is possible to disprove triples using outcome assertions as pre- and postconditions. This may seem dubious given that the syntax does not include logical negation, however Zilberstein et al. [2023, §5] showed that outcome assertions that are expressed as a sequence atomic assertions separated by \( \oplus \) can be disproven using a different sequence of atoms separated by \( \ominus \)—logical negation is, in fact, unnecessary.

**OSL Triples.** The semantics of OSL triples is the same as those of standard OL, and—crucially—makes no mention of safety that is typically needed to guarantee the soundness of the frame rule [Yang and O’Hearn 2002]. As we discussed in Section 2 and discuss further in Section 4.2, the safety requirement is incompatible with dropping program paths, so for OSL it is important to omit it.

**Definition 4.1 (Outcome Separation Logic Triples).** Given an outcome algebra \( \mathcal{A} = (A, +, \cdot, \lnot, 0, 1) \) and an allocator function, the validity of OSL triples is defined as follows:

\[
\vDash \langle \varphi \rangle C \langle \psi \rangle \quad \text{iff} \quad \forall m \in W_{\mathcal{A}} \text{St.} \quad m \vDash \varphi \quad \implies \quad [C]^\varphi(m) \vDash \psi
\]

**The Frame Rule.** In the remainder of this section, we build the necessary foundations to introduce and prove the soundness of the frame rule. Before doing so, we need a separating conjunction for outcome assertions. Rather than adding the separating conjunction as a logical connective, we define it inductively as a transformation on outcome assertions below. We use the symbol \( \ominus \) to distinguish it from the usual separating conjunction \( * \) on symbolic heaps; \( \ominus \) is a binary operation taking an outcome assertion and a symbolic heap (rather than two symbolic heaps like \( * \)).

\[
\begin{align*}
T \ominus F & \triangleq T \\
T^\ominus \ominus F & \triangleq T^\ominus \\
(\varphi \lor \psi) \ominus F & \triangleq (\varphi \ominus F) \lor (\psi \ominus F) \\
(\varphi \oplus_a \psi) \ominus F & \triangleq (\varphi \ominus F) \oplus_a (\psi \ominus F) \\
(e : P) \ominus F & \triangleq e : P \ast F
\end{align*}
\]

So, \( \ominus \) has no effect on \( T \) and \( T^\ominus \), it distributes over \( \lor \) and \( \oplus_a \), and for basic assertions \( e : P \), we simply join \( P \ast F \) with the usual separating conjunction. We can now express the OSL frame rule:

\[
\frac{\langle \varphi \rangle C \langle \psi \rangle \quad \text{mod}(C) \cap \text{fv}(F) = \emptyset}{\langle \varphi \ominus F \rangle C \langle \psi \ominus F \rangle}
\]

This rule resembles the frame rule of separation logic [O’Hearn et al. 2001], with the same side condition stating \( F \) must not mention any modified program variables. However, unlike the standard
frame rule, it can be used to reason about multiple outcomes simultaneously. For example, suppose we wanted to add information about a pointer \( y \) to the specification we saw in Section 2.1.

\[
\begin{align*}
\text{ok : emp} &\quad x := \text{malloc()} \triangleright [x] \leftarrow 1 \langle (\text{ok} : x \mapsto 1) \oplus (\text{er} : x = \text{null} \land \text{emp}) \rangle \\
\text{ok : y \mapsto 2} &\quad x := \text{malloc()} \triangleright [x] \leftarrow 1 \langle (\text{ok} : x \mapsto 1 \ast y \mapsto 2) \oplus (\text{er} : x = \text{null} \land y \mapsto 2) \rangle \\
\end{align*}
\]

As discussed in Section 2.2, proving the soundness of the OSF frame rule requires different underlying assumptions. The key is to show that if \((s, h') \not\models F\), then \([C](s, h \uplus h')\) is related to \([C](s, h)\) (Lemma 4.7), which accordingly dictates that if \([C](s, h) \models \psi\) then \([C](s, h \uplus h') \not\models \psi \circ F\) (Theorem 4.8). Specifically, \([C](s, h \uplus h')\) will have a larger memory footprint (Section 4.1), the allocated addresses may change as the result of running the program in a larger heap, and the undefined states of \([C](s, h)\) may become defined after augmenting the heap (Section 4.2). We will express this by lifting relations on states to relations on weighting functions, as described below.

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

\[
\overline{R} = \left\{ (m_1, m_2) \mid \exists m \in \mathcal{W}_A R. \ m_1 = \lambda x. \sum_{y \in \text{supp}(m_2)} m(x, y) \quad \text{and} \quad m_2 = \lambda y. \sum_{x \in \text{supp}(m_1)} m(x, y) \right\}
\]

### 4.1 Semantics of the Outcome Separating Conjunction

We now prove semantic properties about \( \varphi \circ F \). First, we need to relate states satisfying \( P \) to states satisfying \( P \ast F \).

\[
\text{frame}(F) = \left\{ (1_F(s, h), 1_F(s, h \uplus h')) \mid 1_F(s, h) \in \text{St}, (s, h') \not\models F \right\} \cup \{(\text{undef}, \text{undef})\}
\]

Any state \( 1_F(s, h) \) is related to all states \( 1_F(s, h \uplus h') \) such that \((s, h') \not\models F\), which guarantees that if \( (s, h) \not\models P \), then \( (s, h \uplus h') \not\models P \ast F \). Undefined states are only related to themselves. Now, we can express the semantics of \( \varphi \circ F \) by lifting this relation.

**Lemma 4.3.** If \( m \models \varphi \) and \((m, m') \in \text{frame}(F)\), then \( m' \not\models \varphi \circ F \)

It is tempting to say that the converse should also hold, but that is not quite right. We took \( T \circ F \) to be equal to \( T \), therefore if \( m \models T \circ F \), 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 \( T \), leaving the other states unconstrained.

**Lemma 4.4.** If \( m \models \varphi \circ F \), then there exist \( m_1, m'_1 \), and \( m_2 \) such that \((m_1, m'_1) \in \overline{\text{frame}(F)}\), \( m = m'_1 + m_2 \) and \( m_1 + m'_2 \models \varphi \) for any \( m'_2 \) such that \( |m'_2| \leq |m_2| \).

In the lemma above, \( m'_2 \) represents the nontrivial portion of \( m \) and \( m_2 \) is the portion of \( m \) that is covered by \( T \). As such, \( m'_2 \) must be the result of framing \( F \) into some \( m_1 \). Since \( m_2 \) is covered by \( T \), we can replace it with anything smaller than \( m_2 - T \) can absorb at least \( |m_2| \) worth of mass. These two lemmas provide a semantic basis to reason about what it means for \( \varphi \circ F \) to hold relative to \( \varphi \).

### 4.2 Heap Permutations and Replacement of Unsafe States

Running a program in a larger heap can affect the end configuration in more ways than just adding pointers. In particular, it can cause the addresses of freshly allocated pointers to change, and it can cause previously undefined end states to become defined. The soundness of the frame rule requires that neither of these changes affect the truth of the postcondition.

**Heap Permutations.** In Section 2, we saw what goes wrong when using the frame rule with a triple that specifies the address of an allocated pointer. If the allocator always returns the first
available slot, then \(\langle \text{ok} : \text{emp}\rangle x := \text{alloc}() \langle \text{ok} : x = 1\rangle\) is a valid triple, but using the frame rule to obtain \(\langle \text{ok} : y \mapsto 2\rangle x := \text{alloc}() \langle \text{ok} : y \mapsto 2 \land x = 1\rangle\) is invalid since \(y\) could have address 1.

Yang and O’Hearn [2002] solve this problem by forcing memory allocation to be nondeterministic. If the allocator could return \textit{any} address, then the postcondition cannot say anything too specific about \textit{which} address got returned. Nondeterminism works in unison with the semantics of Hoare Logic, requiring that a single predicate in the postcondition covers all possible end states.

This approach is undesirable in OL for two reasons. First, we do not want to rely on a nondeterministic evaluation model. Second, we want to be able to reason about reachable states and incorrectness, which are not compatible with the latter restriction. For example, if memory allocation is nondeterministic, then \(\langle \text{ok} : \text{emp}\rangle x := \text{alloc}() \langle \bigoplus_{\ell \in \text{Addr}} (\text{ok} : x = \ell)\rangle\) is valid—it asserts that there are reachable outcomes where \(x\) is equal to every address. Framing in information about another pointer would invalidate one of those outcomes, so the frame rule is unsound.

We instead take the approach of considering heap addresses to be \textit{nominal}. That is, in the style of nominal logic [Pitts 2003], the satisfaction of symbolic heaps and outcome assertions are stable under permutation of the heap addresses—the satisfaction relation is \textit{equivariant}. To formalize this, we first define a relation \(\text{Perm} = \{(\sigma, \pi(\sigma)) \mid \sigma \in \text{St}, \pi \text{ is a permutation}\}\) which relates all program states to other states in which the heap addresses are permuted (see Appendix C.2 for more information about these permutations). The equivariance property is stated by lifting \(\text{Perm}\).

\textbf{Lemma 4.5 (Equivariance)}. If \(m \vDash \varphi\) and \((m, m') \in \overline{\text{Perm}}\), then \(m' \vDash \varphi\).

\textbf{Replacement of Unsafe States}. Whereas the semantics of separation logic requires that all states satisfying the precondition are \textit{safe} (they will not fault), we explicitly omit this requirement in OSL. Efficiently reasoning about incorrectness requires us to only explore a subset of the program paths, and the dropped paths are not guaranteed to be safe. To demonstrate this, consider the triple \(\langle \text{ok} : x \not\rightarrow\rangle \text{free}(x) + C \langle (\text{er} : x \not\rightarrow) \oplus \top \rangle\). 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 \textit{undef}. This is fine, since the assertion \(\top\) absorbs undefined states.

However, if we used the frame rule to add information about more pointers to the precondition, the result may no longer be \textit{undef}. This is still valid—\textit{any} outcome from running \(C\) trivially satisfies \(\top\). To formalize this, we use \(\text{Rep} \subseteq \text{St} \times \text{St}_\perp\), which relates \textit{undef} to any state in \(\text{St}\) or \(\perp\) (representing divergent outcomes)\(^5\). All other states \((\text{ok}\) and \(\text{er}\)) are related only to themselves. Since \(\perp\) is not a program state, we define an operation \(\text{prune}(m)\) to remove it. The formal definitions of both \(\text{Rep}\) and \(\text{prune}\) are given in Appendix C.3. Now we can state the replacement lemma which says that undefined states can be replaced by anything without affecting the validity of an outcome assertion.

\textbf{Lemma 4.6 (Replacement)}. If \(m \vDash \varphi\) and \((m, m') \in \overline{\text{Rep}}\), then \(\text{prune}(m') \vDash \varphi\).

\subsection{Soundness of the Frame Rule}

We now have all the ingredients needed to prove the soundness of the frame rule. The first step is to prove the frame property, which describes how the result of running the program on a \textit{framed} start state \((s, h \cup h')\) is related to just running it on the unframed state \((s, h)\).

Running the program on \((s, h \cup h')\) will redistribute the mass of \([C](s, h)\) according to the three relations that we defined previously: the addresses may be permuted (\(\text{Perm}\)), information about additional pointers will be added (\textit{frame} \(F\)), and the undefined states may be replaced (\(\text{Rep}\)). As is

\(^5\)Previously undefined states may diverge after adding more pointers, \textit{e.g.}, running \textit{free} \((x)\) \(\not\rightarrow\) \textit{while} do \textit{skip} in an empty heap leads to \textit{undef} whereas it will not terminate if \(x\) is allocated.
usual for the frame rule, we also require that the modified program variables are disjoint from the free variables of the symbolic heap $F$.

**Lemma 4.7 (The Frame Property).** Let $R = \text{Rep} \circ \text{frame}(F) \circ \text{Perm}$, so $R \subseteq St \times St$. For any program $C$ and symbolic heap $F$ such that $\text{mod}(C) \cap \text{fv}(F) = \emptyset$:

\[
\forall (\langle ok \rangle (s, h), \langle \text{ok} \rangle (s', h')) \in R. \quad \exists m. \quad \exists \emptyset, \exists (C \{s', h'\} = \text{prune}(m)) \text{ and } (\langle C \{s, h\}, m \rangle) \in R
\]

Now, given the frame property, we know how the result of running the program on a framed input will relate to running it on an unframed one. In order to establish the soundness of the frame rule, all that remains is to show what happens as we peel away the three layers of the relation $R$.

**Theorem 4.8 (The Frame Rule).** If $\mathcal{F} = \langle \varphi \rangle C \langle \psi \rangle$ and $\text{mod}(C) \cap \text{fv}(F) = \emptyset$, then $\mathcal{F} = \langle \varphi \odot F \rangle C \langle \psi \odot F \rangle$.

We briefly sketch the proof here, whereas the full version is in Appendix C.4. Suppose $m \vDash \varphi \odot F$, then by Lemma 4.4, we can get an $m' \vDash \varphi$ and by the premise of the rule, we also know that $\exists \emptyset, \exists (C \{m'\} \vDash \psi)$. Using Lemma 4.7, running the program on $m$ will give us a result that we can relate to running the program on $m'$ via the three relations described in the previous sections. We can then peel away those relations using Lemmas 4.3, 4.5 and 4.6 to conclude that $m \vDash \psi \odot F$.

We have devised a sound frame rule for Outcome Separation Logic despite relaxing some of the restrictions of separation logic that are intended to ensure the soundness of framing. Since the OSL frame rule does not rely on nondeterminism, it can be used for deterministic and probabilistic languages too. It also does not require start states to be safe, so OSL can be used to reason about incorrectness without inspecting the entire program. In the next sections, we will see how the OSL theory can be used to build compositional symbolic execution algorithms.

### 5 TRI-ABDUCTION

The last step before defining symbolic execution algorithms for OSL is to address the matter of composing paths in programs with choice mechanisms arising from nondeterminism or random sampling. When symbolically executing such programs, we must unify the preconditions for the two branches. For example, in the following program that chooses to execute $[x] \leftarrow 1$ or $[y] \leftarrow 2$, we need a precondition that mentions both pointers $x$ and $y$, and we need to know what leftover resources to add to the two resulting outcomes.

\[
\langle \text{ok} : ? \rangle \xrightarrow{\top_a} \langle \text{ok} : x \leftrightarrow X \rangle \quad \langle \text{ok} : y \leftrightarrow Y \rangle \xrightarrow{\bot_a} \langle \text{ok} : x \leftrightarrow 1 \rangle \quad \langle \text{ok} : y \leftrightarrow 2 \rangle
\]

Tri-abduction, the parallel composition analogue of bi-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 $P_1 \ast F_1 \equiv M \equiv P_2 \ast F_2$ hold. Using this, we can compose program branches according to the inference below.

\[
\frac{\langle P_1 \rangle C_1 \langle Q_1 \rangle \quad P_1 \ast F_1 \equiv M \equiv P_2 \ast F_2 \quad \langle P_2 \rangle C_2 \langle Q_2 \rangle}{\langle M \rangle C_1 \ast \langle (Q_1 \ast F_1) \odot \langle Q_2 \ast F_2 \rangle \rangle}_{\text{TRI-ABDUCTIVE COMPOSITION}}
\]

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 of them 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 the entire program. Using tri-abduction, we infer more summaries and do so in a single pass.
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 \( P_1 \ast M \models P_2 \ast F \). Using \( P_1 \ast M \) as the anti-frame, \( P_1 \ast M \equiv (P_1 \ast M) \models P_2 \ast F \) is a tri-abduction solution.

However, this approach is inherently asymmetric, with the symbolic heap of the left branch being favored. While it would be possible to also bi-abduce in the opposite direction \( (P_2 \ast ? \equiv P_1 \ast ?) \) for symmetry, this still precludes valid solutions. For example, there is no bi-abduction solution for the symbolic heaps \( X \mapsto Y \ast \text{ls}(Y, Z) \) and \( \text{ls}(X, Y) \ast Y \mapsto Z \) (in either direction), whereas tri-abduction finds the anti-frame \( X \mapsto Y \ast Y \mapsto Z \). Tri-abduction is a fundamentally different operation that is precisely designed for parallel composition.

We now describe the procedure for tri-abductive inference. Similar to [Calcagno et al. 2011, Algorithm 3], tri-abduction is done in two stages. First, in Section 5.1, we describe the abduction stage, in which we infer only the anti-frame \( M \). Next, in Section 5.2 we describe how abduction is used as a subroutine to tri-abduce all three parameters \( M, F_1 \), and \( F_2 \).

5.1 Abductive Inference

The abductive inference step abduce-par\((P, Q)\) is performed as a proof search—similar to Calcagno et al. [2011, Algorithm 1]—using the proof rules in Figure 3 to infer judgements of the form \( P \prec [M] \succ Q \), which indicate that \( M \models P \) and \( M \models Q \). In these judgements, \( P \) and \( Q \) can be read as the inputs to the algorithm and \( M \) is the output. We describe the algorithm briefly here, the full version 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 also gets applied (the full set of rules is shown in Figure 6).

The premise of the 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 \( R \neq \text{false} \), 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 eventually reaches a case with no explicit resources (emp or true), in which a base rule applies. The inference rules are described below.

**Base Rules.** The first step is to attempt to apply a base rule in order to terminate the algorithm. BASE-EMP applies when both branches describe empty heaps. In this rule, we check that 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 spacial assertion \( \Sigma \) and the other contains the spacial assertion true, indicating that it can absorb more resources not explicitly mentioned, so we are able to move \( \Sigma \) 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, \( \text{ls}(e, e') \ast ? \equiv \exists X.e \mapsto X \ast ? \) has a solution \( (e \neq e') \), but it does not have a solution with the existential removed since nothing can be added to \( \text{ls}(e, e') \) to force \( e \) to point to a particular \( X \). 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 the EXISTS gives us the anti-frame \( \exists X.X = Y \land X = 1 \),
which is too weak since \( \exists X.X = 1 \neq X = 1 \). In practice, our symbolic execution algorithm always generates fresh logical variables, so we will not have collisions with our usage of the \textsc{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. \textsc{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 \textsc{Match} rule applies when both branches use \( e_1 \) in a points-to predicate, therefore the value pointed to must be equal too. \textsc{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 \( \Pi \land \Sigma \neq B(e_1, e_2) \neq \text{false} \). The \textsc{Missing-L} rule handles the case wherein one branch refers to resources not present in the other. This is different from the \textsc{Base-True-L} rule, since it handles cases where both branches refer to resources not explicitly present in the other. For example, \textsc{Missing-L} can solve \( x \mapsto X \neq \text{true} <[?]\mapsto y \mapsto Y \neq \text{true} \) even though the \textsc{Base-True} rules do not apply. If one side of the judgement contains a list segment, but the other side does not contain the spacial assertion true, then there is a possible solution where the list segment is empty. \textsc{Emp-Ls-L} handles such cases by forcing the list segment to be empty.
We first abduce a set of anti-frames using Algorithm 1 such that $M$ as we mentioned at the beginning of the section, the tri-abduction algorithm follows a similar

**Theorem 6.1 (Symbolic Execution Soundness).**

If $\langle P, \varphi \rangle \in [C]^d(T)$, then $\equiv \langle \text{ok} : P \rangle C \langle \varphi \rangle$

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
summarize the logical conditions that will occur. Similarly, we produce one summary for each possible number of iterations. We will see two failure cases where the pointer is not allocated or null. Procedure memory operation has three specifications: one in which the pointer is allocated and the operation computes a set of missing anti-frames and postconditions such that \( \langle \varphi \land M \rangle C \langle \psi \rangle \) is a valid specification for \( C \).

