Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning

NOAM ZILBERSTEIN, Cornell University, USA
DEREK DREYER, MPI-SWS, Germany
ALEXANDRA SILVA, Cornell University, USA

Program logics for bug-finding (such as the recently introduced Incorrectness Logic) have framed correctness and incorrectness as dual concepts requiring different logical foundations. In this paper, we argue that a single unified theory can be used for both correctness and incorrectness reasoning. We present Outcome Logic (OL), a novel generalization of Hoare Logic that is both *monadic* (to capture computational effects) and *monoidal* (to reason about outcomes and reachability). OL expresses true positive bugs, while retaining *correctness* reasoning abilities as well. To formalize the applicability of OL to both correctness and incorrectness, we prove that any false OL specification can be disproven in OL itself. We also use our framework to reason about new types of incorrectness in nondeterministic and probabilistic programs. Given these advances, we advocate for OL as a new foundational theory of correctness and incorrectness.

CCS Concepts: • **Theory of computation** → **Hoare logic**; *Separation logic*; **Logic and verification**; **Program specifications**.

Additional Key Words and Phrases: Program Logics, Hoare Logic, Incorrectness Reasoning

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

> *"Program correctness and incorrectness are two sides of the same coin."* – O'Hearn [2019]

1 INTRODUCTION

Developing formal methods to prove program *correctness*—the *absence* of bugs—has been a holy grail in program logic and static analysis research for many decades. However, seeing as many static analyses deployed in practice are *bug-finding* tools, O'Hearn [2019] recently advocated for the development of formal methods for proving program *incorrectness*; we need expressive, efficient, and compositional ways to reliably identify the *presence* of bugs as well.

The aforementioned paper of O'Hearn [2019] proposed Incorrectness Logic (IL) as a logical foundation for reasoning about program incorrectness. IL is inspired by—and in a precise technical sense *dual to*—Hoare Logic. Like Hoare Logic, IL specifications are *compositional*, given in terms of preconditions P and postconditions Q. Hoare Triples $\{P\}\ C\ \{Q\}$ stipulate that the result of running the program C on any state satisfying P will be a state that satisfies Q. Incorrectness Triples

Authors' addresses: Noam Zilberstein, noamz@cs.cornell.edu, Cornell University, USA; Derek Dreyer, dreyer@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Alexandra Silva, alexandra.silva@cornell.edu, Cornell University, USA.

2475-1421/2023/4-ART93
https://doi.org/10.1145/3586045

$[P]\ C\ [Q]$ *go in reverse*—all states satisfying Q must be reachable from some state satisfying P.

$$\begin{array}{llllll}\textbf{Hoare Logic:} & \vDash \{P\}\ C\ \{Q\} & \text{iff} & \forall\sigma \vDash P.\quad \forall\tau. & \tau \in [\![C]\!](\sigma) & \Rightarrow \quad \tau \vDash Q \\ \textbf{Incorrectness Logic:} & \vDash [P]\ C\ [Q] & \text{iff} & \forall\tau \vDash Q.\quad \exists\sigma. & \tau \in [\![C]\!](\sigma) & \text{and} \quad \sigma \vDash P\end{array}$$

Practically speaking, IL differs from Hoare Logic in two key ways. First, whereas Hoare Logic has no false negatives (*i.e., all* executions of a verified program behave correctly), IL has *no false positives*: any bug found using IL is in fact reachable by *some* execution of the program. Second, whereas Hoare Logic is over-approximate, IL is *under-approximate*: to prove that a program is incorrect, one only needs to specify (in the postcondition) a *subset* of the possible outcomes, which helps to ensure the efficiency of large-scale analyses. Subsequent work has focused on extending IL to account for a variety of program errors (*e.g.,* memory errors, memory leaks, data races, and deadlocks) and on using the resulting Incorrectness Separation Logics (ISLs) to explain and inform the development of bug-catching static analyses [Le et al. 2022; Raad et al. 2020, 2022].

Despite these exciting advances, we argue that the foundations of incorrectness reasoning are still far from settled—and worthy of reconsideration. IL achieves *true positives* (reachability of end-states) and *under-approximation* through the same mechanism: quantification over all states that satisfy the postcondition. However, this conflation of concepts leads to several problems:

Expressivity. The semantics of IL only encompasses under-approximate types of incorrectness, which does not fully account for all bugs that may be encountered in real programs. For example, as we will see in Section 2.2, IL can be used to show the *reachability* of bad states, but it cannot prove *unreachability* of good states.

Generality. IL is not amenable to probabilistic execution models and therefore is not a good fit for reasoning about incorrectness in randomized programs (Section 7.2).

Error Reporting. IL cannot easily describe what conditions are *sufficient* to trigger a bug (Section 6.6), meaning that analyses based on IL must implement extra algorithmic checks to determine whether a bug is worth reporting [Le et al. 2022].

Our key insight is that reachability and under-approximation are separate concepts that can (and should) be handled independently. But once reachability is separated from under-approximation, the resulting program logic no longer applies only to bug-finding. In this paper, we show how the full spectrum of correctness and incorrectness reasoning can be achieved with a unified foundation: a generalization of "good old" Hoare Logic that we call ***Outcome Logic (OL)***. In addition to consolidating the foundations of incorrectness with traditional correctness reasoning, OL overcomes all the aforementioned drawbacks of IL.

In OL, assertions are no longer predicates over program states, but rather predicates on an *outcome monoid*, whose elements can be, for instance, *sets of program states* or *probability distributions on program states*. The monoidal structure enables us to model a new *outcome conjunction*, $P \oplus Q$, asserting that the predicates P and Q each hold in *reachable* executions (or hold in subdistributions on program executions). We can also under-approximate by joining a predicate with $\top$, the trivial outcome: $P \oplus \top$ states that P only partially covers the program outcomes. OL offers several advantages as a unifying foundation for correctness and incorrectness:

Generality. OL unifies program analysis across two dimensions. First, since any untrue OL spec can be disproven in OL (Theorem 5.1), correctness and incorrectness reasoning are possible in a single program logic. Second, OL uses a monadic semantics which allows it to be instantiated for different evaluation models such as nondeterminism, erroneous termination, and probabilistic choice, thereby unifying correctness and incorrectness reasoning across execution models.

Beyond Reachability. Until now, the study of incorrectness has revolved primarily around *reachability* of crash states. We prove that OL handles a broader characterization of incorrectness than IL in nondeterministic programs (Theorem 5.6), as well as probabilistic incorrectness (Theorem 5.10).

Manifest Errors. In order to improve fix rates in automated bug finding tools, Le et al. [2022] only report bugs that occur regardless of context. These bugs—called *manifest errors*—are not straightforward to characterize using Incorrectness Logic: an auxiliary algorithm is needed to check whether some bug is truly a manifest error. In contrast, manifest errors are trivial to characterize in OL—Le et al.'s [2022] original definition can be expressed as an OL triple (Lemma 6.7).

The contributions of the paper are as follows:

- We provide an overview of the semantics of IL and explain what is needed in order to characterize broader classes of errors (Section 2). We show how reasoning about outcomes can account for reachability of end-states and enable under-approximation (when desired).
- We define Outcome Logic formally (Section 3 and Section 4), parametric on a monad and an assertion logic. We define syntax and semantics of the logic, using Bunched Implications (BI) formulae for pre- and postconditions, and provide inference rules to reason about validity.
- We show that OL is suitable for both correctness and incorrectness reasoning by proving that false OL triples can be disproven within OL (Section 5). As a corollary, OL can disprove Hoare triples, which was one motivation for IL (Corollary 5.7). We go further and show three kinds of incorrectness that can be captured in OL, only one of which is expressible in IL (Section 5.1).
- We exemplify how OL can be instantiated to find memory errors (Section 6) and probabilistic bugs (Section 7). We argue that the latter use case is not feasible in IL (Section 7.2).

Finally, we conclude in Section 8 and Section 9 by discussing related work and next steps.

2 OVERVIEW: A LANDSCAPE OF TRIPLES

The study of incorrectness has made apparent the need for new program logics that guarantee true positives and support under-approximate reasoning, since standard Hoare Logic—which does not enjoy those properties—is incapable of proving the presence of bugs. Concretely, in a valid Hoare Triple, denoted $\vDash \{P\}\ C\ \{Q\}$, running the program C in any state satisfying the precondition P will result in a state satisfying the postcondition Q (the formal definition is given in Figure 1). Suppose we wanted to use such a triple to prove that the program $x := \mathsf{malloc}()\ ⨾\ [x] \leftarrow 1$ has a bug (malloc may nondeterministically return null, causing the program to crash with a segmentation fault when the subsequent command attempts to store the value 1 at the location pointed to by x). We might be tempted to specify the triple as follows:

$$\{\mathsf{true}\}\ x := \mathsf{malloc}()\ ⨾\ [x] \leftarrow 1\ \{(\mathsf{ok} : x \mapsto 1) \lor (\mathsf{er} : x = \mathsf{null})\} \tag{1}$$

Here, the assertion $(\mathsf{ok} : p)$ means that the program terminated successfully in a state satisfying p and $(\mathsf{er} : q)$ means that it crashed in a state satisfying q. However, this is not quite right. According to the semantics of Hoare Logic, every possible end state must be covered by the postcondition, hence the need to use a disjunction to indicate that two outcomes are possible. But since we do not know that every state described by the postcondition is *reachable*, it is possible that every program trace ends up satisfying the first disjunct $(\mathsf{ok} : x \mapsto 1)$ and the error state is never reached.

Incorrectness Logic offers a solution to this problem. In a valid Incorrectness Triple, $\vDash [P]\ C\ [Q]$, every state satisfying Q is reachable by running C in some state satisfying P. So, simply switching the triple type in the above example *does* give us a witness that the error is possible.

$$[\mathsf{true}]\ x := \mathsf{malloc}()\ ⨾\ [x] \leftarrow 1\ [(\mathsf{ok} : x \mapsto 1) \lor (\mathsf{er} : x = \mathsf{null})] \tag{2}$$

Triple Name	Syntax		Semantics				
Hoare Logic	$\vDash \{P\}\ C\ \{Q\}$	iff	$\forall \sigma \vDash P.$	$\forall \tau.$	$\tau \in [\![C]\!](\sigma)$	$\Rightarrow$	$\tau \vDash Q$
Incorrectness Logic (IL) / Reverse Hoare Logic (RHL)	$\vDash [P]\ C\ [Q]$	iff	$\forall \tau \vDash Q.$	$\exists \sigma.$	$\tau \in [\![C]\!](\sigma)$	and	$\sigma \vDash P$
Outcome Logic (OL)	$\vDash \langle P \rangle\ C\ \langle Q \rangle$	iff	$\forall m.$	$m \vDash P$	$\Longrightarrow$		$[\![C]\!]^\dagger(m) \vDash Q$

Fig. 1. Semantics of triples where P and Q are logical formulae, C is a program, Σ is the set of all program states, $\sigma, \tau \in \Sigma$, and $[\![C]\!] : \Sigma \to 2^\Sigma$ is the reachable states function. In the last line of the table, M is a monad, $m \in M\Sigma$ and $[\![C]\!]^\dagger : M\Sigma \to M\Sigma$ is the monadic lifting of $[\![\cdot]\!] : \Sigma \to M\Sigma$.

Though the conclusion remains a disjunction, the semantics of the incorrectness triple (Figure 1) ensures that *every* state in the disjunction is reachable. Moreover, we can under-approximate by *dropping disjuncts* from the postcondition and use the simpler specification:

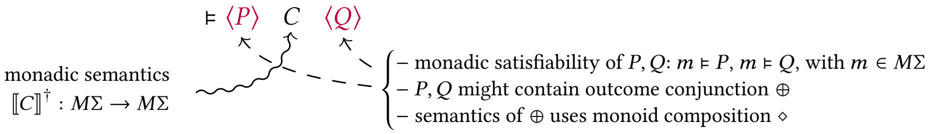
$$[\mathsf{true}]\ x := \mathsf{malloc}() \fatsemi [x] \leftarrow 1\ [\mathsf{er} : x = \mathsf{null}] \tag{3}$$

This more parsimonious specification still witnesses the error while also helping to ensure efficiency of large-scale automated analyses, which must keep descriptions at each program point small.

The duality between Hoare Logic and Incorrectness Logic appears sensible. Hoare Logic has *no false negatives*—a program is only correct if we account for all the possible outcomes. Incorrectness Logic has *no false positives*—an error is only worth reporting if it is truly reachable. However, we argue in this paper that incorrectness reasoning and Hoare Logic are *not* in fact at odds: an approach to incorrectness that is more similar to Hoare Logic is not only possible but, in fact, advantageous for several reasons, including the ability to express when an error will be *manifest* and the ability to reason about additional varieties of incorrectness.

2.1 Unifying Correctness and Incorrectness

Our first insight is that the inability to prove the existence of bugs is not inherent in the semantics of Hoare Logic. Rather, it is the result of an assertion logic that is not expressive enough to reason about reachability. Triple (1) shows how the usual logical disjunction is inadequate in reaching this goal. To remedy this, we use a logic with extra algebraic structure on outcomes, reminiscent of the use of a resource logic in separation logic [O'Hearn et al. 2001; Reynolds 2002]. In this case, resources are program outcomes rather than heap locations. Program outcomes do not necessarily need to be the usual traces in a (non-)deterministic execution model, but can also arise from programs with alternative execution models such as probabilistic computation. To model different types of computations in a uniform way, we use an execution model parametric on a monad. We call this new logic Outcome Logic (OL), with triples denoted by $\langle P \rangle\ C\ \langle Q \rangle$ (defined formally in Figure 1). Let us schematically point out the generalizations in these new triples:


$\vDash \langle P \rangle\ C\ \langle Q \rangle$
monadic semantics
$[\![C]\!]^\dagger : M\Sigma \to M\Sigma$
– monadic satisfiability of P, Q: $m \vDash P$, $m \vDash Q$, with $m \in M\Sigma$
– P, Q might contain outcome conjunction $\oplus$
– semantics of $\oplus$ uses monoid composition $\diamond$

OL triples follow the spirit of Hoare Logic—first quantifying over elements satisfying the precondition and then stipulating that the result of running the program on such an element must satisfy the postcondition. The difference is that in OL triples, the pre- and postconditions are satisfied by a monoidal collection of outcomes $m \in M\Sigma$ rather than individual program states $\sigma \in \Sigma$. This allows us to introduce a new connective in the logic—the *outcome conjunction* $\oplus$—which models program outcomes as resources. Consider the postcondition in triple (2) if we replace $\vee$ by $\oplus$:

$$(\mathsf{ok} : x \mapsto 1) \vee (\mathsf{er} : x = \mathsf{null}) \quad \text{vs.} \quad (\mathsf{ok} : x \mapsto 1) \oplus (\mathsf{er} : x = \mathsf{null})$$