After sequencing each of the outcomes in the precondition \( \varphi_1 \) and \( \varphi_2 \) 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 6.2 (Seq).** If \((M, \psi) \in \text{seq}(\varphi, S, \bar{x})\), \(\bar{x} = \text{mod}(C)\), and \(\vdash (ok : P) C \langle \vartheta \rangle\) for all \((P, \vartheta) \in S\), then \(\not\vdash (\varphi \land M) C \langle \psi \rangle\).

**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 in larger heaps. For example, the semantics for skip is simply the triple \(\langle \text{ok : emp} \rangle\) skip \(\langle \text{ok : emp} \rangle\), but this implies that running the program in any heap will yield that same heap in the end.

Executing \(C_1 \parallel C_2\) is implemented using seq; we produce the summaries for \(C_1\) and then sequence them with all the summaries for \(C_2\). Two summaries are produced for if statements, one where the true branch is taken and one where the false branch is taken. This is similar to the behavior of Abductor and allows us to produce precise specifications for the program behavior without a priori knowledge of the logical conditions that will occur. 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. We analyze choices \(C_1 +_a C_2\) by computing summaries for each program path and reconciling them with tri-abduction.

The abstract semantics of primitive instructions 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 \([C]\). In the remainder of this section, we describe further modifications that can be made to the algorithm in order to improve its performance and ability to analyze various types of programs.

**The single-path algorithm** We now present a variant of the previous algorithm that only traverses a single path through the program at a time, and is therefore suitable only for bug-finding. This algorithm is inspired by Pulse [Raad et al. 2020] and Pulse-X [Le et al. 2022], but is based on OSL rather than Incorrectness Logic. We obtain this new algorithm by simply altering the abstract semantics of program choices. Rather than producing a single summary with two outcomes, this version produces two summaries in which the second outcome is replaced by \(T\).

\[
[C_1 +_a C_2]^\#(T) = \{(P, \varphi \land_\pi T) \mid (P, \varphi) \in [C_1]^\#(T)\} \cup \{(P, \varphi \land_\pi T) \mid (P, \varphi) \in [C_2]^\#(T)\}
\]
$C \in \text{Cmd}$ & $[C]^\#(T)$ \\
| skip & $\{ (\text{emp}, \text{ok} : \text{emp}) \}$ \\
| $C_1 \uplus C_2$ & $\{(P \ast M, \psi) \mid (P, \phi) \in [C_1]^\#(T), (M, \psi) \in \text{seq}(\phi, [C_2]^\#(T), \text{mod}(C_2))\}$ \\
| if $e$ then $C_1$ else $C_2$ & $\{(P \land e \land \psi) \mid (P, \psi) \in \text{seq}(\text{ok} : e \land \text{emp}, [C_1]^\#(T), \text{mod}(C_1))\}$ \text{ } \cup \text{ } $\{(P \land \neg e \land \psi) \mid (P, \psi) \in \text{seq}(\text{ok} : \neg e \land \text{emp}, [C_2]^\#(T), \text{mod}(C_2))\}$ \\
| while $e$ do $C$ & $\{(M_1 \ast M_2 \land e, \psi) \mid (M_1, \psi) \in \text{seq}(\text{ok} : e \land \text{emp}, [C_1]^\#(T), \text{mod}(C)), (M_2, \psi) \in \text{seq}(\phi, S, \text{mod}(C))\}$ \\
| $C_1 +_a C_2$ & $\{(M, \psi_1 \oplus_a \psi_2) \mid (M_1, \psi_1) \in [C_1]^\#(T), (M_2, \psi_2) \in [C_2]^\#(T), (M, \psi_1', \psi_2') \in \text{triab}'(M_1, M_2, \psi_1, \psi_2, \text{mod}(C_1, C_2))\}$ \\
| $c \in \text{Instr}$ & $[c]^\#(T)$ \\
| $x := e$ & $\{(x = X \land \text{emp}, \text{ok} : x = e[X/x] \land \text{emp})\}$ \\
| $x := \text{alloc}()$ & $\{(x = X \land \text{emp}, \text{ok} : \exists Y. x \mapsto Y)\}$ \\
| free($e$) & $\{(e \mapsto X, \text{ok} : e \mapsto), (e \mapsto, \text{er} : e \mapsto), (\text{emp} \land e = \text{null}, \text{er} : \text{emp} \land e = \text{null})\}$ \\
| $[e_1]_e \leftarrow e_2$ & $\{(e_1 \mapsto X, \text{ok} : e_1 \mapsto e_2), (e_1 \mapsto, \text{er} : e_1 \mapsto), (\text{emp} \land e_1 = \text{null}, \text{er} : \text{emp} \land e_1 = \text{null})\}$ \\
| $x \leftarrow [e]$ & $\{(x = X \land e \mapsto Y, \text{ok} : x = Y \land e[X/x] \mapsto Y), (e \mapsto, \text{er} : e \mapsto), (\text{emp} \land e = \text{null}, \text{er} : \text{emp} \land e = \text{null})\}$ \\
| error() & $\{(\text{emp}, \text{er} : \text{emp})\}$ \\
| $f(\bar{x})$ & $\{(P \land \bar{x} = X, \phi) \mid (P, \phi) \in \text{seq}(\text{ok} : \bar{x} = \bar{e}[X/\bar{x}] \land \text{emp}, T(f(\bar{x})), \text{mod}(f))\})\}$ \\

Fig. 5. Symbolic execution of commands and instructions, all logical variables $(X,Y)$ are assumed to be fresh.

Given this modification, the algorithm remains sound with respect to the same semantics (i.e., Theorem 6.1), 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 7.1 how it can be used for efficient bug-finding.

**Bounded unrolling and dropping paths.** As recounted by O’Hearn [2019]; Raad et al. [2020]; Le et al. [2022], the scalability benefits of IL-based analyses comes from their ability to **drop disjuncts**. Bi-abduction analyses such as Abductor accumulate a disjunction of symbolic heaps, representing the possible end states at each program point. When searching for bugs, it is not necessary to remember all of these possible states; it suffices to only remember the one that represents a bug. The semantics of IL allows strengthening of postconditions, so 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).

Given our single path algorithm, in which we split each choice (logical and otherwise) into a separate summary, we can drop paths simply by limiting the size of the set $[C]^\#(T)$. For example, while the least fixed point semantics for while loops is clearly uncomputable—it is a set of (possibly) infinite size—we can compute the first $n$ unrollings for some parameter $n$. We can also choose a fixed size for the set $[C]^\#(T)$, after which point we stop generating more summaries. Le et al. [2022] refer to this as depth and width of the analysis, respectively. Since each element of $[C]^\#(T)$ stands alone as a sound summary for the program $C$, then eliminating elements from the set will only preclude possible summaries without affecting the correctness of the existing ones. Even for correctness analyses, the bounded unrolling solution is reasonable and is used in other symbolic execution systems [Fragoso Santos et al. 2020; Holtzen et al. 2020].
**Loop invariants and partial correctness.** An alternative to bounded unrolling for analyzing loops in correctness applications is to use loop invariants. Loop invariants are not suitable for bug finding, since they only guarantee *partial correctness*—the postcondition holds if the program terminates, but it may diverge. We can alter the rule for while loops to the following.

\[
\text{\textbf{J}} \quad \text{while} \quad e \quad \text{do} \quad C
\]

\[
\# (T) = \{ (I, (ok : I \land \neg e) \lor \top) \mid (I \land e, ok : I) \in \# (T) \}
\]

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 \( \top \).

Finding loop invariants is generally undecidable, however techniques from abstract interpretation 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 as our own, but without outcome conjunctions. In the nondeterministic case, we can convert the outcome conjunctions into disjunctions since

\[
(ok : P) \oplus (ok : Q) \Rightarrow (ok : P \lor Q)
\]

and therefore we can use the same technique.

The probabilistic case is much more complicated since we need to address the question of almost sure termination. Automated techniques using ranking super-martingales exist [Agrawal et al. 2017], but an exploration of that approach is out of scope for this paper.

**Nondeterministic Allocation.** Many memory bugs in C arise 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. Since our language is generic over the execution model, we do not have a nondeterministic malloc operation, but rather only a deterministic alloc operation which is always guaranteed to succeed. We can add \( x := \text{malloc}() \) as syntactic sugar for \( (x := \text{alloc}()) + (x := \text{null}) \), and derive the following semantics:

\[
\text{\textbf{J}} \quad x := \text{malloc}() \quad K
\]

\[
\# (T) = \{ (x = X \land \text{emp}, (ok : x = \text{null} \land \text{emp}) \oplus (ok : \exists Y. x \mapsto Y)) \}
\]

**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 correctness and incorrectness. Many procedures in a given codebase will not include loops or effects, so their summaries are equally valid for both correctness and incorrectness, and also for use in programs with different interpretations of choice.

In other cases, where a procedure does have multiple outcomes, it is relatively 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 7.1.

\[
(\text{ok} : P) \quad C \quad \langle \psi_1 \oplus_a \psi_2 \rangle \quad \Rightarrow \quad (\text{ok} : P) \quad C \quad \langle \psi_1 \oplus a \rangle \quad \text{and} \quad (\text{ok} : P) \quad C \quad \langle \psi_2 \oplus \pi \rangle \quad (T)
\]

### 7 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.

#### 7.1 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 code 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.

```plaintext
main() :
  x ← [v];
push_back(v);
[x] ← 1
```

```
push_back(v) :
  y ← [v];
  free(y);
y := alloc();
[0] ← y
```

Before we can analyze the main procedure, we must store \([\text{push\_back}(v)](T)\) in the procedure table. Since \text{push\_back} is a common library function, it makes sense to compute summaries that describe all the outcomes so that we may reuse these summaries for both correctness and incorrectness analyses. The first step in doing so is to compute summaries for the two nondeterministic branches, which are both simple sequential programs.

\[
\begin{cases}
  y \leftarrow [v]; \\
  \text{free}(y); \\
  [v] \leftarrow y
\end{cases}
\]

\(\text{push\_back}(v)\) \# (T) = \{(ok : \exists B. v \mapsto B \mapsto \text{emp}) \⊕ (ok : \exists B \mapsto B \mapsto [v] \mapsto \text{emp}) \oplus \top\}\)

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.

\[
v \mapsto A * A \mapsto * \text{[emp]} \equiv [v \mapsto A * A \mapsto \text{emp}] \oplus [v \mapsto A * A \mapsto \text{emp}]
\]

So, by framing \text{emp} into the first branch and \(v \mapsto A * A \mapsto \text{emp} \oplus \text{emp} \oplus [v \mapsto A * A \mapsto \top]\) into the second branch, we get a summary for \text{push\_back} as a whole. This can similarly be done for the other summaries of the first branch, yielding the lookup table below.

\[
T = \begin{cases}
  \langle \text{ok} : v \mapsto A * A \mapsto \text{emp} \rangle \\
  \langle \text{ok} : v \mapsto A * A \mapsto \text{emp} \rangle
\end{cases}
\]

The first summary tells us that \text{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 \text{push\_back} itself can fail. We will focus on using the first summary to show how \text{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 \text{main}. The load on the first line has three summaries according to Figure 5, we select the first one in which \(v\) is allocated.

\[
\langle \text{ok} : x = X \land v \mapsto Y \rangle x \leftarrow [v] \langle \text{ok} : x = Y \land v \mapsto Y \rangle [x \leftarrow [v]](T)
\]

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

\[
\langle \text{ok} : v \mapsto A * A \mapsto \text{emp} \rangle \text{push\_back}(v) \langle \text{ok} : \exists B. v \mapsto B \mapsto \text{emp} \oplus \text{emp} \oplus [v \mapsto A * A \mapsto \top]\rangle
\]

Now, we use \text{seq} to sequentially compose these summaries, which involves bi-abducting the post-condition of \(x \leftarrow [v]\) with the precondition of \text{push\_back}(v).

\[
x = Y \land v \mapsto Y * [A = Y * x \mapsto \text{emp}] \lor v \mapsto A * A \mapsto \text{emp}
\]

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

\[
\langle \text{ok} : v \mapsto x * x \mapsto \text{emp} \rangle x \leftarrow [v] \langle \text{ok} : \exists B. v \mapsto B \mapsto \text{emp} \oplus \text{emp} \oplus [v \mapsto A * A \mapsto \top]\rangle
\]

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.

\[
\langle \text{ok} : v \mapsto x * x \mapsto \text{emp} \rangle x \leftarrow [v] \langle \text{ok} : \exists B. v \mapsto B \mapsto \text{emp} \oplus \text{emp} \oplus [v \mapsto A * A \mapsto \top]\rangle
\]
7.2 Consensus in Distributed Computing

The microservice architecture—in which many lightweight components communicate via fixed APIs—is becoming increasingly popular in software engineering. While microservices add flexibility and scale well, the fact that they communicate over a network introduces the possibility of failures at many points. Each microservice will typically publish a Service Level Agreement (SLA)—a contract with the downstream users conveying, for example, what percentages of service calls will succeed.

In this case study, we will show how OSL can be used to lower bound reliability rates of microservices. We use a basic consensus algorithm, shown below, in which each of three processes broadcasts a value \( v \) microservices. We use a basic consensus algorithm, shown below, in which each of three processes broadcasts a value \( v \) to the downstream users conveying, for example, what percentages of service calls will succeed.

Now, we can bi-abduce the first outcome above with the precondition for decide, 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.

\[
T = \left\{ \begin{array}{l}
\langle \text{ok} : \mu \mapsto V \rangle \text{ broadcast}(\mu, p) \ (\langle \text{ok} : \mu \mapsto V \land \nu = V \rangle \oplus_{0.99} (\text{er} : \mu \mapsto \neg \land \nu = V)) \\
\langle \text{ok} : p_1 \mapsto V_1 \land p_2 \mapsto V_2 \land p_3 \mapsto V_3 \land \nu \mapsto \neg \land V_1 = V_2 \rangle \text{ decide}(p_1, p_2, p_3, \nu) \ (\langle \text{ok} : \nu \mapsto V_1 \ast \cdots \rangle) \\
\end{array} \right. \\
\ldots
\]

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.

\[
\langle \text{ok} : v_1 = V_1 \rangle \ p_1 := \text{alloc}() \ \text{broadcast}(v_1, p_1) \ (\langle \text{ok} : v_1 = V_1 \land p_1 \mapsto V_1 \rangle \oplus_{0.99} \text{T})
\]

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

\[
\langle \text{ok} : v_1 = V_1 \land v_2 = V_2 \land v_3 = V_3 \land p_1 \mapsto V_1 \land p_2 \mapsto V_2 \land p_3 \mapsto V_3 \rangle \oplus_{0.99} \text{T}
\]

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

\[
\langle \text{ok} : v_1 = V_1 \land v_2 = V_2 \land v_3 = V_3 \land V_1 = V_2 \rangle \text{ main}() \ (\langle \text{ok} : \nu \mapsto V_1 \ast \cdots \rangle \oplus_{0.9703} \text{T})
\]

8 RELATED WORK

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 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.\textsuperscript{6}
(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 of the bi-abduction procedure of 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. Hence, our single path algorithm can be seen as an alternative model for Pulse, proving that it is sound for bug-finding using the OSL semantics rather than IL.

Pulse does differ from our symbolic execution algorithms in some other ways. For example, Pulse does not support inductive predicates (e.g., list segments) and therefore it has a simplified bi-abduction procedure, which is capable of handling more types of pure assertions. Though the soundness of our frame rule is based on the symbolic heaps of Berdine et al. [2005b], it is possible to add more types of pure assertions than equalities and inequalities (as shown by Baktiev [2006]). This would need to be accompanied by a more powerful bi-abduction procedure, such as that of Brotherston et al. [2017]. It would be interesting whether the inductive predicates offered by OSL could help analyses like Pulse generate more concise summaries, further aiding in their scalability.

Separation logic with effects. While standard separation logic relies on nondeterminism for a sound frame rule [Yang and O’Hearn 2002], local reasoning has been extended into other settings. Baktiev [2006] proved that the frame rule is sound in a deterministic language if heap assertions are unaffected by permutation of addresses. Similarly, Tatsuta et al. [2009] created a deterministic separation logic, but the frame rule only applies to programs that do not allocate memory.

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. As a tradeoff, frame baking introduces additional proof obligations whenever constructing a triple (e.g., it must be shown that every inference rule is frame preserving). In OSL, we preferred to keep the definition of triples simple so as to keep the proof of the frame rule self-contained.

In addition, there are separation logic variants that combine probabilistic computation and nondeterminism in order to recover a sound frame rule. Tassarotti and Harper [2019] introduced Polaris, a probabilistic variant of concurrent separation logic [O’Hearn 2004], and implemented it in Iris [Jung et al. 2015]. Batz et al. [2019] created Quantitative Separation Logic, which uses weakest pre-expectation [Morgan et al. 1996] style predicate transformers to derive expected values in probabilistic pointer programs. Our approach differs from these two developments in two ways. First, by opting for a basic execution model with only probabilistic choice (and without nondeterminism), we avoid a significant amount of complexity in the underlying semantics [Jones 1989; Varacca and Winskel 2006]. Second, our approach of relating probabilistic choices with program outcomes is particularly amenable to bi-abductive symbolic execution and generating re-usable procedure summaries, which was not a goal of the aforementioned developments.

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.

Algebraic Program Semantics Our algebraic definition of program semantics has similarities to Weighted Programming [Batz et al. 2022], however the goals of our development are different.

\textsuperscript{6} Re-allocation makes the program semantics non-local, since executions in a larger heap (with more deallocated pointers) add possible end states. Interestingly, re-allocation is local for under-approximate specifications, including in OSL. It would be an interesting to explore an OSL frame rule that is only sound with respect to specifications of the form \((\varphi) \ C \ (\psi \oplus_{\alpha} \tau)\).
Whereas we wished to use an algebraic interpretation of choice in order 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. Outcome Algebras are reminiscent of Effect Algebras [Foulis and Bennett 1994], which are used to reason about quantum programs. While there are subtle differences between these two definitions, it would be interesting to see if Outcome Algebras are able to capture quantum computation too.