A program state satisfies the first formula just by satisfying one of the disjuncts, whereas the second one requires a collection of states that can be split to witness satisfaction of both. This ability to split outcomes emerges as a requirement that $M\Sigma$ is a (partial commutative) monoid. Given two outcomes $m_1, m_2 \in M\Sigma$, there is an operation $\diamond$ that enables us to combine them $m_1 \diamond m_2 \in M\Sigma$. The satisfiability of $\oplus$ is then defined using $\diamond$ to split the monoidal state:

$$m \vDash P \oplus Q \qquad \text{iff} \qquad \exists m_1, m_2 \in M\Sigma. \quad m = m_1 \diamond m_2 \quad \text{and} \quad m_1 \vDash P \quad \text{and} \quad m_2 \vDash Q$$

Consider instantiating the above to the powerset monad that associates a set A with the set of its subsets 2^A. Given a semantic function $\llbracket C \rrbracket : \Sigma \to 2^\Sigma$ that maps individual start states σ to the set of final states reachable by executing C, we can give a monadic semantics $\llbracket C \rrbracket^\dagger(S) = \bigcup_{\sigma \in S} \llbracket C \rrbracket(\sigma)$ where S is a *set* of start states.[1] The monoid composition $\diamond$ on 2^A is given by set union, which is used compositionally to define satisfiability of $\oplus$ as follows: $S \vDash P \oplus Q$ iff $S_1 \vDash P$ and $S_2 \vDash Q$ such that $S = S_1 \cup S_2$. Given some satisfaction relation for individual program states $\vDash_\Sigma \subseteq \Sigma \times \mathsf{Prop}$, we then define satisfaction of atomic assertions as follows:

$$S \vDash P \qquad \text{iff} \qquad S \neq \emptyset \quad \text{and} \quad \forall \sigma \in S. \, \sigma \vDash_\Sigma P$$

The extra restriction $S \neq \emptyset$ witnesses that P is reachable (and not vacuously satisfied). Putting this all together, we instantiate the generic OL triples (Figure 1) to the powerset monad:

$$\vDash \langle P \rangle \, C \, \langle Q \rangle \qquad \text{iff} \qquad \forall S \in 2^\Sigma. \quad S \vDash P \quad \Rightarrow \quad \llbracket C \rrbracket^\dagger(S) \vDash Q$$

Now, we can revisit the example in triple (2) in OL using $\oplus$ instead of $\vee$:

$$\langle \mathsf{ok} : \mathsf{true} \rangle \; x := \mathsf{malloc}() \,\fatsemi\, [x] \leftarrow 1 \; \langle (\mathsf{ok} : x \mapsto 1) \oplus (\mathsf{er} : x = \mathsf{null}) \rangle \tag{4}$$

This specification *does* witness the bug—for any start state there is at least one end state that satisfies each of the outcomes. However, we are still recording extra, non-erroneous outcomes, which is problematic for a large scale analysis algorithm. Following the example in triple (3), we would like to specify the bug above in a way that mentions only the relevant outcome in the postcondition. We can achieve this by simply weakening the postcondition. According to the semantics above, the following implications hold:

$$S \vDash P \oplus Q \quad \Rightarrow \quad S \vDash P \oplus \top \qquad \text{and} \qquad S \vDash P \oplus Q \quad \Rightarrow \quad S \vDash \top \oplus Q$$

So in a sense, we can *drop outcomes* by converting them to $\top$. For notational convenience, we define the following under-approximate triple:

$$\vDash^\downarrow \langle P \rangle \, C \, \langle Q \rangle \qquad \text{iff} \qquad \vDash \langle P \rangle \, C \, \langle Q \oplus \top \rangle$$

Using this shorthand, the following simpler specification is also valid:

$$\vDash^\downarrow \langle \mathsf{ok} : \mathsf{true} \rangle \; x := \mathsf{malloc}() \,\fatsemi\, [x] \leftarrow 1 \; \langle \mathsf{er} : x = \mathsf{null} \rangle \tag{5}$$

This example demonstrates that OL is suitable for reasoning about crash errors, just like IL. However our goal is not simply to cover the same use cases as IL, but rather to go further. Next, we will show in Section 2.2 that there are bugs expressible in OL that cannot be expressed in IL. In Section 2.3 we will also explain why the semantics of OL are a better fit for characterizing an important class of bugs known as manifest errors.

[1] The $\llbracket - \rrbracket^\dagger$ function is formally the monadic (or Kleisli) extension of $\llbracket - \rrbracket$; we will define this formally in Section 3.

2.2 A Broader Characterization of Correctness and Incorrectness

In the semantics of Incorrectness Logic, the notions of reachability and under-approximation are conflated: both are a consequence of the fact that IL quantifies over the states that satisfy the postcondition. However, reachability and under-approximation are separate concepts and OL allows us to reason about each independently. Reachability is expressed with the outcome conjunction $\oplus$ and under-approximation is achieved by dropping outcomes. Separating reachability and under-approximation is useful for *both* correctness *and* incorrectness reasoning.

To see this, we will first investigate *correctness* properties that rely on reachability. Before the introduction of Incorrectness Logic by O'Hearn [2019], de Vries and Koutavas [2011] devised a semantically equivalent logic, which they called Reverse Hoare Logic. The goal of this work was to prove *correctness* specifications that involved multiple possible end states, all of which must be reachable. As we saw in Example 1, Hoare Logic cannot express such specifications. So, de Vries and Koutavas [2011] proposed the Reverse Hoare Triple, which—like Incorrectness Triples—guarantees that every state described by the postcondition is reachable.

The motivating example for Reverse Hoare Logic was a nondeterministic shuffle function. Consider the following specification, where $\Pi(a)$ is the set of permutations of a:

$$[\mathsf{true}]\ b := \mathsf{shuffle}(a)\ [b \in \Pi(a)]$$

This specification states that *every* permutation of the list is a possible output of shuffle; however, it is not a complete correctness specification. It does not rule out the possibility that the output is not a permutation of the input ($b \notin \Pi(a)$). The semantics of Reverse Hoare Logic is motivated by *reachability*, but—like Incorrectness Logic—it achieves reachability in a manner that is inextricably linked to under-approximation, which is undesirable for correctness reasoning.

de Vries and Koutavas [2011] note this, stating that a complete specification for shuffle would require both Hoare Logic *and* Reverse Hoare Logic, but also that it would be worthwhile to study logics that can "express both the reachability of good states and the non-reachability of bad states" [de Vries and Koutavas 2011, §8]. OL does just that—the full correctness of the shuffle program can be captured using a single OL triple that guarantees reachability *without* under-approximating:

$$\langle \mathsf{true} \rangle\ b := \mathsf{shuffle}(a)\ \langle \bigoplus_{\pi \in \Pi(a)} (b = \pi) \rangle \tag{6}$$

The OL specification above states *not only* that all the permutations are reachable, *but also* that they are the only possible outcomes. So, OL allows us to express a correctness property in a single triple that otherwise would have required *both* a Hoare Triple *and* a Reverse Hoare Triple.