Unified approaches to correctness and incorrectness. Maksimović et al. [2022] recently introduced Exact Separation Logic (ESL), which combines the semantics of SL and ISL in order to derive specifications for both correctness and incorrectness within a single program logic and is implemented inside the Gillian symbolic execution engine [Fragoso Santos et al. 2020]. They additionally show how inductive predicates can be compatible with the IL semantics, which is something that has not been demonstrated by Pulse or Pulse-X. Similarly, Bruni et al. [2021, 2023] introduced Local Completeness Logic (LCL), which 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.

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 [2019]; 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 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].

9 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.

With our introduction of Outcome Separation Logic, 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 can be used to reason about deterministic and probabilistic programs whereas separation logic and IL cannot.

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 safe preconditions) 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.

There are many opportunities for further developments. We plan to augment OSL to support pointer arithmetic based on Array Separation Logic [Brotherston et al. 2017], concurrency (by fusing OSL with Iris [Jung et al. 2015]), and quantum computation. On the applied side, we plan to implement our analysis algorithms to show the ease with which common infrastructure can be used for both correctness and incorrectness and also provide a testbed for experimenting with heuristics to determine when an analysis should abandon a verification attempt and instead switch to single-path mode to find bugs. The power and flexibility of OSL makes it the ideal logical foundation to study these questions.

REFERENCES


A PROGRAM SEMANTICS

We begin by providing definitions for completeness and Scott continuity, which are mentioned in Definition 3.4.

Definition A.1 (Complete Partial Semiring). A partial semiring \( \langle A, +, \cdot, 0, 1 \rangle \) 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 \( J_k \cap J_l = \emptyset \) if \( k \neq l \). 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.2 (Scott Continuity). Consider a semiring \( \langle A, +, \cdot, 0, 1 \rangle \) with partial order \( \leq \). A function (or partial function) \( f : A \rightarrow A \) 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 \( + \) 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 3.4.

Definition A.3 (Normalizable). A semiring \( \langle A, +, \cdot, 0, 1 \rangle \) is normalizable if for any well-defined sum \( \sum_{i \in I} a_i \), there exists \( (b_i)_{i \in I} \) such that \( \sum_{i \in I} b_i = 1 \) 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 \( W_A \) 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.4 (Row-Column Property). A monoid \( \langle A, +, 0 \rangle \) 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 \( (u_{i,j})_{i \in I \times J} \) 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.5. Let \( \langle A, +, \cdot, 0, 1 \rangle \) be a normalizable semiring (Definition A.3), then \( \langle A, +, 0 \rangle \) has the row-column property (Definition A.4).

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 \( \sum_{i \in I} a'_i = \sum_{j \in J} b'_j = 1 \) 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:

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

And

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

\[\square\]

In fact, it is known that if some monoid \( A \) has the row-column property, then the functor \( F_A \) 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 if the functor preserves weak pullbacks [Kurz and Velebil 2016]. Combining these two results, we get that relations lifted by \( F_A \) 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 3.4 we require that sup(A) = 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 $\langle \mathbb{R}^\infty, \min, +, \infty, 0 \rangle$, which can be used to encode optimization problems [Batz et al. 2022].

The fact that sup(A) = 1 is used in order to define Rep as a lifted relation (Appendix C.3), and it is also used in the proof of Lemma C.7. 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.6 (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 $y_i \leq$ sup(A) for each $i \in I$. By the definition of $\leq$, 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:

$$ = \sum_{i \in I} x_i \cdot y_i + x_i \cdot y'_i $$

$$ = \sum_{i \in I} x_i \cdot y_i + \sum_{i \in I} x_i \cdot y'_i $$

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

Lemma A.7 (Totality of Bind). The bind function defined in Definition 3.8 is a total function (this is not immediate, since it uses partial addition).

Proof. First, we note that $\sum_{a \in $ supp$(m) } m(a)$ must be defined by the definition of $\mathcal{W}$. By Lemma A.6, we know that $\sum_{a \in $ supp$(m) } m(a) \cdot f(a)(b)$ must be defined too, for any family $(f(a)(b))_{a \in $ supp$(m)}$.

Now, since bind$(m, f)(b) = \sum_{a \in $ 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 $\text{supp}($bind$(m, f)) = \cup_{a \in $ supp$(m) } $ supp$(f(a))$. \qed

Theorem A.8 (Fixed Point Existence). The function $F_{(C, e)}$ defined above has a least fixed point.

Proof. It will suffice to show that $F_{(C, e)}$ 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 $f_1 \sqsubseteq f_2$ 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.

$$ \sup_{f \in D} \text{bind}(m, f) = \sup_{f \in D} \sum_{s \in $ supp$(m) } m(s) \cdot \begin{cases} f(a) & \text{if } s = 1_R(a) \\ \text{unit}_M(s) & \text{if } s = 1_L(-) \end{cases} $$
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.

\[
\begin{align*}
\sum_{s \in \text{supp}(m)} m(s) \cdot \sup_{f \in D} f(a) &= \begin{cases} 
\sup_{f \in D} f(a) & \text{if } s = 1_R(a) \\
\text{unit}_M(s) & \text{if } s = 1_L(-)
\end{cases} \\
\sum_{s \in \text{supp}(m)} m(s) \cdot (\sup D)(a) &= \begin{cases} 
(\sup D)(a) & \text{if } s = 1_R(a) \\
\text{unit}_M(s) & \text{if } s = 1_L(-)
\end{cases} \\
= \text{bind}(m, \sup D)
\end{align*}
\]

Finally, we show that \( F_{(C,e)} \) is Scott continuous with respect to the order defined above.

\[
\sup_{f \in D} F_{(C,e)}(f) = \lambda(s,h) . \sup_{f \in D} F_{(C,e)}(f)(s,h)
\]

\[
= \lambda(s,h) . \sup_{f \in D} \begin{cases} 
\text{bind}([C](s,h), f) & \text{if } [e](s) = \text{true} \\
\text{unit}(s,h) & \text{if } [e](s) = \text{false}
\end{cases}
\]

\[
= \lambda(s,h) . \sup_{f \in D} \begin{cases} 
\text{bind}([C](s,h), f) & \text{if } [e](s) = \text{true} \\
\text{unit}(s,h) & \text{if } [e](s) = \text{false}
\end{cases}
\]

\[
= \lambda(s,h) . \text{bind}([C](s,h), \sup D)
\]

\[
= F_{(C,e)}(\sup D)
\]

\[\square\]

**Theorem A.9 (Totality of Program Semantics).** If all expressions used in guards are Boolean valued, then the semantics \([C]\) is a total function.

**Proof.** By induction on the structure of \( C \). The cases for skip and all atoms are trivial.

- \( C = C_1 ; C_2 \). By the induction hypothesis, we assume \([C_1]\) and \([C_2]\) are total, and by Lemma A.7 we know that bind is total, therefore \( \text{bind}([C_1](s,h), [C_2]) \) is total.
- \( C = C_1 +_a C_2 \). This syntax can only be used if \( \overline{a} \) is defined. Now, we know that \( a + \overline{a} = 1 \). By Lemma A.6, we conclude that \( a \cdot [C_1](s,h) + \overline{a} \cdot [C_2](s,h) \) is defined too.
- \( C = \text{if } e \text{ then } C_1 \text{ else } C_2 \). By the induction hypothesis, we assume \([C_1]\) and \([C_2]\) are total, therefore \([\text{if } e \text{ then } C_1 \text{ else } C_2]\) is total.
- \( C = \text{while } e \text{ do } C \). Follows from Theorem A.8

\[\square\]

**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:

\[
m_1 = \lambda x . \sum_{y \in \text{supp}(m_2)} m(x,y) \quad \text{and} \quad m_2 = \lambda y . \sum_{x \in \text{supp}(m_1)} m(x,y)
\]

Now, we have:

\[
|m_1| = |\lambda x . \sum_{y \in \text{supp}(m_2)} m(x,y)|
\]

\[
= \sum_{x \in \text{supp}(m_1)} \sum_{y \in \text{supp}(m_2)} m(x,y)
\]
Therefore \( (\Rightarrow) \) We know there must be some \( m' \) such that \( m = \lambda y. \sum_{x \in \text{supp}(0)} m'(x, y) \), but since \( \text{supp}(\emptyset) = \emptyset \), then \( m = \emptyset \).

\( (\Leftarrow) \) Let \( m' = \emptyset \), so clearly \( m' \in \mathcal{W}_R \), and we also have \( \lambda x. \sum_{y \in \text{supp}(\emptyset)} (0(x, y)) = \emptyset \) and \( \lambda y. \sum_{x \in \text{supp}(\emptyset)} (0(x, y)) = \emptyset \)

\( \square \)

**Lemma B.3.** If \( |m_1| + |m_2| \) is defined, then \( m_1 + m_2 \) is defined.

**Proof.** Observe that:

\[ |m_1| + |m_2| = \sum_{x \in \text{supp}(m_1)} m_1(x) + \sum_{x \in \text{supp}(m_2)} m_2(x) \]

By associativity:

\[ = \sum_{x \in \text{supp}(m_1) \cup \text{supp}(m_2)} m_1(x) + m_2(x) \]

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. \( \square \)

**Lemma B.4.** \( (m_1 + m_2, m') \in \mathcal{R} \) iff there exist \( m'_1 \) and \( m'_2 \) such that \( m' = m'_1 + m'_2 \) and \( (m_1, m'_1) \in \mathcal{R} \) and \( (m_2, m'_2) \in \mathcal{R} \).

**Proof.**

\( (\Rightarrow) \) We know that there are some \( m \) such that \( m_1 + m_2 = \lambda x. \sum_{x \in \text{supp}(m')} m(x, y) \) and \( m' = \lambda y. \sum_{x \in \text{supp}(m')} m(x, y) \). So, for each \( x \), we have that \( m_1(x) + m_2(x) = \sum_{x \in \text{supp}(m')} m(x, y) \).

Using Lemma A.5, we know there must be \( (x_k)_{k \in \{1,2\}} \times \text{supp}(m') \) such that \( m_1(x) = \sum_{x \in \text{supp}(m')} x(i, y) \) for \( i \in \{1,2\} \) and \( m(x, y) = x_{(1,y)} + x_{(2,y)} \) for each \( y \in \text{supp}(m') \). Now let \( m'_1(x, y) = x_{(1,y)} \) and \( m'_2(x, y) = x_{(2,y)} \) and let \( m'_1 = \lambda y. \sum_{x \in \text{supp}(m_1)} m'_1(x, y) \) and \( m'_2 = \lambda y. \sum_{x \in \text{supp}(m_2)} m'_2(x, y) \).

First, we establish that \( m'_1 + m'_2 = m' \):

\[ m'_1 + m'_2 = \lambda y. \sum_{x \in \text{supp}(m_1)} m'_1(x, y) + \sum_{x \in \text{supp}(m_2)} m'_2(x, y) = \lambda y. \sum_{x \in \text{supp}(m_1)} x_{(1,y)} + \sum_{x \in \text{supp}(m_2)} x_{(2,y)} \]

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

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

Now, observe that:

\[ \lambda x. \sum_{y \in \text{supp}(m')} m'_1(x, y) = \lambda x. \sum_{y \in \text{supp}(m')} x_{(i,y)} \]
Now, for any \( y \notin \text{supp}(m'_i) \), it must be the case that \( m'_i(y) = 0 \) and so \( \sum_{x \in \text{supp}(m_i)} x(i, y) = 0 \), so each \( x(i, y) = 0 \). This means we can expand the sum to be over \( \text{supp}(m'_1 + m'_2) = \text{supp}(m') \).

\[
\lambda x. \sum_{y \in \text{supp}(m')} x(i, y) = m_i
\]

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

\((\Leftarrow)\) 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 \( m_1 = \lambda x. \sum_{y \in \text{supp}(m'_i)} m''_i(x, y) \) and \( m'_i = \lambda y. \sum_{x \in \text{supp}(m_i)} m''_i(x, y) \) 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:

\[
\lambda x. \sum_{y \in \text{supp}(m')} m''_i(x, y) = \lambda x. \sum_{y \in \text{supp}(m'_i)} m''_i(x, y) = m_1 + m_2
\]

\[
\lambda y. \sum_{x \in \text{supp}(m'_1 + m'_2)} m''_i(x, y) = \lambda y. \sum_{x \in \text{supp}(m'_1)} m''_i(x, y) + \sum_{x \in \text{supp}(m'_2)} m''_i(x, y) = m'_1 + m'_2 = m'
\]

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

\[\square\]

**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 \( a \cdot m_1 = \lambda x. \sum_{y \in \text{supp}(m_2)} m(x, y) \) and \( m_2 = \lambda y. \sum_{x \in \text{supp}(a \cdot m_1)} m(x, y) \). This means that for all \( x \):

\[
a \cdot m_1(x) = \sum_{y \in \text{supp}(m_2)} m(x, y)
\]

By Definition A.3, we can obtain \( (b(x, y))_{y \in \text{supp}(m_2)} \) such that \( \sum_{y \in \text{supp}(m_2)} b(x, y) = 1 \) and \( m(x, y) = (\sum_{z \in \text{supp}(m_2)} m(x, z)) \cdot b(x, y) \) for all \( y \in \text{supp}(m_2) \). Now, define \( m''_1(x, y) = m_1(x) \cdot b(x, y) \) and \( m'_2(y) = \sum_{x \in \text{supp}(m_1)} m''_1(x, y) \). We now show that \( m_2 = a \cdot m'_2 \):

\[
a \cdot m'_2 = \lambda y. a \cdot \sum_{x \in \text{supp}(m_1)} m''_1(x, y)
\]

\[
= \lambda y. a \cdot \sum_{x \in \text{supp}(m_1)} m_1(x) \cdot b(x, y)
\]

\[
= \lambda y. \sum_{x \in \text{supp}(m_1)} a \cdot m_1(x) \cdot b(x, y)
\]

\[
= \lambda y. \sum_{x \in \text{supp}(m_1)} (\sum_{z \in \text{supp}(m_2)} m(x, z)) \cdot b(x, y)
\]

\[
= \lambda y. \sum_{x \in \text{supp}(m_1)} m(x, y) = m_2
\]

We also have:

\[
\lambda x. \sum_{y \in \text{supp}(m'_2)} m''_1(x, y) = \lambda x. \sum_{y \in \text{supp}(m'_2)} m_1(x) \cdot b(x, y)
\]

Since we already showed that \( m_2 = a \cdot m'_2 \), then it must be the case that \( \text{supp}(m_2) = \text{supp}(m'_2) \).

\[
= \lambda x. m_1(x) \cdot \sum_{y \in \text{supp}(m_2)} b(x, y) = \lambda x. m_1(x) \cdot 1 = m_1
\]
And clearly $m'_2 = \lambda y \cdot \sum_{x \in supp(m_1)} m''(x, y)$ by definition, so $(m_1, m'_2) \in \overline{R}$.

\[(\Leftarrow)\] By the definition of relation lifting, there is some $m$ such that $m_1 = \lambda x \cdot \sum_{y \in supp(m_1)} m(x, y)$ and $m'_2 = \lambda y \cdot \sum_{x \in supp(m_1)} m(x, y)$. Now, let $m' = a \cdot m$, so this clearly means that $a \cdot m_1 = \lambda x \cdot \sum_{y \in supp(a \cdot m'_1)} m'(x, y)$ and $a \cdot m'_2 = \lambda y \cdot \sum_{x \in supp(a \cdot m'_1)} m'(x, y)$, so $(a \cdot m_1, a \cdot m'_2) \in \overline{R}$.

\[\square\]

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

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

\[\lambda x' \cdot \sum_{y' \in supp(\text{unit}(y))} m(x', y') = \lambda x' \cdot \text{unit}(x, y)(x', y) = \text{unit}(x)\]

And similarly, $\lambda y' \cdot \sum_{x' \in supp(\text{unit}(x))} m(x', y') = \text{unit}(y)$, so $(\text{unit}(x), \text{unit}(y)) \in \overline{R}$. \[\square\]

**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 $m_1 = \lambda x \cdot \sum_{z \in supp(m_2)} m(x, z)$ and $m_2 = \lambda y \cdot \sum_{x \in supp(m_1)} m(x, z)$. This means that for each $(x, z) \in supp(m)$, 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:

\[m'(y, z) = \begin{cases} m_2(z) & \text{if } (y, z) \in S' \\ 0 & \text{if } (y, z) \notin S' \end{cases}\]

\[m''(y, z) = \sum_{z \in supp(m_2)} m(x, z)\]

And now, we have the following:

\[\lambda x \cdot \sum_{y \in supp(m_1)} m''(x, y) = \lambda x \cdot \sum_{y \in supp(m_1) \cap supp(m_2) \cap (y, z) \in S'} m(x, z)\]

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

\[\lambda y \cdot \sum_{x \in supp(m_1)} m''(x, y) = \lambda y \cdot \sum_{x \in supp(m_1) \cap supp(m_2) \cap (y, z) \in S'} m(x, z)\]

\[\lambda y \cdot \sum_{x \in supp(m_1)} m''(x, y) = \lambda y \cdot \sum_{x \in supp(m_1) \cap supp(m_2) \cap (y, z) \in S'} m(x, z)\]

\[= \lambda y \cdot \sum_{z \in supp(m_2) \cap (y, z) \in S'} m_2(z)\]

\[= \sum_{z \in supp(m_2) \cap (y, z) \in S'} m_2(z)\]

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

**Lemma B.8.** For any pair of directed chains $m_{(i, 1)} \subseteq m_{(i, 2)} \subseteq \cdots$ 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 $m_{(1,n)} = \lambda x. \sum_{y \in \text{supp}(m_{(1,n)})} m_n(x,y)$ and $m_{(2,n)} = \lambda y. \sum_{x \in \text{supp}(m_{(1,n)})} m_n(x,y)$. Since each $(m_{(i,n)})_{n \in \mathbb{N}}$ is a chain, $\sup_{n \in \mathbb{N}} m_{(i,n)}$ exists, and:

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

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.

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

so:

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

C. OUTCOME SEPARATION LOGIC

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

Proof. By property (3) of Definition 3.4, there must be $(b_s)_{s \in \text{supp}(m)}$ such that $m(s) = (\sum_{t \in \text{supp}(m)} m(t)) \cdot b_s$ and $\sum_{s \in \text{supp}(m)} b_s = 1$. Now, let $m'$ be defined as follows:

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

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

Lemma C.2 (Splitting). If $|m'| \leq a \cdot |m_1| + \bar{a} \cdot |m_2|$, then there exist $m'_1$ and $m'_2$ such that $|m'_1| \leq |m_1|$ and $|m'_2| \leq |m_2|$ and $m' = a \cdot m'_1 + \bar{a} \cdot m'_2$.

Proof. Since $a \cdot |m_1| + \bar{a} \cdot |m_2| \geq |m'|$, then there must be some $b$ such that $a \cdot |m_1| + \bar{a} \cdot |m_2| = |m'| + b$. and therefore there is also an $a'$ such that:

$$|m'| = (a' \cdot b) \cdot a' = (a \cdot |m_1| + \bar{a} \cdot |m_2|) \cdot a'$$

By Lemma C.1, we know there must be an $m''$ such that $|m''| = 1$ and $m' = |m'| \cdot m''$. Now, let $m'_1 = |m_1| \cdot a' \cdot m''$ and $m'_2 = |m_2| \cdot a' \cdot m''$. So, now we have:

$$m' = |m'| \cdot m'' = (a \cdot |m_1| + \bar{a} \cdot |m_2|) \cdot a' \cdot m'' = a \cdot |m_1| \cdot a' \cdot m'' + \bar{a} \cdot |m_2| \cdot a' \cdot m'' = a \cdot m'_1 + \bar{a} \cdot m'_2$$

Recall that $a + \bar{a} = 1$ and so $a \leq 1$ and $\bar{a} \leq 1$.

$$|m'_1| = |m_1| \cdot a' \cdot |m''| = |m_1| \cdot a' \cdot |m'| = |m_1| \cdot a \cdot 1 \leq |m_1|$$

By a symmetric argument, $|m'_2| \leq |m_2|$. \hfill \square

C.1 The Outcome Separating Conjunction

Lemma 4.3. If $m \models \varphi$ and $(m, m') \in \overline{\text{frame}(F)}$, then $m' \not\models \varphi \otimes F$

Proof. By induction on the structure of $\varphi$.

$\varphi = \mathbb{T}$. Since $\varphi \otimes F = \mathbb{T}$, then clearly $m' \not\models \varphi \otimes F$

$\varphi = \mathbb{T}^\circ$. This means that $m = 0$ and therefore by Lemma B.2 $m' = 0$ as well. Since $\varphi \otimes F = \mathbb{T}^\circ$, then clearly $m' \not\models \varphi \otimes F$. 

\( \varphi = \varphi_1 \lor \varphi_2 \). 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 \( m' \vDash \varphi_1 \circ F \). We can therefore weaken this to conclude that \( m' \vDash (\varphi_1 \lor \varphi_2) \circ F \). 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 \( m = a \cdot m_1 + \bar{a} \cdot m_2 \). Now, since \( (a \cdot m_1 + \bar{a} \cdot m_2, m') \in \text{frame}(F) \), by Lemmas B.4 and B.5 there must be \( m'_1 \) and \( m'_2 \) such that \( (m_1, m'_1) \in \text{frame}(F) \) and \( (m_2, m'_2) \in \text{frame}(F) \) and \( m' = a \cdot m'_1 + \bar{a} \cdot m'_2 \). By the induction hypothesis, \( m'_1 \vDash \varphi_1 \circ F \) and \( m'_2 \vDash \varphi_2 \circ F \), so \( m' \vDash (\varphi_1 \oplus \varphi_2) \circ F \).

\( \varphi = \varepsilon : P \). We know that \( |m| = 1 \) and every \( \sigma \in \text{supp}(m) \) has the form \( \mathbb{L}_e(s, h) \) such that \( (s, h) \vDash P \). Since \( (m, m') \in \text{frame}(F) \), we know by Lemma B.1 that \( |m'| = |m| = 1 \). Additionally, for every element in \( \text{supp}(m') \), there must be an element in \( \text{supp}(m) \) related by \( \text{frame}(F) \), so each element of \( m' \) has the form \( \mathbb{L}_e(s, h \cup h') \) such that \( (s, h) \vDash P \) and \( (s, h') \vDash P \), and so clearly \( (s, h \cup h') \vDash P \ast P \), and therefore also \( m' \vDash (\varepsilon : P) \circ F \).

\( \square \)

**Lemma 4.4.** If \( m \vDash \varphi \circ F \), then there exist \( m_1, m'_1 \) and \( m_2, m'_2 \) such that \( (m_1, m'_1) \in \text{frame}(F), m = m'_1 + m_2 \) 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 \( \varphi \).

\( \varphi = \top \). Suppose \( m \vDash \top \circ F \). Let \( m_1 = m'_1 = 0 \) and \( m_2 = m \). Clearly \((0, 0) \in \text{frame}(F) \) and \( m'_1 + m_2 = 0 + m = m \). Now, taking any \( m'_2 \), it is obvious that \( 0 + m'_2 \vDash \top \).

\( \varphi = \top^\circ \). Suppose \( m \vDash \top^\circ \circ F \), so \( m = 0 \). Let \( m_1 = m'_1 = m_2 = 0 \). Clearly \((0, 0) \in \text{frame}(F) \) and \( m'_1 + m_2 = 0 + 0 = 0 = m \). Now, take any \( m'_2 \) such that \( |m'_2| \leq |m_2| = 0 \). This means that \( m'_2 = 0 \), and clearly \( 0 \vDash \top^\circ \).

\( \varphi = \varphi_1 \lor \varphi_2 \). We know \( m \vDash \varphi_1 \circ F \) or \( m \vDash \varphi_2 \circ F \). Without loss of generality, suppose that \( m \vDash \varphi_1 \). By the induction hypothesis, there are \( m_1, m'_1 \) and \( m_2, m'_2 \) such that \( (m_1, m'_1) \in \text{frame}(F) \) 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 \( m_1 + m'_2 \vDash \varphi_1 \lor \varphi_2 \).

\( \varphi = \varphi_1 \oplus \varphi_2 \). We know that \( m_1 \) and \( m_2 \) such that \( m_1 \vDash \varphi_1 \circ F \) and \( m_2 \vDash \varphi_2 \circ F \) and \( m = a \cdot m_1 + \bar{a} \cdot m_2 \). 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 \((u_1, u'_1) \in \text{frame}(F) \) and \((v_1, v'_1) \in \text{frame}(F) \) and \( u_1 + u'_1 \vDash \varphi_1 \) and \( v_1 + v'_1 \vDash \varphi_2 \) whenever \( |u'_2| \leq |u_2| \) and \( |v'_2| \leq |v_2| \).

Now, let \( m_1 = a \cdot u_1 + \bar{a} \cdot v_1 \) and \( m'_2 = a \cdot u'_1 + \bar{a} \cdot v'_1 \) and \( m_2 = a \cdot u_2 + \bar{a} \cdot v_2 \). By Lemmas B.4 and B.5, we get that \((m_1, m'_1) \in \text{frame}(F) \). We also have that:

\[
\begin{align*}
    m'_1 + m_2 &= (a \cdot u'_1 + \bar{a} \cdot v'_1) + (a \cdot u_2 + \bar{a} \cdot v_2) = a \cdot (u'_1 + u_2) + \bar{a} \cdot (v'_1 + v_2) = a \cdot m_1 + \bar{a} \cdot m_2 = m
\end{align*}
\]

Now, take any \( m'_2 \) such that \( |m'_2| \leq |m_2| = a \cdot |u_2| + \bar{a} \cdot |v_2| \). 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 \( m_2 = a \cdot u'_2 + \bar{a} \cdot v'_2 \). This means that \( u_1 + u'_2 \vDash \varphi_1 \) and \( v_1 + v'_2 \vDash \varphi_2 \). We also have:

\[
\begin{align*}
    a \cdot (u_1 + u'_2) + \bar{a} \cdot (v_1 + v'_2) &= (a \cdot u_1 + \bar{a} \cdot v_1) + (a \cdot u'_2 + \bar{a} \cdot v'_2) = m_1 + m'_2
\end{align*}
\]

So, \( m_1 + m'_2 \vDash \varphi_1 \oplus \varphi_2 \).

\( \varphi = \varepsilon : P \). We know that \( m \vDash \varepsilon : P \ast P \), so \( |m| = 1 \) and every element of \( \text{supp}(m) \) has the form \( \mathbb{L}_e(s, h \cup h') \) such that \( (s, h) \vDash P \) and \( (s, h') \vDash P \). Now, let \( m_2 = 0 \), \( m'_1 = m \), so clearly \( m = m'_1 + m_2 \). To define \( m_1 \), first we fix a relation \( S \subseteq \text{frame}(F) \) such that for any \( h'' \) such that \( (s, h'') \vDash P \ast P \), there is a unique \( h \) and \( h' \) such that \( h'' = h \cup h' \) and \( (\mathbb{L}_e(s, h), \mathbb{L}_e(s, h \cup h')) \in S \). Now, \( m_1 \) is defined as follows:

\[
m_1(\sigma) = \sum_{\tau \mid (\sigma, \tau) \in S} m(\tau)
\]
Now, we must show that \((m_1, m) \in \text{frame}(F)\). To do so, first let:

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

Now, we have:

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

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

\[
= \lambda \tau. m(\tau) = m
\]

So, \((m_1, m) \in \text{frame}(F)\), and this also means that \(|m_1| = |m| = 1\). Also, by construction, each element of \(\text{supp}(m_1)\) has the form \(\hat{\iota}_e(s, h)\) where \((s, h) \vDash P\), so \(m_1 \vDash e : P\). Since \(m_2 = 0\), then it follows that \(m_1 + m'_2 \vDash e : P\) for any \(m'_2\) such that \(|m'_2| \leq |m_2| = 0\).

\[\square\]

C.2 Nominal Heaps

In this section, we prove that outcome assertions based on symbolic heaps are invariant to address permutation. Practically speaking, this means that symbolic heaps cannot use addresses as constants or do pointer arithmetic. This approach to proving the soundness of the frame rule was suggested by Yang and O’Hearn [2002, §4.1], and has been explored by Baktiev [2006], who created a deterministic separation logic with a sound frame rule. Building on that previous work, we extend the idea to outcome assertions, so as to reason about, e.g., probabilistic programs too. We also show that the symbolic heaps of Berdine et al. [2005b] have the equivariance property, so the OSL frame rule is sound using the same basic assertions as were used in previous bi-abduction algorithms [Calcagno et al. 2009, 2011].

Now, let \(\pi_{\text{Addr}} : \text{Addr} \rightarrow \text{Addr}\) be a bijection that permutes the addresses. We can extend it to arbitrary values as follows:

\[
\pi_{\text{Val}}(v) = \begin{cases} \pi_{\text{Addr}}(v) & \text{if } v \in \text{Addr} \\ v & \text{if } v \not\in \text{Addr} \end{cases}
\]

And clearly \(\pi_{\text{Val}}\) is still a bijection. We also extend permutations to stores and heaps as follows:

\[
\pi_{\text{S}}(s) = \pi_{\text{Val}} \circ s \quad \pi_{\text{H}}(h) = \pi_{\text{Val}} \circ h \circ \pi_{\text{Addr}}^{-1}
\]

And finally, we extend permutations to program states as well:

\[
\pi_{\text{St}}(\hat{\iota}_e(s, h)) = \hat{\iota}_e(\pi_{\text{S}}(s), \pi_{\text{H}}(h)) \quad \pi_{\text{St}}(\text{undef}) = \text{undef}
\]

In the remainder of this section, we will drop the subscripts and refer to all the permutation functions above simply as \(\pi\). We will prove that permuting heap addresses via some permutation \(\pi\) does not affect the truth of symbolic heaps and outcome assertions.

**Lemma C.3 (Equivariance of Expressions).** For any \(e \in \text{Exp}, s \in S\), and permutation \(\pi\):

\[
\llbracket e \rrbracket (\pi(s)) = \pi(\llbracket e \rrbracket (s))
\]

**Proof.** By induction on the expression \(e\).

\[\triangleright e = x.\]

\[
\llbracket x \rrbracket (\pi(s)) = \pi(s(x)) = (\pi \circ s)(x) = \pi(s(x)) = \pi(\llbracket x \rrbracket (s))
\]
Lemma C.5 (Equivariance of Spatial Assertions).

If $e = X$. Same as the previous case.

If $e = \kappa$. Note that $\kappa \notin \text{Addr}$ since $\kappa \in \text{Const}$ and $\text{Const} \cap \text{Addr} = \emptyset$, so $\pi(\kappa) = \kappa$, and therefore $\llbracket \kappa \rrbracket (\pi(s)) = \kappa = \pi(\kappa) = \pi(\llbracket \kappa \rrbracket (s))$

$e_1 = e_2$.

$\llbracket e_1 = e_2 \rrbracket (\pi(s)) = \begin{cases} 
\text{true} & \text{if } \llbracket e_1 \rrbracket (\pi(s)) = \llbracket e_2 \rrbracket (\pi(s)) \\
\text{false} & \text{if } \llbracket e_1 \rrbracket (\pi(s)) \neq \llbracket e_2 \rrbracket (\pi(s)) 
\end{cases}$

By the induction hypothesis:

$= \begin{cases} 
\text{true} & \text{if } \pi(\llbracket e_1 \rrbracket (s)) = \pi(\llbracket e_2 \rrbracket (s)) \\
\text{false} & \text{if } \pi(\llbracket e_1 \rrbracket (s)) \neq \pi(\llbracket e_2 \rrbracket (s)) 
\end{cases}$

Since $\pi$ is a bijection, $\pi(x) = \pi(y)$ iff $x = y$, so:

$= \begin{cases} 
\text{true} & \text{if } \llbracket e_1 \rrbracket (s) = \llbracket e_2 \rrbracket (s) \\
\text{false} & \text{if } \llbracket e_1 \rrbracket (s) \neq \llbracket e_2 \rrbracket (s) 
\end{cases}$

$= \llbracket e_1 = e_2 \rrbracket (s)$

Finally, $\pi$ has no effect on Boolean values.

$= \pi(\llbracket e_1 = e_2 \rrbracket (s))$

$e = \neg e'$. By the induction hypothesis, $\llbracket e' \rrbracket (\pi(s)) = \pi(\llbracket e' \rrbracket (s))$. Clearly, logically negating both will preserve equality.

Lemma C.4 (Equivariance of Pure Assertions). If $s \models \Pi$, then $\pi(s) \models \Pi$.

Proof. By induction on $\Pi$.

If $\Pi = \text{true}$. Trivial since $\pi(s) \models \text{true}$ always holds.

If $\Pi = \Pi_1 \land \Pi_2$. We know that $s \models \Pi_1$ and $s \models \Pi_2$. By the induction hypothesis, $\pi(s) \models \Pi_1$ and $\pi(s) \models \Pi_2$, so $\pi(s) \models \Pi_1 \land \Pi_2$.

If $\Pi = e$. Suppose $s \models e$, so that means that $\llbracket e \rrbracket (s) = \text{true}$. By Lemma C.3, we also know that $\llbracket e \rrbracket (\pi(s)) = \pi(\llbracket e \rrbracket (s)) = \pi(\text{true}) = \text{true}$, so $\pi(s) \models e$.

Lemma C.5 (Equivariance of Spacial Assertions). If $(s, h) \models \Sigma$, then $\pi(s, h) \models \Sigma$.

Proof. By induction on $\Sigma$.

If $\Sigma = \text{true}$. Trivial since $\pi(s, h) \models \text{true}$ always holds.

If $\Sigma = \text{emp}$. We know that $h = \emptyset$, therefore $\pi(h) = \emptyset$ too, so $\pi(s, h) \models \text{emp}$.

If $\Sigma = \Sigma_1 * \Sigma_2$. We know that $(s, h_1) \models \Sigma_1$ and $(s, h_2) \models \Sigma_2$ such that $h = h_1 \uplus h_2$. By the induction hypothesis, $\pi(s, h_1) \models \Sigma_1$ and $\pi(s, h_2) \models \Sigma_2$. Since permuting distributes over $\uplus$, this means that $\pi(h) = \pi(h_1 \uplus h_2) = \pi(h_1) \uplus \pi(h_2)$, and so $\pi(s, h) \models \Sigma_1 * \Sigma_2$.

If $\Sigma = e_1 \mapsto e_2$. We know that $h$ is a singleton heap where the address $\llbracket e_1 \rrbracket (s)$ points to the value $\llbracket e_2 \rrbracket (s)$. Now:

$\pi(h) = \pi \circ h \circ \pi^{-1}$

$= \pi \circ (\lambda t. \llbracket e_2 \rrbracket (s)) \circ \pi^{-1}$

$= (\lambda t. \pi(\llbracket e_2 \rrbracket (s)) \circ \pi^{-1} \circ \pi (\llbracket e_1 \rrbracket (s))$
By Lemma C.3:
\[ \pi(h) = \lambda \ell. [e_2] (\pi(s)) \quad \text{if} \quad \ell = [e_1] (\pi(s)) \]

So \( \pi(h) \) is a singleton heap where the address \([e_1] (\pi(s))\) points to the value \([e_2] (\pi(s))\), and therefore \( \pi(s, h) \vdash e_1 \mapsto e_2 \).

\( \Sigma = Is(e_1, e_2) \). The proof is by induction on the length of the list segment. Suppose the list segment has length zero, then it must be the case that \( (s, h) \vdash e_1 = e_2 \land \text{emp} \). We know that this assertion is invariant to renaming from the previous cases and Lemmas C.4 and C.6. Now, suppose that the claim holds for a list of length \( n \), and the current list has length \( n + 1 \). That means that \( (s, h) \vdash \exists X. e_1 \mapsto X * Is(X, e_2) \). By the previous cases and the induction hypothesis, the claim holds.

\( \square \)

**Lemma C.6 (Equivariance of Symbolic Heaps).** If \( (s, h) \vdash P \), then \( \pi(s, h) \vdash P \) for any \( \pi \).

**Proof.** By definition, \( P \) must have the form \( P = \exists \bar{X}. \Pi \land \Sigma \), so if \( (s, h) \vdash \exists \bar{X}. \Pi \land \Sigma \), then we know that \( (s', h) \vdash \Pi \) and \( (s', h) \vdash \Sigma \) where \( s' = s[\bar{X} \mapsto \bar{v}] \) for some \( \bar{v} \). Using Lemmas C.4 and C.5, we get that \( \pi(s', h) \vdash \Pi \) and \( \pi(s', h) \vdash \Sigma \). Now, we can see that \( \pi(s') = \pi(s[\bar{X} \mapsto \bar{v}]) = \pi(s)[\bar{X} \mapsto \pi(\bar{v})] \), so \( \pi(s, h) \vdash \exists \bar{X}. \Pi \land \Sigma \).

\( \square \)

**Lemma 4.5 (Equivariance).** If \( m \vdash \varphi \) and \( (m, m') \in \text{Perm} \), then \( m' \vdash \varphi \).

**Proof.** By induction on the assertion \( \varphi \).

\( \varphi = \top \). Clearly \( m' \vdash \top \) since \( \top \) is always true.