We now turn to consider incorrectness reasoning. Given that the above OL triple is a complete correctness specification, we are interested to know what it would mean for shuffle to be incorrect. In other words, what would it take to *disprove* the specification of shuffle? There are two ways that the triple could be false: either one particular permutation $\pi \in \Pi(a)$ is not reachable or the output b is (sometimes) not a permutation of a. Both bugs can be expressed as OL triples:

$$\exists \pi \in \Pi(a).\ \langle \mathsf{true} \rangle\ b := \mathsf{shuffle}(a)\ \langle b \neq \pi \rangle \qquad\qquad \langle \mathsf{true} \rangle\ b := \mathsf{shuffle}(a)\ \langle (b \notin \Pi(a)) \oplus \top \rangle$$

These triples both denote *true bugs* since the validity of either triple implies that specification (6) is false. In fact, these are the *only* ways that specification (6) can be false. This follows from a more general result called Falsification, which we prove in Theorem 5.6:

$$\nvDash \langle P \rangle\ C\ \langle \bigoplus_{i=1}^{n} Q_i \rangle \quad \text{iff} \quad \exists P' \Rightarrow P.\quad \exists i.\ \vDash \langle P' \rangle\ C\ \langle \neg Q_i \rangle \quad \text{or} \quad \vDash \langle P' \rangle\ C\ \langle (\bigwedge_{i=1}^{n} \neg Q_i) \oplus \top \rangle$$

Intuitively, a nondeterministic program is incorrect iff either one of the desired outcomes never occurs or some undesirable outcome sometimes occurs.[2] Incorrectness Logic can only characterize the latter type of incorrectness, whereas OL accounts for both and is thus strictly more expressive in the nondeterministic setting. An analogous result holds for probabilistic programs (Section 5.2), whereas IL is not suitable for reasoning about probabilistic incorrectness at all (Section 7.2).

2.3 Semantic Characterizations of Bugs

In addition to enabling us to witness a larger class of incorrectness than IL (unreachable states and probabilistic incorrectness), OL also provides a more *intuitive* way to reason about the type of bugs that IL was designed for: reachability of unsafe states.

Recalling the crash error in Section 2.1, both IL triples and OL triples soundly characterize the bug, as they both witness a trace that reaches the crash. The Incorrectness Triple (3) states that any failing execution where x is null is reachable from some starting state. In other words, true is a *necessary* condition to reach a segmentation fault. However, true is trivially a necessary condition, so this triple does not tell us much about what will trigger the bug in practice. By contrast, the OL triple (5) states that true is a *sufficient* condition, which gives us more information—the bug can *always* occur no matter what the starting state is.

The latter semantics has a close correspondence to a class of bugs, known as *manifest errors* [Le et al. 2022], which occur regardless of how the enclosing procedure is used and are of particular interest in automated bug-finding tools. Le et al. [2022, Def. 3.2] give a formal characterization of manifest errors, but it is not a natural fit for Incorrectness Logic: determining whether an IL triple is a manifest error requires an auxiliary algorithmic check. Though Le et al. [2022] note that there are connections between manifest errors and under-approximate variants of Hoare Logic, we go further in proving that their original definition of a manifest error is semantically equivalent to an OL triple of the form $\vDash^{\downarrow} \langle \mathsf{ok} : \mathsf{true} \rangle \; C \; \langle \mathsf{er} : q * \mathsf{true} \rangle$ (Lemma 6.7). Manifest errors are therefore trivial to characterize in OL by a simple syntactic inspection. This suggests that OL is semantically closer to the way in which programmers naturally characterize bugs.

In addition to being an intuitive foundation for incorrectness, OL unifies program analysis across two dimensions. First, it unifies correctness and incorrectness reasoning within a single program logic, and second, it does so across execution models (*e.g.*, nondeterministic and probabilistic). In the remainder of the paper, we will formalize the ideas that have been exemplified thus far. We formalize the OL model in Section 3 and Section 4, prove the applicability of OL to nondeterministic and probabilistic correctness and incorrectness in Section 5, and show how OL can be used in nondeterministic and probabilistic domains in Section 6 and Section 7, respectively. Given these advantages, we argue that OL offers a promising alternative foundation for incorrectness reasoning.

3 A MODULAR PROGRAMMING LANGUAGE

We start by defining a programming language, inspired by Dijkstra's guarded command language [Dijkstra 1975], see Figure 2. The syntax includes $\mathbb{0}$, which represents divergence, $\mathbb{1}$, acting as skip, sequential composition $C_1 \fatsemi C_2$, choice $C_1 + C_2$, iteration $C^\star$, and parametrizable atomic commands c. At first sight this looks like a standard imperative language (with nondeterministic choice). However, we will interpret the syntax in a semantic model that is parametric on a monad and a partial commutative monoid. The former enables a generic semantics of sequential composition, whereas the latter provides a generic interpretation of choice.

[2] In general, there is also a third option: the program diverges (has no outcomes). See Theorem 5.6.

$$\llbracket C \rrbracket : \Sigma \rightharpoonup M\Sigma$$

$$\begin{array}{lr}
C \ ::= \mathbb{0} & \llbracket \mathbb{0} \rrbracket(\sigma) = \varnothing \\
\quad \mid \mathbb{1} & \llbracket \mathbb{1} \rrbracket(\sigma) = \mathsf{unit}(\sigma) \\
\quad \mid C_1 \fatsemi C_2 & \llbracket C_1 \fatsemi C_2 \rrbracket(\sigma) = \mathsf{bind}(\llbracket C_1 \rrbracket(\sigma), \llbracket C_2 \rrbracket) \\
\quad \mid C_1 + C_2 & \llbracket C_1 + C_2 \rrbracket(\sigma) = \llbracket C_1 \rrbracket(\sigma) \diamond \llbracket C_2 \rrbracket(\sigma) \\
\quad \mid C^\star & \llbracket C^\star \rrbracket(\sigma) = \mathsf{lfp}(\lambda f.\lambda\sigma.f^\dagger(\llbracket C \rrbracket(\sigma))) \diamond \mathsf{unit}(\sigma))(\sigma) \\
\quad \mid c & \llbracket c \rrbracket(\sigma) = \llbracket c \rrbracket_{\mathsf{atom}}(\sigma)
\end{array}$$

Fig. 2. Syntax and Semantics of the Command Language parameterized by an execution model $\langle M, \mathsf{bind}, \mathsf{unit}, \diamond, \varnothing \rangle$ and a language of atomic commands with semantics $\llbracket c \rrbracket_{\mathsf{atom}} : \Sigma \to M\Sigma$

Before we define the semantic model we need to recall the definition of a monad and partial commutative monoid. We assume familiarity with basic category theory (categories, functors, natural transformations), see Pierce [1991] for an introduction.

Definition 3.1 (Monad). A monad is a triple $\langle M, \mathsf{bind}, \mathsf{unit} \rangle$ in which M is a functor on a category $\mathcal{C}$, $\mathsf{unit} \colon \mathsf{Id} \Rightarrow M$ is a natural transformation, and $\mathsf{bind} \colon MA \times (A \to MB) \to MB$ satisfies:

(1) $\mathsf{bind}(m, \mathsf{unit}) = m$
(2) $\mathsf{bind}(\mathsf{unit}(x), f) = f(x)$
(3) $\mathsf{bind}(\mathsf{bind}(m, f), g) = \mathsf{bind}(m, \lambda x.\mathsf{bind}(f(x), g))$

Typical examples of monads include powerset, error, and distribution monads (defined in Section 5 and Section 6). Given a function $f \colon A \to MB$, its monadic extension $f^\dagger \colon MA \to MB$ is defined as $f^\dagger(m) = \mathsf{bind}(m, f)$.