\( \varphi = \top^\oplus \). We know that \( m \vdash \top^\oplus \), so \( m = \emptyset \) and therefore \( m' = \emptyset \) as well (by Lemma B.2), so \( m' \vdash \top^\oplus \).

\( \varphi = \varphi_1 \lor \varphi_2 \). Without loss of generality, suppose \( m \vdash \varphi_1 \). By the induction hypothesis, \( m' \vdash \varphi_1 \). We can now weaken this assertion to conclude that \( m' \vdash \varphi_1 \lor \varphi_2 \).

\( \varphi = \varphi_1 \oplus_a \varphi_2 \). We know that \( m_1 \vdash \varphi_1 \) and \( m_2 \vdash \varphi_2 \) for some \( m_1 \) and \( m_2 \) such that \( a \cdot m_1 + \bar{a} \cdot m_2 = m \). Since \( (a \cdot m_1 + \bar{a} \cdot m_2, m') \in \text{Perm} \), by Lemmas B.4 and B.5 we know that \( m' = a \cdot m_1 + \bar{a} \cdot m_2' \) for some \( m_1' \) and \( m_2' \) such that \( (m_1', m_2') \in \text{Perm} \) and \( (m_2, m_2') \in \text{Perm} \). By the induction hypotheses, we get that \( m_1' \vdash \varphi_1 \) and \( m_2' \vdash \varphi_2 \). Now, putting these together, we get \( m' \vdash \varphi_1 \oplus_a \varphi_2 \).

\( \varphi = (\varepsilon : P) \). We know that \( |m| = 1 \) and the entire support of \( m \) has the form \( \indic_{\varepsilon}(s, h) \) such that \( (s, h) \vdash P \). Since \( (m, m') \in \text{Perm} \), every element \( \sigma' \in \text{supp}(m') \) must have a corresponding \( \sigma \in \text{supp}(m) \) such that \( \sigma' = \pi(\sigma) \) for some permutation \( \pi \). Now, since sigma has the form \( \indic_{\varepsilon}(s, h) \), then \( \sigma' \) must have the form \( \pi(\indic_{\varepsilon}(s, h)) = \indic_{\varepsilon}(\pi(s, h)) \), and since we know that \( (s, h) \vdash P \), then by Lemma C.6, \( \pi(s, h) \vdash P \) too. Since this applies to all elements of \( m' \) and since \( |m'| = |m| = 1 \), then \( m' \vdash (\varepsilon : P) \).

\( \square \)

### C.3 Replacement of Unsafe States

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

\[
\text{Rep} = \{(\text{undef}, \sigma) \mid \sigma \in \text{St}_\bot\} \cup \{(\sigma, \sigma) \mid \sigma \in \text{St}\}
\]

\[
\text{prune}(m)(\sigma) = \begin{cases} 
  m(\sigma) & \text{if } \sigma \neq \bot \\
  \emptyset & \text{if } \sigma = \bot 
\end{cases}
\]
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 4.6 (Replacement).** If $m \vdash \varphi$ and $(m, m') \in \overline{\text{Rep}}$, then prune$(m') \vdash \varphi$

**Proof.** By induction on the assertion $\varphi$.

- $\varphi = \top$. prune$(m') \vdash \varphi$ since everything satisfies $\top$.
- $\varphi = T^\oplus$. We know that $m = 0$. By Lemma B.2, that means that $m' = 0$ and so prune$(m') = 0$, so prune$(m') \vdash T^\oplus$.
- $\varphi = \varphi_1 \lor \varphi_2$. Without loss of generality, suppose $m \vdash \varphi_1$. By the induction hypothesis, prune$(m') \vdash \varphi_1$. We can weaken this to conclude that prune$(m') \vdash \varphi_1 \lor \varphi_2$.
- $\varphi = \varphi_1 \otimes_a \varphi_2$. We know that $m_1 \vdash \varphi_1$ and $m_2 \vdash \varphi_2$ for some $m_1$ and $m_2$ such that $a \cdot m_1 + \overline{a} \cdot m_2 = m$. Since $(a \cdot m_1 + \overline{a} \cdot m_2, m') \in \text{Rep}$, there must be some $m'_1$ and $m'_2$ such that $(m_1, m'_1) \in \text{Rep}$ and $(m_2, m'_2) \in \text{Rep}$ and $m' = a \cdot m'_1 + \overline{a} \cdot m'_2$. By the induction hypothesis, prune$(m'_1) \vdash \varphi_1$ and prune$(m'_2) \vdash \varphi_2$. It is easy to see that $a \cdot \text{prune}(m'_1) + \overline{a} \cdot \text{prune}(m'_2) = \text{prune}(a \cdot m'_1 + \overline{a} \cdot m'_2) = \text{prune}(m')$, therefore prune$(m') \vdash \varphi_1 \otimes_a \varphi_2$.
- $\varphi = (\epsilon : P)$. By definition, undef $\notin \text{supp}(m)$, so $m' = m$ (since \text{Rep} only relates defined states to themselves), which satisfies $\epsilon : P$ by assumption.

\[\square\]

C.4 The Frame Rule

For all of the proofs in this section, we let $R = \text{Rep} \circ \text{frame}(F) \circ \text{Perm}$.

**Lemma C.7 (Sequencing).** For any $f : S \times H \rightarrow \mathcal{W}_A\text{St}$, if $(m_1, m_2) \in \overline{R}$ and:

\[\forall (\mathbb{1}_{\text{ok}}(s_1, h_1), \mathbb{1}_{\text{ok}}(s_2, h_2)) \in R. \ \exists m. \ f(s_2, h_1) = \text{prune}(m) \ \text{and} \ (f(s_1, h_1), m) \in \overline{R}\]

Then, there exists $m'_2$ such that bind$(\text{prune}(m_2), f) = \text{prune}(m'_2)$ and $(\text{bind}(m_1, f), m'_2) \in \overline{R}$.

**Proof.** First, let $f' : \text{St} \rightarrow \mathcal{W}_A\text{St}$ be defined as follows:

\[f'(\sigma) = \begin{cases} f(s, h) & \text{if } \sigma = \mathbb{1}_{\text{ok}}(s, h) \\ \text{unit}_\mathcal{W}(\sigma) & \text{otherwise} \end{cases}\]

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

- $\sigma = \mathbb{1}_{\text{ok}}(s, h)$ and $\tau = \mathbb{1}_{\text{ok}}(s', h')$. By definition, $f'(\tau) = f(s', h')$ and $f'(\sigma) = f(s, h)$ such that $(\mathbb{1}_{\text{ok}}(s, h), \mathbb{1}_{\text{ok}}(s', h')) \in R$. By assumption, $(f(s, h), f(s', h')) \in \overline{R}$.
- $\sigma = \text{undef}$. In this case, $f'(\sigma) = \text{unit}_\mathcal{W}(\text{undef})$, which means that $f'(\sigma)$ is related to all configurations of size $\mathbb{1}$ according to $\overline{R}$. Now, since $\text{sup}(A) = \mathbb{1}$, it must be that $|f'(\tau)| = \mathbb{1}$ and so there is some $u$ such that $|f'(\tau)| + u = \mathbb{1}$. Now, let $m_{\sigma, \tau} = f'(\tau) + u \cdot \text{unit}_\mathcal{W}(\text{undef})$, so clearly $f'(\tau) = \text{prune}(m_{\sigma, \tau})$ and $(f'(\sigma), m_{\sigma, \tau}) \in \overline{R}$.
- In this final case, $\tau$ cannot have the form $\mathbb{1}_{\text{ok}}$, 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 $\sigma$ must also not be an ok state, since ok states are only related to other ok states. Therefore, $f'(\sigma) = \text{unit}_\mathcal{W}(\sigma)$ and $f'(\tau) = \text{unit}_\mathcal{W}(\tau)$. By Lemma B.6, we conclude that $(\text{unit}_\mathcal{W}(\sigma), \text{unit}_\mathcal{W}(\tau)) \in \overline{R}$. 
Given this, we know that for each $\sigma$ and $\tau$, there must be some $m_{\sigma,\tau}$ and $m_{\sigma,\tau}'$ such that $f'(\tau) = \text{prune}(m_{\sigma,\tau})$ and:

$$f'(\sigma) = \lambda\sigma'.\sum_{\tau \in \text{supp}(m_{\sigma,\tau})} m_{\sigma,\tau}'(\sigma', \tau') \quad m_{\sigma,\tau} = \lambda\tau'.\sum_{\sigma' \in \text{supp}(f'(\sigma))} m_{\sigma,\tau}'(\sigma', \tau')$$

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

$$m'(\sigma', \tau') = \sum_{\sigma' \in \text{supp}(m_1)} \sum_{\tau' \in \text{supp}(m_2)} m(\sigma, \tau) \cdot m_{\sigma,\tau}'(\sigma', \tau') \quad m_2'(\tau') = \sum_{\sigma' \in \text{supp}(\text{bind}(m_1, f))} m'(\sigma', \tau')$$

Now, we show that:

$$\lambda\sigma'.\sum_{\tau' \in \text{supp}(m_2')} m(\sigma', \tau') = \lambda\sigma'.\sum_{\sigma' \in \text{supp}(m_1)} \sum_{\tau' \in \text{supp}(m_2')} m(\sigma, \tau) \cdot m'(\sigma', \tau')$$

$$= \lambda\sigma'.\sum_{\sigma' \in \text{supp}(m_1)} \sum_{\tau' \in \text{supp}(m_2')} m(\sigma, \tau) \cdot (\sum_{\sigma' \in \text{supp}(m_{\sigma,\tau})} m_{\sigma,\tau}'(\sigma', \tau'))$$

$$= \lambda\sigma'.\sum_{\sigma \in \text{supp}(m_1)} m_1(\sigma) \cdot f'(\sigma)(\sigma') = \text{bind}(m_1, f)$$

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

$$\text{prune}(m_2') = \text{prune}(\lambda\tau'.\sum_{\sigma' \in \text{supp}(\text{bind}(m_1, f))} m(\sigma', \tau'))$$

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

$$= \text{prune}(\lambda\tau'.\sum_{\tau \in \text{supp}(m_2)} \sum_{\sigma' \in \text{supp}(m_1)} m(\sigma, \tau) \cdot m_{\sigma,\tau}'(\sigma', \tau'))$$

$$= \text{prune}(\lambda\tau'.\sum_{\tau \in \text{supp}(m_2)} m_2(\tau) \cdot m_{\sigma,\tau}(\tau'))$$

$$= \lambda\tau'.\sum_{\tau \in \text{supp}(m_2)} m_2(\tau) \cdot \text{prune}(m_{\sigma,\tau})(\tau')$$

$$= \lambda\tau'.\sum_{\tau \in \text{supp}(m_2)} m_2(\tau) \cdot f'((\tau')) = \text{bind}(m_2, f)$$

□

**Lemma C.8.** If $(\mathbb{I}_e(s, h), \mathbb{I}_e(s', h')) \in R$, then there is some $h''$ and permutation $\pi$ such that $s' = \pi(s)$ and $h' = \pi(h) \uplus h''$ and $(\pi(s), h'') \not\in F$.

**Proof.** First note that $R = \text{Rep} \circ \text{frame}(F) \circ \text{Perm}$, so there must be $\sigma$ and $\sigma'$ such that $(\mathbb{I}_e(s, h), \sigma) \in \text{Perm}$ and $(\sigma, \sigma') \in \text{frame}(F)$, and $(\sigma', \mathbb{I}_e(s', h')) \in \text{Rep}$. By the definition of $\text{Perm}$, $\sigma = \pi(\mathbb{I}_e(s, h)) = \mathbb{I}_e(\pi(s), \pi(h))$ for some permutation $\pi$. By the definition of $\text{frame}(F)$, $\sigma' = \mathbb{I}_e(\pi(s), \pi(h) \uplus h'')$
such that \( \pi(s) \neq s' \) is not an undef state, then it can only be related to itself according to \( \mathcal{R} \), and so \( s = s' \), therefore \( s' = \pi(s) \)

\( \Box \)

Lemma C.9. If \( (\mathcal{I}_{ok}(s_1, h_1), \mathcal{I}_{ok}(s_2, h_2)) \in R \) whenever \( h(\mathcal{I}(s)) \in \text{Val} \), then

\[
\text{update}(s, h, [e](s_1, h_1), \text{update}(\pi(s), \pi(h) \cup h, [e](\pi(s), s_2, h_2)) \in R
\]

Proof. Let \( \ell = [e](s) \). By Lemma C.3, \( \pi(s) = \pi([e](s)) = \pi(\ell) \). We complete the proof by case analysis:

\( \ell \in \text{Val} \). Since \( \pi(h)(\pi(\ell)) = (\pi \circ h \circ \pi^{-1})(\pi(\ell)) = \pi(h(\ell)) \), then \( \pi(h)(\pi(\ell)) \) must also be a value and then so must \( (\pi(h) \cup h') \). So, it just remains to show that \( (\text{unit}(s_1, h_1), \text{unit}(s_2, h_2)) \in R \), which follows from Lemma B.6.

\( \ell \in \text{dom}(h) \). By a similar argument to the previous case, it must be that \( (\pi(h) \cup h')(\pi(\ell)) = \ell \) too. So, we just need to show that \( (\text{error}(s, h), \text{error}(\pi(s), \pi(h) \cup h')) \in R \). We know that \( \mathcal{I}_{ok}(s, h), \mathcal{I}_{ok}(\pi(s), \pi(h) \cup h')) \in R \) by the definition of \( \mathcal{R} \) and so the claim follows by Lemma B.6.

\( \ell \not\in \text{dom}(h) \). So, \( \text{update}(s, h, [e](s_1, h_1), \text{unit}(\text{undef}(\text{undef})) = \text{unit}(\text{undef}(\text{undef})) \), and since \( \mathcal{R} \) relates \text{undef} to all states, \( \text{update}(s, h, [e](s_1, h_1), m) \in \mathcal{R} \) for all \( m \), which means that \( \text{update}(\pi(s), \pi(h) \cup h', [e](\pi(s), s_2, h_2)) \) is related trivially.

\( \Box \)

Lemma C.10. If \( (\sigma, \tau) \in R \) for every \( \sigma \in \text{supp}(m_1) \) and \( \tau \in \text{supp}(m_2) \) and \( |m_1| = |m_2| \), then \( (m_1, m_2) \in \mathcal{R} \).

Proof. We know that \( \sum_{x \in \text{supp}(m_1)} m_1(x) = \sum_{y \in \text{supp}(m_2)} m_2(y) \), so by Lemma A.5, we know there must be \( (u_k)_{k \in \text{supp}(m_1) \times \text{supp}(m_2)} \) such that \( m_1(x) = \sum_{y \in \text{supp}(m_2)} u(x, y) \) for all \( x \in \text{supp}(m_1) \) and \( m_2(y) = \sum_{x \in \text{supp}(m_1)} u(x, y) \) for all \( y \in \text{supp}(m_2) \). Letting \( m(x, y) = u(x, y) \) we have a witness that \( (m_1, m_2) \in \mathcal{R} \).

\( \Box \)

Lemma 4.7 (The Frame Property). Let \( R = \mathcal{R} \circ \text{frame}(F) \circ \text{Perm} \), so \( R \subseteq \text{St} \times \text{St}_1 \). For any program \( C \) and symbolic heap \( F \) such that \( \text{mod}(C) \cap \text{fv}(F) = \emptyset \):

\[
\forall (\mathcal{I}_{ok}(s, h), \mathcal{I}_{ok}(s', h')) \in R. \exists m. [C](s', h') = \text{prune}(m) \quad \text{and} \quad ([C](s, h), m) \in R
\]

Proof. By induction on the structure of the program \( C \).

\( C = \text{skip} \). In this case, \([C](s, h) = \text{unit}(s, h) \) and \([C](s', h') = \text{unit}(s', h') \), so the claim follows from Lemma B.6.

\( C = C_1 + C_2 \). By definition, we know that \([C](s, h) = \text{bind}(C_1, [C_1](s, h), [C_2](s, h)) \) and \([C](s', h') = \text{bind}(C_1(s', h'), [C_2](s')) \). By the induction hypothesis, we know there is some \( m \) such that \([C_1](s', h') = \text{prune}(m) \) and \((C_1, s, h, m) \in R \). Therefore by the induction hypothesis and Lemma C.7, we can conclude that there is some \( m' \) such that \([C_1](s', h'), [C_2](s') \) is equal to \( \text{prune}(m') \) and \((C_1, s, h, [C_1](s', h'), [C_2](s')) \) is equal to \( \text{prune}(m') \) and \((C_1, s, h, [C_1](s', h'), [C_2](s')) \in R \), which completes the proof.

\( C = C_1 + C_2 \). We know that \([C_1 + C_2](s, h) = a \cdot [C_1](s, h) + \bar{a} \cdot [C_2](s, h) \) and \([C_1 + C_2](s', h') = a \cdot [C_1](s', h') + \bar{a} \cdot [C_2](s', h') \). By the induction hypothesis, for each \( i \in \{1, 2\} \) we get that there is some \( m_i \) such that \([C_i](s', h') = \text{prune}(m_i) \) and \(([C_i](s, h), m_i) \in R \). Using Lemmas B.4 and B.5 we can conclude that \( ([C_1 + C_2](s, h), a \cdot m_1 + \bar{a} \cdot m_2) \in R \). It now only remains to show that:

\[
\text{prune}(a \cdot m_1 + \bar{a} \cdot m_2) = a \cdot \text{prune}(m_1) + \bar{a} \cdot \text{prune}(m_2) = a \cdot [C_1](s', h') + \bar{a} \cdot [C_2](s', h') = [C_1 + C_2](s', h')
\]
We now move on to the cases involving expressions and primitive instructions. By Lemma C.8, $s' = \pi(s)$ and $h' = \pi(h) \cup h''$ such that $(\pi(s), \pi(h) \cup h'') \Vdash F$, so we will use that fact in the following cases. Additionally, many of the cases have a single outcome, so by Lemma B.6 it suffices to show that $(s, \tau) \in R$ where $[C] (s, h) = \text{unit}_W(\sigma)$ and $[C] (s', h') = \text{unit}_W(\tau)$ in those cases.

- $C = \text{if } e \text{ then } C_1 \text{ else } C_2$. Since $s' = \pi(s)$, then $[e] (s') = [e] (\pi(s)) = \pi([e] (s)) = [e] (s)$ where the last step is valid since we assume $e$ is boolean valued and therefore the result of evaluating it will not be changed by permuting addresses. This means that both programs will take the same path. Without loss of generality, suppose $[e] (s) = \text{true}$. So, that means that $[C] (s, h) = [C_1] (s, h)$ and $[C] (s', h') = [C_1] (s', h')$. We complete the proof by applying the induction hypothesis to conclude that there exists $m$ with $[C_2] (s', h') = \text{prune}(m)$ and $([C_1] (s, h), m) \in \mathcal{R}$. The case where $[e] (s) = \text{false}$ is symmetric.

- $C = \text{while } e \text{ do } C'$. First, we will show that there exists $m$ such that $F^n_{(C', e)} (\bot) (s', h') = \text{prune}(m)$ and $F^n_{(C', e)} (\bot) (s, h), m) \in \mathcal{R}$ for any $(\mathbb{I}_{\text{ok}}(s, h), \mathbb{I}_{\text{ok}}(s', h')) \in \mathcal{R}$. The proof is by induction on $n$. If $n = 0$, then the claim holds using Lemma B.2:

$$F^0_{(C', e)} (\bot) (s', h') = \bot = \bot (s', h') = 0 = \bot (s, h) = F^0_{(C', e)} (\bot) (s, h)$$

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

$$F^{n+1}_{(C', e)} (\bot) (s', h') = \begin{cases} \text{bind} ([C'] (s', h'), F^n_{(C', e)} (\bot)) & \text{if } [e] (s') = \text{true} \\ \text{unit}(s', h') & \text{if } [e] (s') = \text{false} \end{cases}$$

and

$$F^{n+1}_{(C', e)} (\bot) (s, h) = \begin{cases} \text{bind} ([C'] (s, h), F^n_{(C', e)} (\bot)) & \text{if } [e] (s) = \text{true} \\ \text{unit}(s, h) & \text{if } [e] (s) = \text{false} \end{cases}$$

Note that as we showed in the previous case for if statements, $[e] (s) = [e] (s')$, so both executions will take the same path. If $[e] (s') = [e] (s) = \text{true}$, then the claim holds by Lemma C.7 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:

$$(F^n_{(C', e)} (\bot) (s, h), F^n_{(C', e)} (\bot) (s', h') + a_n \cdot \text{unit}(\bot)) \in \mathcal{R}$$

Recall that $\bot$ 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 $F^n_{(C', e)} (\bot) (s', h') + a_n \cdot \text{unit}(\bot)$ is a chain, so by Lemma B.8 we know that:

$$(\sup_{n \in \mathbb{N}} F^n_{(C', e)} (\bot) (s, h), \sup_{n \in \mathbb{N}} F^n_{(C', e)} (\bot) (s', h') + a_n \cdot \text{unit}(\bot)) \in \mathcal{R}$$

And we can therefore conclude that there exists $m$ such that $[\text{while } e \text{ do } C'] (s', h') = m$ and $(\text{while } e \text{ do } C') (s, h), m) \in \mathcal{R}$.

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

$$[C] (s, h) = \text{unit}(s [x \mapsto [e] (s)])$$

$$[C] (\pi(s), \pi(h) \cup h'') = \text{unit}(\pi(s) [x \mapsto [e] (\pi(s))) \cup h'')$$

By Lemma C.3, $[e] (\pi(s)) = \pi([e] (s))$, therefore $\pi(s) [x \mapsto [e] (\pi(s))] = \pi(s [x \mapsto [e] (s))$. So, it is clearly the case that $(\mathbb{I}_{\text{ok}}(s [x \mapsto [e] (s), h), \mathbb{I}_{\text{ok}}(\pi(s [x \mapsto [e] (s)), \pi(h)) \in \text{Perm}$.

Now, since $x \in \text{mod}(C)$, then $x \notin \text{fv}(F)$, so updating $x$ in $\pi(s)$ will not affect the truth of $F$, therefore $(\pi(s [x \mapsto [e] (s)), h'') \Vdash F$. So, we know that:

$$(\mathbb{I}_{\text{ok}}(s [x \mapsto [e] (s), h), \mathbb{I}_{\text{ok}}(\pi(s [x \mapsto [e] (s)), \pi(h) \cup h'')) \in \text{frame}(F)$$
Putting these together along with the fact that Rep is reflexive gives us that $1_{\text{ok}}(s[x \mapsto [e](s)], h), 1_{\text{ok}}(\pi(s[x \mapsto [e](s)]), \pi(h \cup h')) \in R$.

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

$$[C](s, h) = \text{bind}_W(\text{alloc}(\text{dom}(h)), \lambda\ell.\text{unit}(s[x \mapsto \ell], h[\ell \mapsto 0]))$$

$$[C](\pi(s), \pi(h \cup h')) = \text{bind}_W(\text{alloc}(\pi(h) \cup h'), \lambda\ell.\text{unit}(\pi(s)[x \mapsto \ell], (\pi(h) \cup h')[\ell \mapsto 0]))$$

Since $|C|(s, h) = |C|(\pi(s), \pi(h \cup h')) = 1$, we can use Lemma C.10 to complete the proof as long as all the elements in both supports are related. Take $\sigma \in \text{supp}([C](s, h))$ and $\tau \in \text{supp}([C](\pi(s), \pi(h \cup h')))$, so $\sigma = 1_{\text{ok}}(s[x \mapsto \ell_1], h[\ell_1 \mapsto 0])$ and $\tau = 1_{\text{ok}}(\pi(s)[x \mapsto \ell_2], (\pi(h) \cup h')[\ell_2 \mapsto 0])$ for some $\ell_1$ and $\ell_2$ such that $\ell_1 \notin \text{dom}(h) \cup \text{im}(s) \cup \text{im}(h)$ and $\ell_2 \notin \text{dom}(\pi(h) \cup h') \cup \text{im}(\pi(s)) \cup \text{im}(\pi(h) \cup h')$. Now, let $\pi'$ be defined as follows:

$$\pi'(\ell) = \begin{cases} 
\ell_2 & \text{if } \ell = \ell_1 \\
\pi(\ell_1) & \text{if } \ell = \pi^{-1}(\ell_2) \\
\pi(\ell) & \text{otherwise}
\end{cases}$$

Clearly $\pi'$ is also a valid permutation since it just swaps the values of $\ell_1$ and $\pi^{-1}(\ell_2)$ in $\pi$. Note that since $\ell_2 \notin \text{dom}(\pi(h) \cup h) \cup \text{im}(\pi(s)) \cup \text{im}(\pi(h) \cup h')$, then $\pi^{-1}(\ell_2) \notin \text{dom}(h) \cup \text{im}(s) \cup \text{im}(h)$, and so when $\pi'$ is applied to $s$ or $h$, it has the same effect as $\pi$.

Now, we have:

$$1_{\text{ok}}(\pi(s)[x \mapsto \ell_2], (\pi(h) \cup h')[\ell_2 \mapsto 0]) = 1_{\text{ok}}(\pi'(s)[x \mapsto \ell_2], (\pi'(h) \cup h')[\ell_2 \mapsto 0])$$

Since $\ell_2 \notin \text{dom}(h')$:

$$1_{\text{ok}}(\pi(s)[x \mapsto \ell_2], (\pi(h) \cup h')[\ell_2 \mapsto 0]) = 1_{\text{ok}}(\pi'(s)[x \mapsto \pi'(\ell_1)], \pi'(h)[\pi'(\ell_1) \mapsto 0] \cup h')$$

$$= 1_{\text{ok}}(\pi'(s)[x \mapsto \ell_1]), \pi'(h[\ell_1 \mapsto 0]) \cup h')$$

Now, clearly $(1_{\text{ok}}(s[x \mapsto \ell_1], h[\ell_1 \mapsto 0]), 1_{\text{ok}}(\pi'(s)[x \mapsto \ell_1], \pi'(h[\ell_1 \mapsto 0]))) \in \text{Perm}$. Also, since $x \in \text{mod}(C)$, then updating $x$ in $\pi(s)$ will not affect the truth of $F$, so $(\pi(s)[x \mapsto \ell_2], h' \not= F$ and therefore $(1_{\text{ok}}(\pi'(s)[x \mapsto \ell_1]), \pi'(h[\ell_1 \mapsto 0])), 1_{\text{ok}}(\pi'(s)[x \mapsto \ell_1], \pi'(h[\ell_1 \mapsto 0]) \cup h')) \in \text{frame}(F)$, so we are done.

$\blacktriangleright$ $C = \text{free}(e)$. Using Lemma C.9, we only need to show that if $h([e](s)) \in \text{Val}$, then:

$$(1_{\text{ok}}(s, h[[e](s)] \mapsto \bot), 1_{\text{ok}}(\pi(s), (\pi(h \cup h')[[e](s)]) \mapsto \bot)) \in R$$

If $h([e](s)) \in \text{Val}$, then $[e](\pi(s)) \in \text{dom}(\pi(h))$, and since $h'$ is disjoint from $\pi(h)$, then $[e](\pi(s)) \not\in \text{dom}(h')$, so:

$$(\pi(h \cup h')[[e](s) \mapsto \bot] = \pi(h)\langle [e](s) \mapsto \bot \rangle \cup h'$$

By Lemma C.3

$$= \pi(h)\langle [e_1](s) \mapsto \bot \rangle \cup h'$$

$$= \pi(h\langle [e_1](s) \mapsto \bot \rangle \cup h')$$

So, clearly $(1_{\text{ok}}(s, h[[e](s)] \mapsto \bot), 1_{\text{ok}}(\pi(s), (\pi(h[[e](s)]) \mapsto \bot))) \in \text{Perm}$ and also $(1_{\text{ok}}(\pi(s), \pi(h[[e](s)]) \mapsto \bot)), 1_{\text{ok}}(\pi(s), (\pi(h[[e](s)]) \mapsto \bot) \cup h')) \in \text{frame}(F)$, so we are done.

---

7The fact that $\ell_2 \notin \text{im}(\pi(s)) \cup \text{im}(\pi(h) \cup h')$ must hold was overlooked in the proof of Baktiev [2006, Thm. 3.1], but without this requirement we cannot guarantee that $\pi'(s) = \pi(s)$ and $\pi'(h) = \pi(h)$. 

Noam Zilberstein, Angelina Saliling, and Alexandra Silva
\( C = ([e_1] \leftarrow e_2) \). Using Lemma C.9, we only need to show that if \( h([e_1] (s)) \in \text{Val} \), then:

\[
(1_{ok}(s, h([e_1] (s)) \mapsto [e_2] (s))), 1_{ok}(\pi(s), (\pi(h) \uplus h')([e_1] (\pi(s)) \mapsto [e_2] (\pi(s)))) \in R
\]

If \( h([e_1] (s)) \in \text{Val} \), then \([e_1] (\pi(s)) \in \text{dom}(\pi(h))\), and since \( h' \) is disjoint from \( \pi(h) \), then \([e_1] (\pi(s)) \not\in \text{dom}(h')\), so:

\[
(\pi(h) \uplus h')([e_1] (\pi(s)) \mapsto [e_2] (\pi(s))) = \pi(h)([e_1] (\pi(s)) \mapsto [e_2] (\pi(s)) \uplus h')
\]

By Lemma C.3

\[
= \pi(h)[\pi([e_1] (s)) \mapsto \pi([e_2] (s))] \uplus h'
\]

So, clearly \((1_{ok}(s, h([e_1] (s)) \mapsto [e_2] (s))), 1_{ok}(\pi(s), (\pi(h)[h])([e_1] (\pi(s)) \mapsto [e_2] (\pi(s)))) \in \text{Perm} \) and \((1_{ok}(\pi(s), \pi(h)[h])([e_1] (\pi(s)) \mapsto [e_2] (\pi(s)))), 1_{ok}(\pi(s), \pi(h)[h])([e_1] (\pi(s)) \mapsto [e_2] (\pi(s)) \uplus h')) \in \text{frame}(F)\), so we are done.

\( C = (x \leftarrow [e]) \). Using Lemma C.9, we only need to show that if \( h([e] (s)) \in \text{Val} \), then:

\[
(1_{ok}(s[x \mapsto h([e] (s))), 1_{ok}(\pi(s)[x \mapsto (\pi(h) \uplus h')(\pi(s))], (\pi(h) \uplus h')) \in R
\]

If \( h([e] (s)) \in \text{Val} \), then \([e] (\pi(s)) \in \text{dom}(\pi(h))\), and since \( h' \) is disjoint from \( \pi(h) \), then \([e] (\pi(s)) \not\in \text{dom}(h')\), so:

\[
\pi(s)[x \mapsto (\pi(h) \uplus h')(\pi(s))]) = \pi(s)[x \mapsto \pi(h)(\pi([e] (\pi(s))))]
\]

\[
= \pi(s)[x \mapsto \pi(h)(\pi([e] (\pi(s))))]
\]

\[
= \pi(s)[x \mapsto \pi(h)(\pi([e] (\pi(s))))]
\]

So, clearly \((1_{ok}(s[x \mapsto h([e] (s))), 1_{ok}(\pi(s)[x \mapsto h([e] (s))]), \pi(h)), 1_{ok}(\pi(s[x \mapsto h([e] (s))]), \pi(h) \uplus h')) \in \text{frame}(F)\), so we are done.

\( C = \text{error}() \). By Lemma B.6, it suffices to show that \((1_{er}(s, h), 1_{er}(\pi(s), \pi(h) \uplus h')) \in R\), which follows from the fact that \((1_{ok}(s, h), 1_{ok}(\pi(s), \pi(h) \uplus h')) \in R\) since \( R \) treats \( \text{ok} \) and \( \text{er} \) states in the same way.

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

\[\square\]

**Theorem 4.8 (The Frame Rule).** If \( \langle \phi \rangle C \langle \psi \rangle \) and \( \text{mod}(C) \cap \text{fv}(F) = \emptyset \), then \( \langle \phi \otimes F \rangle C \langle \psi \otimes F \rangle \).

**Proof.** Suppose \( m \vdash \phi \otimes F \). Then by Lemma 4.4 we know that there are \( m_1, m'_1 \), and \( m_2 \) such that \( (m_1, m'_1) \in \text{frame}(F) \) and \( m = m'_2 + m_2 \) and \( m_1 + m'_2 \not\equiv \varphi \) for any \( m'_2 \) such that \( |m'_2| \leq |m_2| \). So that means that \( m_1 + |m_2| \cdot \text{unit(undef)} \not\equiv \varphi \). Now, we know that:

\[ [C]^{\uparrow} (m_1 + |m_2| \cdot \text{unit(undef)}) = [C]^{\uparrow} (m_1) + |m_2| \cdot \text{unit(undef)} \]

So, therefore \([C]^{\uparrow} (m_1) + |m_2| \cdot \text{unit(undef)} \not\equiv \psi \) since \( \langle \phi \rangle C \langle \psi \rangle \). Now, observe that \((m_1, m'_1) \in \text{frame}(F) \subseteq \overline{R} \) since both additional components of \( R \) are reflexive. We can also conclude that \(|m_2| \cdot \text{unit(undef)} \), \( m_2 \rangle \in \overline{R} \) since \( R \) permits \( \text{undef} \) states to be remapped to anything. Therefore, using Lemma B.4 we get that:

\[(m_1 + |m_2| \cdot \text{unit(undef)} , m) = (m_1 + |m_2| \cdot \text{unit(undef)} , m'_2 + m_2) \in \overline{R} \]
Algorithm 1 abduce-par(\(P, Q\))

1: if either Base-Emp, Base-True-L, or Base-True-R apply then
2: return anti-frame M as indicated by that rule
3: else
4: for Each remaining row in Figure 6 from top to bottom do
5: result = \(\emptyset\)
6: for Each inference rule in the row of the form below do
7: \(P' \vdash [M'] \triangleright Q' \quad R\)
8: if The input parameters match \(P\) and \(Q\) in the inference rule and \(R\) is true then
9: result = result \(\cup \{M | M' \in abduce-par(P', Q')\}\)
10: end if
11: end for
12: if result \(\neq \emptyset\) then
13: return result
14: end if
15: end for
16: return \(\emptyset\)

So, using Lemmas 4.7 and C.7, we know that there exists some \(m'\) such that \([C]^{\dagger}(m) = \text{prune}(m')\) and:

\((\text{prune}(m_1 + |m_2| \cdot \text{unit(undef)}), m') \in \overline{R} \subseteq \text{Rep} \circ \text{frame}(F) \circ \text{Perm}\)

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 \(m''\) and \(m'''\) such that \((\text{prune}(m_1 + |m_2| \cdot \text{unit(undef)}), m'') \in \text{Perm}\) and \((m'', m''') \in \text{frame}(F)\) and \((m''', m') \in \text{Rep}\). By Lemma 4.5, \(m'' \equiv \psi\), and by Lemma 4.3 \(m''' \equiv \psi \circ F\). Finally, by Lemma 4.6, \([C]^{\dagger}(m) \equiv \psi \circ F\).

\(\square\)

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 \(P \vdash [M] \triangleright Q\) is derivable, then \(M \vDash P\) and \(M \vDash Q\)

**Proof.** The proof is by induction on the derivation of \(P \vdash [M] \triangleright Q\).

1. **Base-Emp.** We need to show that \(\Pi \land \Pi' \land \text{emp} \vDash \Pi \land \text{emp}\) and \(\Pi \land \Pi' \land \text{emp} \vDash \Pi' \land \text{emp}\), both hold by the semantics of logical conjunction.
2. **Base-True-L.** We need to show that \(\Pi \land \Pi' \land \Sigma' \vDash \Pi \land \text{true}\) 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 \(\overline{X}\) is not free in \(\Delta'\) with \(\overline{Y}\) removed and vice versa. Now let:

\[\overline{Z} = \overline{X} \cap \overline{Y} \quad \overline{X}' = \overline{X} \setminus \overline{Z} \quad \overline{Y}' = \overline{Y} \setminus \overline{Z}\]
Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects

Base Cases

\[
\begin{align*}
\Pi \land \Pi' & \neq \text{false} \quad \text{BASE-EMP} \\
\Pi \land \text{emp} & < [\Pi \land \Pi' \land \text{emp}] \rightarrow \Pi' \land \text{emp} \\
\Pi \land \Pi' \land \Sigma' & \neq \text{false} \quad \text{BASE-TRUE-L} \\
\Pi \land \text{true} & < [\Pi \land \Pi' \land \Sigma'] \rightarrow \Pi' \land \Sigma' \\
\Pi \land \Pi' \land \Sigma & \neq \text{false} \quad \text{BASE-TRUE-R} \\
\end{align*}
\]

Quantifier Elimination

\[
\begin{align*}
\Delta & \lessdot \{M\} \rightarrow \Delta' \quad \exists \bar{X} \Delta \lessdot \{\exists \bar{X} \bar{Y} . M\} \rightarrow \exists \bar{Y} \Delta' \quad \text{EXISTS} \\
\end{align*}
\]

Resource Matching

\[
\begin{align*}
\Delta + \text{ls}(e_1, e_2) & < [M + e_1 \mapsto e_3] \rightarrow \Delta' + e_1 \mapsto e_3 \quad \text{LS-START-L} \\
\Delta + \text{ls}(e_1) & < [M + e_1 \mapsto e_3] \rightarrow \Delta' + \text{ls}(e_1, e_2) \quad \text{LS-START-R} \\
\Delta + e_1 & < [M + e_1 \mapsto e_3] \rightarrow \Delta' + \text{ls}(e_1, e_3) \quad \text{LS-END-L} \\
\Delta + \text{ls}(e_1, e_2) & < [M + \text{ls}(e_1, e_3)] \rightarrow \Delta' + \text{ls}(e_1, e_3) \quad \text{LS-END-R} \\
\end{align*}
\]

Resource Adding

\[
\begin{align*}
\Delta & \lessdot \{M\} \rightarrow \Pi' \land (\Sigma' + \text{true}) \quad \Pi' \land \Sigma' + B(e_1, e_2) & \neq \text{false} \quad \text{MISSING-L} \\
\Pi \land (\Sigma + \text{true}) & < [M + B(e_1, e_2)] \rightarrow \Pi' \land (\Sigma' + \text{true}) \quad \text{MISSING-R} \\
\Delta + e_1 = e_2 & < [M] \rightarrow \Delta' + e_1 = e_2 \quad \text{EMP-LS-L} \\
\Delta + \text{ls}(e_1, e_2) & < [M] \rightarrow \Delta' + \text{ls}(e_1, e_2) \quad \text{EMP-LS-R} \\
\end{align*}
\]

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 \(\text{ls}(e_1, e_2)\) or \(e_1 \mapsto e_2\).