Definition 3.2 (PCM). A partial commutative monoid (PCM) is a triple $\langle X, \diamond, \varnothing \rangle$ consisting of a set X and a partial binary operation $\diamond \colon X \to X \rightharpoonup X$ that is associative, commutative, and has unit $\varnothing$.

A typical example of a PCM, used in probabilistic reasoning, is $\langle [0, 1], +, 0 \rangle$ (+ is partial, it is undefined when the addition is out-of-bounds). We are now ready to define the execution model we need to provide semantics to our language.

Definition 3.3 (Execution Model). An Execution Model is a structure $\langle M, \mathsf{bind}, \mathsf{unit}, \diamond, \varnothing \rangle$ such that $\langle M, \mathsf{bind}, \mathsf{unit} \rangle$ is a monad in the category of sets, and for any set A, $\langle MA, \diamond, \varnothing \rangle$ is a PCM that preserves the monad bind: $\mathsf{bind}(m_1 \diamond m_2, k) = \mathsf{bind}(m_1, k) \diamond \mathsf{bind}(m_2, k)$ and $\mathsf{bind}(\varnothing, k) = \varnothing$.

In Figure 2 we present the semantics of the language. The monad operations are used to provide semantics to $\mathbb{1}$ and sequential composition $\fatsemi$ whereas the monoid operation is used in the semantics of choice and iteration. Note that in general the semantics of the language is partial since $\diamond$ is partial, which is necessary in order to express a probabilistic semantics, since two probability distributions can only be combined if their cumulative probability mass is at most 1. For the languages we will work with in this paper, there are simple syntactic checks to ensure totality of the semantics. In the probabilistic case, this involves ensuring that all uses of + and $\star$ are guarded. We show that the semantics is total for the execution models of interest in Zilberstein et al. [2023, §A].

Example 3.4 (State and Guarded Commands). The base language introduced in the previous section is parametric over a set of program states Σ. In this example, we describe a specific type of program state, the semantics of commands over those states, and a mechanism to define the typical control flow operators (if and while). First, we assume some syntax of program expressions $e \in \mathsf{Exp}$

which includes variables $x \in \mathsf{Var}$ as well as the typical Boolean and arithmetic operators. Atomic commands come from the following syntax.

$$c ::= \mathsf{assume}\ e \mid x := e \qquad (x \in \mathsf{Var}, e \in \mathsf{Exp})$$

The command $\mathsf{assume}\ e$ does nothing if e is true and eliminates the current outcome if not; $x := e$ is variable assignment. A program stack is a mapping from variables to values $\mathcal{S} = \{s : \mathsf{Var} \to \mathsf{Val}\}$ where program values $\mathsf{Val} = \mathbb{Z} \cup \mathbb{B}$ are integers ($\mathbb{Z}$) or Booleans ($\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$). Expressions are evaluated to values given a stack using $[\![e]\!]_{\mathsf{Exp}} : \mathcal{S} \to \mathsf{Val}$. The semantics of atomic commands $[\![c]\!] : \mathcal{S} \to M\mathcal{S}$, parametric on an execution model, is defined below.

$$[\![\mathsf{assume}\ e]\!](s) = \begin{cases} \mathsf{unit}(s) & \text{if } [\![e]\!]_{\mathsf{Exp}}(s) = \mathsf{true} \\ \varnothing & \text{if } [\![e]\!]_{\mathsf{Exp}}(s) = \mathsf{false} \end{cases} \qquad [\![x := e]\!](s) = \mathsf{unit}(s[x \mapsto [\![e]\!]_{\mathsf{Exp}}(s)])$$

While a language instantiated with the atomic commands described above is still nondeterministic, we can use assume to define the usual (deterministic) control flow operators as syntactic sugar.

$$\begin{aligned} \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 &= (\mathsf{assume}\ e \fatsemi C_1) + (\mathsf{assume}\ \neg e \fatsemi C_2) & \mathsf{skip} &= \mathbb{1} & C^0 &= \mathbb{1} \\ \mathsf{while}\ e\ \mathsf{do}\ C &= (\mathsf{assume}\ e \fatsemi C)^\star \fatsemi \mathsf{assume}\ \neg e & \mathsf{for}\ N\ \mathsf{do}\ C &= C^N & C^{k+1} &= C \fatsemi C^k \end{aligned}$$

In fact, when paired with a nondeterministic evaluation model, this language is equivalent to Dijkstra's [1975] Guarded Command Language (GCL) by a straightforward syntactic translation.

4 OUTCOME LOGIC

In this section, we formally define Outcome Logic (OL). We first define the logic of outcome assertions which will act as the basis for writing pre- and postconditions in OL. Next, we give the semantics of OL triples, which is parametric on an execution model, atomic command semantics, and an assertion logic. Finally, we give proof rules that are sound for all OL instances.

4.1 A Logic for Monoidal Assertions: Modeling the Outcome Conjunction

We now give a formal account of the outcome assertion logic that was briefly described in Section 2.1. The outcome assertion logic is an instance of the Logic of Bunched Implications (BI) [O'Hearn and Pym 1999], a substructural logic that is used to reason about resource usage. Separation logic [Reynolds 2002] and its extensions [O'Hearn 2004] are the most well-known applications of BI. In our case, the relevant resources are *program outcomes* rather than heap locations.

We use the formulation of BI due to Docherty [2019]. While Docherty [2019] gives a thorough account of the BI proof theory, we are mainly interested in the semantics for the purposes of this paper. The syntax and semantics are given in Figure 3 with logical negation $\neg\varphi$ being defined as $\varphi \Rightarrow \bot$. The semantics is parametric on a BI frame $\langle X, \diamond, \preccurlyeq, \varnothing\rangle$—where $\langle X, \diamond, \varnothing\rangle$ is a PCM and $\preccurlyeq\ \subseteq X \times X$ is a preorder—and a satisfaction relation for basic assertions $\vDash_{\mathsf{atom}} \subseteq X \times \mathsf{Prop}$.

The two non-standard additions are the *outcome conjunction* $\oplus$, a connective to join outcomes, and $\top^\oplus$, an assertion to specify that there are no outcomes. These intended meanings are reflected in the semantics: $\top^\oplus$ is only satisfied by the monoid unit $\varnothing$, whereas $\varphi \oplus \psi$ is satisfied by m iff m can be partitioned into $m_1 \diamond m_2$ to satisfy each outcome formula separately. We will focus on *classical* interpretations of BI where the preorder $\preccurlyeq$ is equality.[3]

Definition 4.1 (Outcome Assertion Logic). Given an execution model $\langle M, \mathsf{bind}, \mathsf{unit}, \diamond, \varnothing\rangle$ and a satisfaction relation for atomic assertions $\vDash_{\mathsf{atom}} \subseteq M\Sigma \times \mathsf{Prop}$, an Outcome Assertion Logic is an

[3] Intuitionistic interpretations of BI with non-trivial preorders can be used as an alternative way to encode under-approximate program logics. This idea is explored in Zilberstein et al. [2023, §B.1].