That is, \(\bar{Z}\) is the variables occurring in both \(\bar{X}\) and \(\bar{Y}\), \(\bar{X}'\) is the variables only occurring in \(\bar{X}\) and \(\bar{Y}'\) is the variables only occurring in \(\bar{Y}\). This means that \(\bar{X}', \bar{Y}', \) and \(\bar{Z}\) are disjoint.

We first show that \(\exists \bar{X} \bar{Y} . M \supseteq \exists \bar{X} \Delta\). Suppose \((s, h) \equiv \exists \bar{X} \bar{Y} . M\). We know that \((s', h) \equiv M\) where \(s' = s[\bar{X}' \mapsto \bar{v}_1][\bar{Y}' \mapsto \bar{v}_2][\bar{Z} \mapsto \bar{v}_3]\) for some \(\bar{v}_1, \bar{v}_2, \) and \(\bar{v}_3\). Given that we know \(M \supseteq \Delta\), we now have \((s', h) \equiv \exists \bar{X} \Delta\) (since \(\bar{X} = \bar{X}' \cup \bar{Z}\)). Now, given that \(\bar{Y} \cap (\text{fv}(\Delta) \setminus \bar{X}) = \emptyset\), we know that none of the variables in \(\bar{Y}'\) are free in \(\Delta\), and so we can remove them from the state to conclude that \((s, h) \equiv \exists \bar{X} \Delta\).

It can also be shown that \(\exists \bar{X} \bar{Y} . M \equiv \exists \bar{Y} \Delta'\) by a symmetric argument.

((5) LS-START-L. Here, we know \(M \equiv \Delta * \text{ls}(e_3, e_2)\) and \(M \equiv \Delta'\). We now want to show \(M \equiv e_1 \mapsto e_3 \equiv \Delta * \text{ls}(e_1, e_2)\) and \(M * e_1 \equiv e_3 \equiv \Delta' * e_1 \equiv e_3\).

Suppose that \((s, h) \equiv M \equiv e_1 \mapsto e_3\), and so \((s, h_1) \equiv M\) and \((s, h_2) \equiv e_1 \equiv e_3\) for some \(h_1\) and \(h_2\) such that \(h_1 \cup h_2\).

Since \(M \equiv \Delta * \text{ls}(e_3, e_2)\), we get that \((s, h_1) \equiv \Delta * \text{ls}(e_3, e_2)\) and recombining, we get that \((s, h) \equiv \Delta * \text{ls}(e_3, e_2) \mapsto e_1 \equiv e_3\). Now, \(\text{ls}(e_3, e_2) \equiv e_1 \equiv e_3 \equiv \exists \bar{X} . e_1 \equiv X * \text{ls}(X, e_2)\) and
Theorem 5.1 (Tri-abduction).

Let us first show $M \models \Delta \land e_1 \rightarrow e_2$, which means that because $(s, h_1) \models M$, $(s, h_1) \models \Delta'$. This means that $(s, h_1 \uplus h_2) \models \Delta' \land e_1 \rightarrow e_3$ and $h = h_1 \uplus h_2$, so we now have $(s, h) \models \Delta' \land e_1 \rightarrow e_3$, therefore $M \models e_1 \rightarrow e_3 \models \Delta' \land e_1 \rightarrow e_3$.

LS-START-R. This case is symmetric to Case 5.

(7) MATCH. Here, we know that $M \models \Delta' \land e_2 = e_3$ and $M \models \Delta' \land e_2 = e_3$. We want to show that $M \models e_1 \rightarrow e_2 \models e_1 \rightarrow e_2$ and $M \models e_1 \rightarrow e_2 \models e_1 \rightarrow e_3$.

Let us first show $M \models e_1 \rightarrow e_2 \models e_1 \rightarrow e_2$. Because we know that $M \models \Delta$ and $e_1 \models e_2 \rightarrow e_1$, it follows that $M \models e_1 \rightarrow e_2 \models e_2 \rightarrow e_1$.

Let us now show $M \models e_1 \rightarrow e_2 \models e_1 \rightarrow e_3$. We know $M \models \Delta' \land e_2 = e_3$, now suppose that $(s, h) \models M \models e_1 \rightarrow e_2$, so $(s, h) \models \Delta' \land e_2 = e_3$ as well. This means that $e_2 = e_3$ in state $s$, and therefore it must also be the case that $(s, h) \models e_1 \rightarrow e_3$. Given what else we know, we conclude that $(s, h) \models \Delta' \land e_1 \rightarrow e_3$.

LS-END-R. This case is symmetric to Case 8.

(9) LS-END-R. This case is symmetric to Case 9. (10) MISSING-L. Here, we know that $M \models \Delta$ and $M \models \Pi' \land (\Sigma' \land e_3)$, which means this also holds that $M \models \Pi'$ and $M \models \Sigma' \land e_3$ by semantic definition.

Let us first show that $M \models B(e_1, e_2) = \Delta \land B(e_1, e_2)$. Suppose that $(s, h) \models M \models B(e_1, e_2)$. We know that $(s, h_1) \models M$ and $(s, h_2) \models B(e_1, e_2)$ for some $h_1$ and $h_2$ such that $h = h_1 \uplus h_2$. Since $M \models \Delta$, then $(s, h_1) \models \Delta$. This means that $(s, h) \models \Delta \land B(e_1, e_2)$, therefore $M \models B(e_1, e_2) = \Delta \land B(e_1, e_2)$.

Let us now show that $M \models B(e_1, e_2) = \Pi' \land (\Sigma' \land e_3)$. Here, we know that $M \models \Pi' \land (\Sigma' \land e_3)$ and trivially $B(e_1, e_2) = true$. This means $M \models B(e_1, e_2) = \Pi' \land (\Sigma' \land e_3)$, therefore, $M \models B(e_1, e_2) = \Pi' \land (\Sigma' \land e_3)$.

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

(12) EMP-LS-L. We know that $M \models \Delta \land e_1 = e_2$ and $M \models \Delta' \land e_1 = e_2$. This means that $M \models \Delta'$, so the right side of the tri-abduction judgement is taken care of.

Let us now show $M \models \Delta \land e_1 = e_2$. We first establish that $e_1 = e_2 \land \text{emp} \models \Delta \land e_1 = e_2$ by definition, since $\Delta \land e_1 = e_2 \models X \models X \land e_1 = e_2$. Now, we know that $M \models \Delta \land e_1 = e_2$, which also means that $M \models (\Delta \land e_1 = e_2)$, or equivalently, $M \models (\Delta \land e_1 = e_2)$. Using $e_1 = e_2 \land \text{emp} \models \Delta \land e_1 = e_2$, we get $M \models (\Delta \land e_1 = e_2) \models \Delta \land e_1 = e_2$, and by weakening we get $M \models \Delta \land e_1 = e_2$.

(13) EMP-LS-R. This case is symmetric to Case 12.

\[\exists X. e_1 \mapsto X \models X \models e_1 \models e_2, \text{ so we get that } (s, h) \models \Delta \models e_1 \models e_2 \text{ and therefore } M \models e_1 \mapsto e_3 \models \Delta \models e_1 \models e_2.\]

\begin{proof}
In our tri-abduction algorithm, we call abduce-par on $P \land e_1$ and $Q \land e_1$, so we know based on Lemma 8.1 that if $P \land e_1 \land [M] \models Q \models e_1$ is derivable, then $M \models P \models e_1$ and $M \models Q \models e_1$ 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 $M \models P \models e_1$ and $M \models Q \models e_1$ by Berdine et al. [2005b, Theorem 7].
\end{proof}
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 rename(\( \Delta, M, Q, Q, \vec{X}, \vec{x} \))

Let \( \vec{Y} \) be the free logical variables of \( Q \) and all the assertions in \( Q \).
Pick \( \vec{e} \) disjoint from \( \vec{Y} \) and \( \vec{x} \) such that \( \Delta * M \not\subseteq \vec{e} = \vec{Y} \).
Pick \( M' \) disjoint from \( \vec{X} \), \( \vec{Y} \), and \( \Vec{Var} \) such that \( \Delta * M' \not\subseteq \Delta * M[\vec{e}/\vec{Y}] \).
return \((\vec{e}, \vec{Y}, M')\)

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

\[
\text{biab}'(\exists \vec{Z}, \Delta, Q, \psi, \vec{x}) = \begin{cases} 
(M', (\psi \circ \exists \vec{X}.F[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]) & | (M, F) \in \text{biab}(\Delta, Q) \bigg) \\
(\vec{e}, \vec{Y}, M') = \text{rename}(\Delta, M, Q, \{\psi\}, \vec{Z}) \bigg) 
\end{cases}
\]

\[
\text{triab}'(P_1, P_2, \psi_1, \psi_2, \vec{x}) = \begin{cases} 
(M', (\psi_1 \circ \exists \vec{X}.F_1[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}], \\
(\psi_2 \circ \exists \vec{X}.F_2[\vec{X}/\vec{x}])[\vec{e}/\vec{Y}]) & | (M, F_1, F_2) \in \text{triab}(P_1, P_2) \\
(\vec{e}, \vec{Y}, M') = \text{rename}(\text{emp}, M, \{\psi_1, \psi_2\}, \emptyset) \bigg) 
\end{cases}
\]

The \( \text{biab}' \) procedure is similar to \( \text{AbduceAndAdapt} \) 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 \( \text{biab}' \) using the following correctness lemma, stating that \( \text{biab}' \) produces a suitable frame and anti-frame so as to adapt a specification \( \models (\text{ok} : Q) C \langle \psi \rangle \) to use a different precondition \( P \).

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

\[
\models (\text{ok} : P * M) C \langle \psi' \rangle
\]

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

\[
\Delta * M' \not\subseteq \Delta * M[\vec{e}/\vec{Y}]
\]

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:

\[
P * M' \not\subseteq \exists \vec{Z}.\Delta * M[\vec{e}/\vec{Y}]
\]  \hspace{1cm} (1)

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

\[
\Delta * M \not\subseteq Q * F
\]
In Figure 5, we assumed all the logical variables used are fresh, so Δ must be disjoint from Y (the free variables of Q and ψ), and therefore Δ[\bar{e}/\bar{Y}] = Δ, so substituting both sides, we get:

\[ \Delta \cup M[\bar{e}/\bar{Y}] = (Q \cup F)[\bar{e}/\bar{Y}] \]

We also weaken the right hand side by replacing x with fresh existentially quantified variables in F.

\[ \Delta \cup M[\bar{e}/\bar{Y}] = (Q \cup \exists X.F[\bar{X}/\bar{x}])[\bar{e}/\bar{Y}] \]

Now, we can existentially quantify both sides of the entailment. Since logical variables are fresh, \(Z\) is disjoint from Q.

\[ \exists Z. \Delta \cup M[\bar{e}/\bar{Y}] = (Q \cup \exists Z.X.F[\bar{X}/\bar{x}])[\bar{e}/\bar{Y}] \]

(2)

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

\[ P \cup M' = (Q \cup \exists Z.X.F[\bar{X}/\bar{x}])[\bar{e}/\bar{Y}] \]

(3)

Now, given that \(\vdash \langle \text{ok} : Q \rangle C \langle \psi \rangle\), we can use the frame rule to get:

\[ \vdash \langle \text{ok} : Q \cup \exists Z.X.F[\bar{X}/\bar{x}] \rangle C \langle \psi \cup \exists Z.X.F[\bar{X}/\bar{x}] \rangle \]

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

\[ \vdash \langle \text{ok} : (Q \cup \exists Z.X.F[\bar{X}/\bar{x}])[\bar{e}/\bar{Y}] \rangle C \langle (\psi \cup \exists Z.X.F[\bar{X}/\bar{x}])[\bar{e}/\bar{Y}] \rangle \]

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

\[ \vdash \langle \text{ok} : P \cup M' \rangle C \langle (\psi \cup \exists Z.X.F[\bar{X}/\bar{x}])[\bar{e}/\bar{Y}] \rangle \]

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, C₁ and C₂.

**Lemma E.2.** If \((M, \psi_1, \psi_2) \in \text{ triab’(M₁, M₂, ψ₁, ψ₂, x)}\) and \(\vdash \langle \psi_1 \odot M_1 \rangle C \langle \psi_1 \rangle\) and \(\vdash \langle \psi_2 \odot M_2 \rangle C \langle \psi_2 \rangle\) and \(x = \text{mod}(C)\), then \(\vdash \langle \psi_1 \odot M \rangle C \langle \psi_1 \rangle\) and \(\vdash \langle \psi_2 \odot M \rangle C \langle \psi_2 \rangle\).

**Proof.** By definition, any element of triab’(M₁, M₂, ψ₁, ψ₂, x) will have the form:

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

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

\[ \vdash \langle \psi_1 \odot (M_i \odot \exists X.F_i[\bar{X}/\bar{x}]) \rangle C \langle \psi_1 \odot \exists X.F_i[\bar{X}/\bar{x}] \rangle \]
This is valid since \( \exists X. F_i[X/x] \) is disjoint from \( x \) (the modified program variables) by construction. We also assumed in rename that \( \theta \) is disjoint from \( x \), so we can substitute \( \theta \) for \( \hat{Y} \) to get:

\[
\vdash ((\varphi_i \otimes (M_i \mapsto \exists X. F_i[X/x])))[\overline{\theta}/\hat{Y}]) C \langle (\psi_i \otimes \exists X. F_i[X/x])i[\overline{\theta}/\hat{Y}] \rangle
\]

Note that \( \varphi_i[\overline{\theta}/\hat{Y}] = \varphi_i \), since the logical variables \( \hat{Y} \) are generated freshly, independent of \( \varphi_i \), as was mentioned in Figure 5. So, using the rule of consequence we get:

\[
\vdash (\varphi_i \otimes M) C \langle (\psi_i \otimes \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}] \rangle
\]

\[\square\]

**Lemma E.3.** If \((M, \psi'_1, \psi'_2) \in \text{triab}^\prime(P_1, P_2, \psi_1, \psi_2, \tilde{x})\) and \(\vdash \langle \text{ok} : P_1 \rangle C \langle \psi_1 \rangle and \vdash \langle \text{ok} : P_2 \rangle C \langle \psi_2 \rangle\) and \(\tilde{x} = \text{mod}(C_1, C_2)\), then \(\vdash \langle \text{ok} : M \rangle C \langle \psi'_1 \rangle and \vdash \langle \text{ok} : M \rangle C \langle \psi'_2 \rangle\).

**Proof.** By definition, any element of \(\text{triab}^\prime(P_1, P_2, \psi_1, \psi_2, \tilde{x})\) will have the form:

\[
(M', (\psi_1 \otimes \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}], (\psi_2 \otimes \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}])
\]

Where \((\tilde{e}, \tilde{Y}, M') = \text{rename}(\text{emp}, M, \{\psi_1, \psi_2\}, \emptyset, \tilde{x})\) and \((M, F_1, F_2) \in \text{triab}(P_1, P_2)\). From rename, we know that \(M' \mapsto M[\tilde{e}/\tilde{Y}]\) and from tri-abduction, we know that \(M \not\equiv P_i \cdot F_i\) for \(i = 1, 2\), so \(M' \equiv (P_i \cdot F_i)[\overline{\theta}/\hat{Y}]\). We can weaken this by replacing \(\tilde{x}\) in \(F_i\) with fresh existentially quantified variables to obtain \(M' \equiv (P_i \cdot \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}]\) by substituting into both the pre and postconditions, we get \(\vdash \langle \text{ok} : P_i \cdot \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}] \rangle C_i \langle (\psi_i \otimes \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}] \rangle\) (this is valid since rename guarantees that \(\tilde{e}\) is disjoint from \(x\)). Finally, we complete the proof by applying the rule of consequence with \(M' \equiv (P_i \cdot \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}]\) to obtain:

\[
\vdash (\text{ok} : M') C_i \langle (\psi_i \otimes \exists X. F_i[X/x])[\overline{\theta}/\hat{Y}] \rangle
\]

\[\square\]

### E.2 Sequencing Proof

**Lemma 6.2 (Seq).** If \((M, \psi) \in \text{seq}(\varphi, S, \tilde{x})\), \(\tilde{x} = \text{mod}(C)\), and \(\vdash \langle \text{ok} : P \rangle C \langle \emptyset \rangle\) for all \((P, \emptyset) \in S\), then \(\vdash \langle \varphi \otimes M \rangle C \langle \psi \rangle\).

**Proof.** By induction on the structure of \(\varphi\).

- \(\varphi = \top\). We need to show that \(\vdash \langle \top \otimes \text{emp} \rangle C \langle \top \rangle\) holds. This triple is clearly valid since any triple with the postcondition \(\top\) is trivially true.

- \(\varphi = \top^\circ\). We need to show that \(\vdash \langle \top^\circ \otimes \text{emp} \rangle C \langle \top^\circ \rangle\) holds. This triple is clearly valid since only \(\emptyset\) satisfies the precondition and \([C]^\top(\emptyset) = \emptyset\).

- \(\varphi = \phi_1 \bowtie \phi_2\) where \(\bowtie \in \{\lor, \bowtie_a\}\). We need to show:

\[
\vdash \langle (\phi_1 \bowtie \phi_2) \otimes M \rangle C \langle \psi'_1 \bowtie \psi'_2 \rangle
\]

Where \((M, \psi'_1, \psi'_2) \in \text{triab}^\prime(M_1, M_2, \psi_1, \psi_2, \tilde{x})\) and \((M_i, \psi_i) \in \text{seq}(\varphi_i, S, \tilde{x})\) for each \(i \in \{1, 2\}\). By the induction hypothesis, we know that \(\vdash \langle \phi_i \otimes M_i \rangle C \langle \psi_i \rangle\), so by Lemma E.2 we get that \(\vdash \langle \phi_i \otimes M \rangle C \langle \psi'_i \rangle\). We now complete the proof separately for the two logical operators:

- \(\varphi = \phi_1 \lor \phi_2\). Suppose that \(m \equiv \langle \phi_1 \lor \phi_2 \rangle \otimes M\), so \(m \equiv \phi_i \otimes M\) for some \(i \in \{1, 2\}\). Since \(\vdash \langle \phi_i \otimes M \rangle C \langle \psi'_i \rangle\), we know that \([C]^\top(m) \equiv \psi'_i\), and we can weaken this to conclude that \([C]^\top(m) \equiv \psi'_1 \lor \psi'_2\).
\( \varphi = \varphi_1 \oplus_a \varphi_2 \). Suppose that \( m \models (\varphi_1 \oplus_a \varphi_2) \otimes M \), and so there are \( m_1 \) and \( m_2 \) such that \( m = a \cdot m_1 + \bar{a} \cdot m_2 \) and \( m_i \models \varphi_i \otimes M \) for each \( i \). Since \( \models (\varphi_i \otimes M) \ C \langle \psi'_i \rangle \), we know that \( \mathbb{E}(m_i) \not\models \psi'_i \). We also know that \( \mathbb{E}(m) = \mathbb{E}(a \cdot m_1 + \bar{a} \cdot m_2) = a \cdot \mathbb{E}(m_1) + \bar{a} \cdot \mathbb{E}(m_2) \), and so \( \mathbb{E}(m) \not\models \psi'_i \oplus_a \psi'_j \).

\( \varphi = \text{ok} : P \). We need to show that \( \models \langle \text{ok} : P \otimes M \rangle \ C \langle \psi \rangle \) where \( (M, \psi') \in \text{biab}(P, Q, \psi, \bar{x}) \) and \( (Q, \psi) \in S \). By assumption, we know that \( \models \langle \text{ok} : Q \rangle \ C \langle \psi \rangle \). The remainder of the proof follows directly from Lemma E.1.

\( \varphi = \text{er} : Q \). We need to show that \( \models \langle \text{er} : Q \rangle \ C \langle \text{er} : Q \rangle \). This trivially holds since any \( m \) satisfying \( \text{er} : Q \) must consist only of \( \text{I}_{\text{er}}(s, h) \) states, and so \( \mathbb{E}(m) = m \).

\[ \square \]

### E.3 Symbolic Execution Proofs

**Lemma E.4.** Let:

\[
f(S) = \{(\neg e \land \text{emp}, \text{ok} : \neg e \land \text{emp})\} \cup \\
\{(M_1 \cdot M_2 \land e, \psi) \mid (M_1, \varphi) \in \text{seq}(e \land \text{emp}, \mathbb{E}(T), \text{mod}(C)), (M_2, \psi) \in \text{seq}(\varphi, S, \text{mod}(C))\}
\]

For any \( n \in \mathbb{N} \) and \( (P, \varphi) \in f^n(0) \), \( \models \langle \text{ok} : P \rangle \) while \( e \) do \( C \langle \varphi \rangle \).

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

\[
f^{n+1}(0) = f(f^n(0)) = \{(\neg e \land \text{emp}, \text{ok} : \neg e \land \text{emp})\} \cup \\
\{(M_1 \cdot M_2 \land e, \psi) \mid (M_1, \varphi) \in \text{seq}(e \land \text{emp}, \mathbb{E}(T), \text{mod}(C)), (M_2, \psi) \in \text{seq}(\varphi, f^n(0), \text{mod}(C))\}
\]

So any \( (P, \varphi) \in f^{n+1}(0) \) comes from one of the two sets in the above union. Suppose it is in the first, so we need to show that \( \models \langle \text{ok} : \neg e \land \text{emp} \rangle \) while \( e \) do \( C \langle \text{ok} : \neg e \land \text{emp} \rangle \). 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 \( (M_1 \cdot M_2 \land e, \psi) \) where \( (M_1, \varphi) \in \text{seq}(e \land \text{emp}, \mathbb{E}(T), \text{mod}(C)) \) and \( (M_2, \psi) \in \text{seq}(\varphi, f^n(0), \text{mod}(C)) \). By Lemma C.7, we know that \( \models \langle \text{ok} : M_1 \land e \rangle \ C \langle \varphi \rangle \) and by Lemma C.7 and the induction hypothesis, we get \( \models \langle \varphi \otimes M_2 \rangle \) while \( e \) do \( C \langle \psi \rangle \). Using the frame rule, we also get \( \models \langle \text{ok} : M_1 \cdot M_2 \land e \rangle \ C \langle \varphi \otimes M_2 \rangle \), and so we can sequence the previous specifications to get \( \models \langle \text{ok} : M_1 \cdot M_2 \land e \rangle \ C \langle \psi \rangle \) 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 \( m \models \langle \text{ok} : M_1 \cdot M_2 \land e \rangle \), \( \mathbb{E} \langle m \rangle = \mathbb{E} \langle \text{while} e \text{ do } C \langle \psi \rangle \rangle \), and so \( \models \langle \text{ok} : M_1 \cdot M_2 \land e \rangle \) while \( e \) do \( C \langle \psi \rangle \).

\[ \square \]

**Lemma E.5.** If for every \( (s, h) \models P \), there exists \( s' \) and \( t' \) such that \( \mathbb{E}(s, h) = \text{unit}_{\mathbb{W}}(\mathbb{I}_e(s', t')) \) and \( (s', h') \models Q \), then \( \models \langle \text{ok} : P \rangle \ C \langle \epsilon : Q \rangle \)

**Proof.** Suppose that \( m \models \text{ok} : P \). That means that \( m = 1 \) and all elements of \( \text{supp}(m) \) have the form \( \mathbb{I}_e(s, h) \) where \( (s, h) \models P \). By assumption, we know that \( \mathbb{E}(s, h) = \text{unit}_{\mathbb{W}}(\mathbb{I}_e(s', t')) \) such that \( (s', t') \models Q \). This means that every element of \( \mathbb{E}(m) \) must have the form \( \mathbb{I}_e(s', t') \) where \( (s', t') \models Q \) and also \( \mathbb{E}(m) = 1 \) since each \( (s, h) \) does not change the mass of the distribution, so \( \mathbb{E}(m) \models \epsilon : Q \).

\[ \square \]

**Theorem 6.1 (Symbolic Execution Soundness).** If \( (P, \varphi) \in \mathbb{E}(T) \), then \( \models \langle \text{ok} : P \rangle \ C \langle \varphi \rangle \)
Proof. By induction on the structure of the program $C$.

- $C =$ skip. We need to show that $\vdash \langle ok : \text{emp} \rangle \text{skip} \langle ok : \text{emp} \rangle$, which is trivially true.

- $C = C_1 +_t C_2$. By definition, any element of $[C_1 +_t C_2]^= (T)$ must have the form $(P \oplus M, \psi)$ where $(P, \varphi) \in [C_1]^= (T)$ and $(M, \psi) \in \text{seq}(\varphi, [C_2]^= (T), \text{mod}(C_2))$. By the induction hypothesis, we know that $\vdash \langle ok : P \rangle \ C_1 \langle \varphi \rangle$ and by Lemma C.7 we know that $\vdash \langle \varphi \otimes M \rangle \ C_2 \langle \psi \rangle$. Using the frame rule, we get that $\vdash \langle ok : P \oplus M \rangle \ C_1 \langle \varphi \otimes M \rangle$ (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 $\vdash \langle ok : P \oplus M \rangle \ C_1 \langle \varphi \rangle \ C_2 \langle \psi \rangle$.

- $C =$ if $e$ then $C_1$ else $C_2$. Any element of $[\text{if } e \text{ then } C_1 \text{ else } C_2]^= (T)$ must either have the form $(P \land e, \varphi)$ where $(P, \varphi) \in \text{seq}(e, [C_1]^= (T), \text{mod}(C_1))$ or $(P \land \neg e, \varphi)$ where $(P, \varphi) \in \text{seq}(\neg e, [C_2]^= (T), \text{mod}(C_2))$. Suppose the former, then by the induction hypothesis and Lemma C.7, we know that $\vdash \langle ok : P \land e \rangle \ C_1 \langle \varphi \rangle$. Since $e$ must be true for all states satisfying the precondition, the result of running $C_1$ is the same as running if $e$ then $C_1$ else $C_2$ (the true branch will always be taken), so $\vdash \langle ok : P \land e \rangle$ if $e$ then $C_1$ else $C_2 \langle \psi \rangle$. The case for the false branch is symmetric.

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

- $C =$ $C_1 +_t C_2$. Any element of $[C_1 +_t C_2]^= (T)$ must have the form $(M, \psi') \oplus a \psi''$ where $(M, \psi', \psi'') \in \text{triab}'(M_1, M_2, \psi_1, \psi_2, \text{mod}(C_1, C_2))$ and $(M_1, \psi_1) \in [C_1]^= (T)$ and $(M_2, \psi_2) \in [C_2]^= (T)$. By the induction hypothesis, we know that $\vdash \langle ok : M_1 \rangle \ C_1 \langle \psi_1 \rangle$ for $i = 1, 2$. By Lemma E.3 we know that $\vdash \langle ok : M \rangle \ C_1 \langle \psi'' \rangle$. Now, we show that $\vdash \langle M \rangle \ C_1 +_t C_2 \langle \psi' \oplus a \psi'' \rangle$: suppose $m \in M$. By definition, $[C_1 +_t C_2]^= (m) = a \cdot [C_1]^= (m) + \alpha \cdot [C_2]^= (m)$. Now, using what we obtained from Lemma E.3, we know that since $m \in M$, $[C_i]^= (m) \in \psi''$ for each $i \in \{1, 2\}$. Combining these two, we get that $a \cdot [C_1]^= (m) + \alpha \cdot [C_2]^= (m) \in \psi' \oplus a \psi''$.

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 $(P, \varphi) \in [c]^= (T)$, then $[c] (s, h) \equiv \varphi$ for all $(s, h) \in P$ by Lemma E.5.

- $C =$ $(x := e)$. Suppose that $(s, h) \vdash ok : x = X \land \text{emp}$, so $s(x) = s(X)$ and $h = \emptyset$. Now, $[x := e] (s, h) = \text{unit}(s[x \mapsto [e] (s)], h)$, 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 $(s', h) \vdash ok : x = e[X/x] \land \text{emp}$. The remainder of the proof follows by Lemma E.5.

- $C =$ $(x := \text{alloc})$. We need to show that $\vdash \langle ok : \text{emp} \land x = X \rangle \ x := \text{alloc} \langle ok : \exists Y. x \mapsto Y \rangle$. Suppose that $m \in M \land x = X$, so each state in $m$ has the form $1_{ok}(s, h)$ where $(s, h) \vdash \text{emp} \land x = X$, so $s(x) = s(X)$ and $h = \emptyset$. We know that $[x := \text{alloc}] (s, h) = \text{bind}_{\text{alloc}}(s(h), \lambda f.\text{unit}(s[x \mapsto \ell], h[\ell \mapsto 0]))$ where $\ell$ does not appear in $s$ or $h$. Let $s' = s[x \mapsto \ell]$ and $h' = h[\ell \mapsto 0]$. Clearly, $h'([x] (s')) = h'([\ell] = 0)$, 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 $[x := \text{alloc}]^= (m) \vdash \exists Y. x \mapsto Y$.

- $C =$ free$(x)$. There are three cases since specifications for free can take on multiple forms. In the first case, we need to show that $\vdash \langle ok : e \mapsto X \rangle \text{free}(e) \langle ok : e \not\mapsto \rangle$. Suppose $(s, h) \vdash e \mapsto X$, so $h([e] (s)) = s(X)$. We also know that $[\text{free}(e)] (s, h) = \text{unit}(s(h)[[e] (s) \mapsto \bot])$, 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:

\[ \langle \text{ok} : e \mapsto \rangle \text{free}(e) \langle \text{er} : e \mapsto \rangle \quad \text{and} \quad \langle \text{ok} : e = \text{null} \rangle \text{free}(e) \langle \text{er} : e = \text{null} \rangle \]

Suppose \((s, h) \models e \mapsto \) and so \(h([e] (s)) = \bot\). Clearly, \([\text{free}(e)] (s, h) = \text{error}(s, h)\), so by Lemma E.5 the claim holds. The case where \(e = \text{null}\) is nearly identical.

\[ C = [e]_1 \leftarrow e_2. \] There are three cases, first we must show that \(\langle \text{ok} : e_1 \mapsto X \rangle [e_1] \leftarrow e_2 \langle \text{ok} : e_1 \mapsto e_2 \rangle\). Suppose \((s, h) \models e_1 \mapsto X\), so \(h([e_1] (s)) = s(X)\) and therefore that memory address is allocated since \(s(X) \in \text{Val}\). This means that \([e_1] \leftarrow e_2\), \((s, h)[[e_1] (s) \mapsto [e_2] (s))\) and clearly \((s, h)[[e_1] (s) \mapsto [e_2] (s)) \models e_1 \mapsto e_2\) by definition, so the claim holds by Lemma E.5.

In the remaining case, we must show that \(\langle \text{ok} : e_1 \mapsto \rangle [e_1] \leftarrow e_2 \langle \text{er} : e_1 \mapsto \rangle\) and \(\langle \text{ok} : e_1 = \text{null} \rangle [e_1] \leftarrow e_2 \langle \text{er} : e_1 = \text{null} \rangle\). The proof is similar to the second case for \(\text{free}(e)\).

\[ C = x \leftarrow [e]. \] There are three cases, first we must show that \(\langle \text{ok} : x = X \land e \mapsto Y \rangle x \leftarrow [e] \langle \text{ok} : x = X \land e \mapsto Y \rangle x \leftarrow [e]\). Suppose \((s, h) \models x = X \land e \mapsto Y\), so \(s(x) = s(X)\) and \(h([e] (s)) = s(Y)\). We also know that \([x \leftarrow [e]] (s, h) = \text{unit}(s[x \mapsto h([e] (s)), h]\). 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 \(h([e[X/x]] (s')) = h([e] (s)) = \text{unit}(s)\) and \(s'(x) = [e] (s) = s(Y), \) so clearly \((s', h) \models x = X \land e \mapsto Y \langle \text{ok} : e \mapsto \rangle x \leftarrow [e] \langle \text{er} : e \mapsto \rangle\) and \(\langle \text{ok} : e = \text{null} \rangle x \leftarrow [e] \langle \text{er} : e = \text{null} \rangle\). The proof is similar to the second case for \(\text{free}(e)\).

\[ C = f(\vec{e}). \] Any element of \([f(\vec{e})]^T\) must have the form \((P \land \vec{x} = \vec{x'}) (P, \varphi) \in \text{seq}(\vec{x} = \vec{c}[X/x], T(f(\vec{e})), \text{mod}(f))\). By Lemma C.7, we get:

\[ \models \langle \text{ok} : P \land \vec{x} = \vec{c}[X/x] \rangle f(\vec{x}) \langle \varphi \rangle \]

Now, we need to show that \(\models \langle \text{ok} : P \land \vec{x} = \vec{x'} \rangle f(\vec{x}) \langle \varphi \rangle\). Suppose \([m \models P \land \vec{x} = \vec{x'}] = [C] \langle \varphi \rangle\). Let \(m'\) be obtained by taking every state \(\text{id}_{ok}(s, h) \in \text{supp}(m)\) and modifying it to be \(\text{id}_{ok}(s[\vec{x} \mapsto \vec{z}])\). By a similar argument to the \(x := e\) case, we know that \([m' \models \text{ok} : \vec{x} = \vec{c}[X/x]] = [C] \langle \varphi \rangle\). We know \(P\) is disjoint from the program variables by the definition of \(\text{seq}\), so \(m' \models \text{ok} : P\) as well, since \(m \models \text{ok} : P\) and the only difference between \(m\) and \(m'\) is updates to the program variables. Let \(C\) be the body of \(f\) and note that \([f(\vec{x})]^T(m') = [C]^T(m')\) since the initial modification of the program state is just updating variable values to themselves. We know from \(\models \langle \text{ok} : P \land \vec{x} = \vec{c}[X/x] \rangle f(\vec{x}) \langle \varphi \rangle\) that \([C]^T(m') \models \varphi\) and by definition, \([f(\vec{e})]^T(m) = [C]^T(m')\), so \([f(\vec{e})]^T(m) \models \varphi\).

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

- Single Path. Any element of \([C_1 +_a C_2]^T\) has one of two forms. In the first case, we need to show that \(\models \langle \text{ok} : P \rangle C_1 +_a C_2 \langle \varphi \oplus_a T \rangle\) given that \(\models \langle \text{ok} : P \rangle C_1 \langle \varphi \rangle\). Suppose that \(m \models \text{ok} : P\). By our assumption, we know that \([C_1]^T(m) \models \varphi\). Now, \([C_1 +_a C_2]^T(m) = a \cdot [C_1]^T(m) + \vec{a} \cdot [C_2]^T(m)\) and clearly \([C_2]^T(m) \models T\), so \([C_1 +_a C_2]^T(m) \models \varphi \oplus_a T\). The second case is symmetrical, using the fact that \(+ \) is commutative and \(\vec{a} = a\).

- Loop invariants. We need to show that \(\models \langle \text{ok} : I \rangle\) while \(e \neq 0\) for \(C\langle \langle \text{ok} : I \land \neg e \rangle \lor T^\oplus \rangle\) given that \(\models \langle \text{ok} : I \land e \rangle\) \(C\langle \text{ok} : I \rangle\). Note that this case is only valid for deterministic or nondeterministic programs (not probabilistic ones). Suppose \(m \models \text{ok} : I\), so every state in \(\text{supp}(m)\) has the form \(\text{id}_{ok}(s, h)\) where \((s, h) \models I\). 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 \neq 0 \text{ do } C\] \((s, h)\) must satisfy \(I\) and \(\neg e\),...
or there are no terminating states. In other words, \( [\text{while } e \text{ do } C] \ (s, h) \vdash (\text{ok} : I \land \neg e) \lor \top \). 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 \((\text{ok} : I \land \neg e) \lor \top\), then the union of all these states will also satisfy \((\text{ok} : I \land \neg e) \lor \top\).

\(\square\)