$$
\begin{array}{rll@{\qquad}lll}
\varphi ::= & \top & & m \vDash \top & \text{always} & \\
| & \bot & & m \vDash \bot & \text{never} & \\
| & \top^{\oplus} & & m \vDash \top^{\oplus} & \text{iff} & m = \varnothing \\
| & \varphi \wedge \psi & & m \vDash \varphi \wedge \psi & \text{iff} & m \vDash \varphi \text{ and } m \vDash \psi \\
| & \varphi \oplus \psi & & m \vDash \varphi \oplus \psi & \text{iff} & \exists m_1, m_2.\ m_1 \diamond m_2 \preccurlyeq m \text{ and } m_1 \vDash \varphi \text{ and } m_2 \vDash \psi \\
| & \varphi \Rightarrow \psi & & m \vDash \varphi \Rightarrow \psi & \text{iff} & \forall m'.\ \text{if } m \preccurlyeq m' \text{ and } m' \vDash \varphi \text{ then } m' \vDash \psi \\
| & P & & m \vDash P & \text{iff} & P \in \mathsf{Prop} \text{ and } m \vDash_{\mathsf{atom}} P
\end{array}
$$

Fig. 3. Syntax and semantics of BI given a BI frame $\langle X, \diamond, \preccurlyeq, \varnothing \rangle$ and satisfaction relation $\vDash_{\mathsf{atom}} \subseteq X \times \mathsf{Prop}$

instance of BI based on the BI frame $\langle M\Sigma, \diamond, =, \varnothing \rangle$. Informally, we refer to BI assertions φ, ψ as outcome assertions and the atomic assertions $P, Q \in \mathsf{Prop}$ as individual outcomes.

Remark 1 (Notation for Assertions). For the remainder of the paper, lowercase Greek metavariables φ, ψ refer to (syntactic) outcome assertions (Definition 4.1), uppercase Latin metavariables P, Q refer to atomic assertions (individual outcomes), and lowercase Latin metavariables p, q refer to assertions on individual program states.

Example 4.2 (Outcomes). We mentioned one example of a PCM in Section 2: X can be sets of program states and the monoid operation $\diamond$ is set union. Another example is probability (sub)distributions over a set and $\diamond$ is $+$. This monoid operation is partial; adding two subdistributions is only possible if the mass associated with a point (and the entire distribution) remains in $[0, 1]$.

As discussed in Section 2, under-approximation and the ability to drop outcomes is an important part of incorrectness reasoning as it allows large scale analyses to only track pertinent information. We use the following shorthand to express under-approximate outcome assertions.

Definition 4.3 (Under-Approximate Outcome Assertions). Given an outcome assertion logic with satisfaction relation $\vDash \subseteq M\Sigma \times \mathsf{Prop}$, we define an under-approximate version $\vDash^{\downarrow} \subseteq M\Sigma \times \mathsf{Prop}$ as $m \vDash^{\downarrow} \varphi$ iff $m \vDash \varphi \oplus \top$.

Intuitively, $\varphi \oplus \top$ corresponds to under-approximation since it states that φ only covers a subset of the outcomes (with the rest being unconstrained, since they are covered by $\top$). Defining under-approximation in this way allows us to reason about correctness and incorrectness within a single program logic. It also enables us to drop outcomes simply by weakening; it is always possible to weaken an outcome to $\top$, so $m \vDash P \oplus Q$ implies that $m \vDash P \oplus \top$. Equivalently, $m \vDash^{\downarrow} P \oplus Q$ implies that $m \vDash^{\downarrow} P$. These facts are proven in Zilberstein et al. [2023, §B]. A similar formulation would be possible using an intuitionistic interpretation of BI (where, roughly speaking, we take the preorder to be $m_1 \preccurlyeq m_2$ iff $\exists m.\ m_1 \diamond m = m_2$). We prove this correspondence in Zilberstein et al. [2023, §B.1].

4.2 Outcome Triples

We now have all the ingredients needed to define the validity of the program logic.

Definition 4.4 (Outcome Triples). The parameters needed to instantiate OL are:

(1) An execution model: $\langle M, \mathsf{bind}, \mathsf{unit}, \diamond, \varnothing \rangle$
(2) A set of program states Σ and semantics of atomic commands: $[\![c]\!]_{\mathsf{atom}} : \Sigma \to M\Sigma$
(3) A syntax of atomic assertions Prop and satisfaction relation: $\vDash_{\mathsf{atom}} \subseteq M\Sigma \times \mathsf{Prop}$

Now, let $[\![-]\!] : \Sigma \to M\Sigma$ be the semantics of the language in Figure 2 with parameters (1) and (2) and $\vDash$ be the outcome assertion satisfaction relation (Definition 4.1) with parameters (1) and (3). For any program C (Figure 2), and outcome assertions φ and ψ:

$$\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle \qquad \text{iff} \qquad \forall m \in M\Sigma.\quad m \vDash \varphi \quad \implies \quad [\![C]\!]^{\dagger}(m) \vDash \psi$$

OL is a generalization of Hoare Logic—the triples first quantify over elements satisfying the precondition and then stipulate that the result of running the program on those elements satisfies the postcondition. The difference is that now the pre- and postconditions are expressed as outcome assertions and thus satisfied by a monoidal collection $m \in M\Sigma$, which can account for execution models based on nondeterminism and probability distributions.

Using outcome assertions for pre- and postconditions adds significant expressive power. We already saw in Section 2 how Outcome Logic allows us to reason about reachability and under-approximation. We can also encode other useful concepts such as partial correctness—the postcondition holds *if* the program terminates—by taking a disjunction with $\top^{\oplus}$ to express that the program may diverge[4]. For convenience, we define the following notation where the left triple encodes under-approximation and the right triple encodes partial correctness.

$$\vDash^{\downarrow} \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \text{iff} \quad \vDash \langle \varphi \rangle\ C\ \langle \psi \oplus \top \rangle \qquad\qquad \vDash_{\text{pc}} \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \text{iff} \quad \vDash \langle \varphi \rangle\ C\ \langle \psi \vee \top^{\oplus} \rangle$$

In fact, the right triple corresponds exactly to standard Hoare Logic (Figure 1) if we instantiate OL using the powerset semantics (Definition 5.3) and limit the pre- and postconditions to be atomic assertions. This result is stated below and proven in Zilberstein et al. [2023, §C].

Theorem 4.5 (Subsumption of Hoare Triples). $\vDash \{P\}\ C\ \{Q\}$ *iff* $\vDash_{\text{pc}} \langle P \rangle\ C\ \langle Q \rangle$

While capturing many logics in one framework is interesting and demonstrates the versatility of Outcome Triples, our primary goal is to investigate the roles that these program logics can play for expressing correctness and incorrectness properties. We justify OL as a theoretical basis for correctness and incorrectness reasoning in Section 5 and give examples for how OL can be applied to nondeterministic and probabilistic programs in Section 6 and Section 7.

4.3 Proof Systems

Now that we have formalized the *validity* of Outcome triples (denoted $\vDash \langle \varphi \rangle\ C\ \langle \psi \rangle$), we can construct proof systems for this family of logics. We write $\vdash \langle \varphi \rangle\ C\ \langle \psi \rangle$ to mean that the triple $\langle \varphi \rangle\ C\ \langle \psi \rangle$ is *derivable* from a set of inference rules. Each set of inference rules that we define throughout the paper will be *sound* with respect to a certain OL instance.

Global rules. Some generic rules that are valid for any OL instance are shown at the top of Figure 4. Most of the rules including Zero, One, and Seq are standard. The Rule of Consequence allows the strengthening and weakening of pre- and postconditions respectively using any semantically valid BI implication. The Split rule allows us to analyze the program C with two different pre/postcondition pairs and join the results using an outcome conjunction.

Rules for nondeterministic programs. In the middle of Figure 4 we see two rules that are only valid in nondeterministic languages where the semantics is based on the powerset monad. The Plus rule characterizes nondeterministic choice by joining the outcomes from analyzing each branch using an outcome conjunction. Repeated uses of the Induction rule allow us to unroll an iterated command for a finite number of iterations.

Rules for guarded programs. Finally, at the bottom of Figure 4 is a collection of rules for expression-based languages that have the syntax introduced in Example 3.4. We write $P \vDash e$ to mean that P *entails* e. Formally, if $P \vDash e$ and $Q \vDash \neg e$ and $m \vDash P \oplus Q$, then $[\![\text{assume } e]\!]^{\dagger}(m) \vDash P$. Substitutions $P[e/x]$ must be defined for basic assertions and satisfy $m \vDash P[e/x]$ implies $[\![x := e]\!]^{\dagger}(m) \vDash P$.

[4] Disjunctions are defined $\varphi \vee \psi$ iff $\neg(\neg\varphi \wedge \neg\psi)$, a standard encoding in classical logic.

Generic Rules

$$\frac{}{\langle \varphi \rangle\ \mathbb{0}\ \langle \top^{\oplus} \rangle}\ \textsc{Zero} \qquad \frac{}{\langle \varphi \rangle\ \mathbb{1}\ \langle \varphi \rangle}\ \textsc{One} \qquad \frac{\langle \varphi \rangle\ C_1\ \langle \psi \rangle \qquad \langle \psi \rangle\ C_2\ \langle \vartheta \rangle}{\langle \varphi \rangle\ C_1 \fatsemi C_2\ \langle \vartheta \rangle}\ \textsc{Seq} \qquad \frac{\forall i \in \mathbb{N}.\ \langle \varphi_i \rangle\ C\ \langle \varphi_{i+1} \rangle}{\langle \varphi_0 \rangle\ \mathsf{for}\ N\ \mathsf{do}\ C\ \langle \varphi_N \rangle}\ \textsc{For}$$

$$\frac{\langle \varphi_1 \rangle\ C\ \langle \psi_1 \rangle \quad \langle \varphi_2 \rangle\ C\ \langle \psi_2 \rangle}{\langle \varphi_1 \oplus \varphi_2 \rangle\ C\ \langle \psi_1 \oplus \psi_2 \rangle}\ \textsc{Split} \qquad \frac{\varphi' \Rightarrow \varphi \quad \langle \varphi \rangle\ C\ \langle \psi \rangle \quad \psi \Rightarrow \psi'}{\langle \varphi' \rangle\ C\ \langle \psi' \rangle}\ \textsc{Consequence}$$

$$\frac{}{\langle \top^{\oplus} \rangle\ C\ \langle \top^{\oplus} \rangle}\ \textsc{Empty} \qquad \frac{}{\langle \varphi \rangle\ C\ \langle \top \rangle}\ \textsc{True} \qquad \frac{}{\langle \bot \rangle\ C\ \langle \varphi \rangle}\ \textsc{False}$$

Nondeterministic Rules

$$\frac{\langle \varphi \rangle\ C_1\ \langle \psi_1 \rangle \qquad \langle \varphi \rangle\ C_2\ \langle \psi_2 \rangle}{\langle \varphi \rangle\ C_1 + C_2\ \langle \psi_1 \oplus \psi_2 \rangle}\ \textsc{Plus} \qquad \frac{\langle \varphi \rangle\ \mathbb{1} + C \fatsemi C^{\star}\ \langle \psi \rangle}{\langle \varphi \rangle\ C^{\star}\ \langle \psi \rangle}\ \textsc{Induction}$$

Expression-Based Rules

$$\frac{}{\langle P[e/x] \rangle\ x := e\ \langle P \rangle}\ \textsc{Assign} \qquad \frac{P_1 \vDash e \qquad P_2 \vDash \neg e}{\langle P_1 \oplus P_2 \rangle\ \mathsf{assume}\ e\ \langle P_1 \rangle}\ \textsc{Assume}$$

$$\frac{P_1 \vDash e \qquad \langle P_1 \rangle\ C_1\ \langle Q_1 \rangle \qquad P_2 \vDash \neg e \qquad \langle P_2 \rangle\ C_2\ \langle Q_2 \rangle}{\langle P_1 \oplus P_2 \rangle\ \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \langle Q_1 \oplus Q_2 \rangle}\ \textsc{If (Multi-Outcome)}$$

Fig. 4. Inference rules that are valid for a variety of OL instantiations. The metavariables φ, ψ refer to arbitrary outcome assertions and P, Q refer to atomic (single-outcome) assertions.

The Assign rule uses *weakest-precondition* style backwards substitution. Assume uses expression entailment to annihilate the outcome where the guard is false. Similarly, If (Multi-Outcome) uses entailment to map entire outcomes to the true or false branches of an if statement, respectively.

All the rules in Figure 4 are sound (see Zilberstein et al. [2023, §F] for details of the proof).

Theorem 4.6 (Soundness of Proof System). *If* $\vdash \langle P \rangle\ C\ \langle Q \rangle$ *then* $\vDash \langle P \rangle\ C\ \langle Q \rangle$

Note that it is not possible to have generic loop-invariant based iteration rules that are valid for *all* instances of Outcome Logic. This is because loop invariants assume a *partial* correctness specification; they do not guarantee termination. Outcome Logic—in some instantiations—guarantees reachability of end states and therefore must witness a *terminating* program execution. This is in line with the Backwards Variant rule from Incorrectness Logic [O'Hearn 2019, Fig.2], the While rule from Reverse Hoare Logic [de Vries and Koutavas 2011, Fig.2], and Loop Variants from Total Hoare Logic [Apt 1981]. Such a rule for GCL is available in Zilberstein et al. [2023, §G].

5 MODELING CORRECTNESS AND INCORRECTNESS VIA OUTCOMES

Incorrectness Logic was motivated in large part by its ability to *disprove* correctness specifications (*i.e.,* Hoare Triples) [Möller et al. 2021, Thm 4.1]. In this section, we prove that OL can not only disprove Hoare Triples (Corollary 5.7), but it can also express strictly *more* types of incorrectness than IL can. Theorem 5.6 shows three classes of bugs in nondeterministic programs that can be characterized in OL, only one of which is expressible in IL. Section 5.2 shows that OL can express probabilistic incorrectness too, whereas IL cannot.

Our first result is stated in terms of *semantic* triples in which the pre- and postconditions are *semantic* assertions (which we denote with uppercase Greek metavariables $\Phi, \Psi \in 2^{M\Sigma}$) rather than the *syntactic* assertions $\varphi, \psi \in \mathsf{Prop}$ we have seen thus far. The advantage of this approach is that we can show the power of the OL model without worrying about the expressiveness of the syntactic assertion language (as a point of reference, the formal development of Incorrectness Logic is purely semantic [Le et al. 2022; Möller et al. 2021; O'Hearn 2019], as was the metatheory for separation logic [Calcagno et al. 2007; Yang 2001]).

The following Falsification theorem states that any false OL triple can be disproven within OL. Since we already know that OL subsumes Hoare Logic (Theorem 4.5), it follows that any correctness property that is expressible in Hoare Logic can be disproven using OL. We use $\vDash_S \langle \Phi \rangle\ C\ \langle \Psi \rangle$ to denote a valid semantic OL triple, that is: if $m \in \Phi$, then $[\![C]\!]^\dagger(m) \in \Psi$. The assertion $\mathsf{sat}(\Phi)$ means that Φ is satisfiable, in other words $\Phi \neq \emptyset$.

Theorem 5.1 (Semantic Falsification). *For any OL instance and any program C and semantic assertions Φ, Ψ:*

$$\nvDash_S \langle \Phi \rangle\ C\ \langle \Psi \rangle \qquad \textit{iff} \qquad \exists \Phi'. \quad \textit{such that} \quad \Phi' \Rightarrow \Phi, \quad \mathsf{sat}(\Phi'), \quad \textit{and} \quad \vDash_S \langle \Phi' \rangle\ C\ \langle \neg\Psi \rangle$$

Proof. We provide a proof sketch here. If $\nvDash_S \langle \Phi \rangle\ C\ \langle \Psi \rangle$, then there must be an $m \in \Phi$ such that $[\![C]\!]^\dagger(m) \notin \Psi$. Choosing $\Phi' = \{m\}$ gives us $\vDash_S \langle \Phi' \rangle\ C\ \langle \neg\Psi \rangle$. For the reverse direction, we know from $\mathsf{sat}(\Phi')$ that there is an $m \in \Phi'$ and from $\Phi' \Rightarrow \Phi$, we know that $m \in \Phi$ and from $\vDash_S \langle \Phi \rangle\ C\ \langle \neg\Psi \rangle$, we know that $[\![C]\!]^\dagger(m) \notin \Psi$, so $\nvDash_S \langle \Phi \rangle\ C\ \langle \Psi \rangle$. □

The full proof of this theorem and formulation of semantic triples are given in Zilberstein et al. [2023, §D.1]. While this result shows the power of the OL *model*, we also seek to answer whether the outcome assertion syntax given in Definition 4.1 can express the pre- and postconditions needed to disprove other triples. We answer this question in the affirmative, although the forward direction of the result has to be proven separately for nondeterministic and probabilistic models. While the semantic proof above applies to *any* OL instance, the syntactic versions that we present in Section 5.1 and Section 5.2 rely on additional properties of the specific OL instance. Despite the added complexity, we deem this worthwhile since syntactic descriptions give us a characterizations that can be used in the design of automated bug-finding tools.

The reverse direction of Theorem 5.1 corresponds to O'Hearn's [2019] Principle of Denial, though the original Principle of Denial used two triple types (IL and Hoare) and now we only need to use one (OL). We can prove a syntactic version of The Principle of Denial for OL, which can be thought of as a generalization of the *true positives* property, since it tells us when an OL triple (denoting a bug) disproves another OL triple (denoting correctness).

Theorem 5.2 (Principle of Denial). *For any OL instance and any program C and syntactic assertions φ, φ', and ψ:*

$$\textit{If} \quad \varphi' \Rightarrow \varphi, \quad \mathsf{sat}(\varphi'), \quad \textit{and} \quad \vDash \langle \varphi' \rangle\ C\ \langle \neg\psi \rangle \quad \textit{then} \quad \nvDash \langle \varphi \rangle\ C\ \langle \psi \rangle$$

This theorem is a consequence of Theorem 5.1, together with a result stating how to translate syntactic triples to equivalent semantic ones [Zilberstein et al. 2023, Lemma D.1].

Proving a syntactic version of the forward direction of Theorem 5.1 is more complicated—it requires us to witness the existence of a syntactic assertion corresponding to Φ'. The way in which this assertion is constructed depends on several properties of the OL instance. One additional requirement is that the program C must terminate after finitely many steps, otherwise the precondition may not be finitely expressible. This is a common issue when generating preconditions and as a result many developments choose to work with semantic assertions rather than syntactic ones [Kaminski 2019]. The IL falsification results are also only given semantically [Möller et al. 2021; O'Hearn 2019], which avoids infinitary assertions in loop cases.

In the following sections, we will investigate falsification in both nondeterministic and probabilistic OL instances. In doing so, we will provide more specific falsification theorems which both deal with syntactic assertions and more precisely characterize the ways in which particular programs can be incorrect. While we have just seen that we can obtain a falsification witness for correctness specifications $\langle\varphi\rangle\ C\ \langle\psi\rangle$ by negating the postcondition, proving a triple with postcondition $\neg\psi$ may not be convenient. For example, if ψ is a sequence of outcomes $Q_1 \oplus \cdots \oplus Q_n$, then it is not immediately clear what $\neg\psi$ expresses. We therefore provide more intuitive assertions for canonical types of incorrectness encountered in programs.

5.1 Falsification in Nondeterministic Programs

In this section, we explore falsification for nondeterministic programs. The first step is to formally define a nondeterministic instance of OL by defining an evaluation model and BI frame.

Definition 5.3 (Nondeterministic Evaluation Model). A nondeterministic evaluation model based on program states $\sigma \in \Sigma$ is $\langle 2^\Sigma, \mathsf{bind}, \mathsf{unit}, \cup, \emptyset\rangle$ where $\langle 2^{(-)}, \mathsf{bind}, \mathsf{unit}\rangle$ is the powerset monad:

$$\mathsf{bind}(S,k) \triangleq \bigcup_{x\in S} k(x) \qquad\qquad \mathsf{unit}(x) \triangleq \{x\}$$

Definition 5.4 (Nondeterministic Outcome Assertions). Given some satisfaction relation on program states $\vDash_\Sigma \subseteq \Sigma \times \mathsf{Prop}$, we create an instance of the outcome assertion logic (Definition 4.1) with the BI frame $\langle 2^\Sigma, \cup, =, \emptyset\rangle$ such that atomic assertions come from Prop and are satisfied as follows:

$$S \vDash P \qquad \text{iff} \qquad S \neq \emptyset \quad \text{and} \quad \forall\sigma \in S.\ \sigma \vDash_\Sigma P$$

We impose one additional requirement, that the atomic assertions $Q \in \mathsf{Prop}$ can be logically negated[5], which we will denote $\overline{Q}$. Now, we return to the question of how to falsify a sequence of nondeterministic outcomes $Q_1 \oplus \cdots \oplus Q_n$. Lemma 5.5 shows that there are exactly three ways that this assertion can be false.

Lemma 5.5 (Falsifying Assertions). *For any $S \in 2^\Sigma$ and atomic assertions $Q_1, \ldots, Q_n$,*

$$S \nvDash Q_1 \oplus \cdots \oplus Q_n \qquad \textit{iff} \qquad \exists i.\ S \vDash \overline{Q}_i \quad \textit{or} \quad S \vDash (\overline{Q}_1 \land \cdots \land \overline{Q}_n) \oplus \top \quad \textit{or} \quad S \vDash \top^\oplus$$

If we take $Q_1 \oplus \cdots \oplus Q_n$ to represent a desirable set of program outcomes, then Lemma 5.5 tells us that said program can be wrong in exactly three ways. Either there is some desirable outcome (Q_i) that the program never reaches, there is some undesirable outcome ($\overline{Q}_1 \land \cdots \land \overline{Q}_n$) that the program sometimes reaches, or there is an input that causes it to diverge ($\top^\oplus$). Now, following from this result, we can state what it means to falsify a nondeterministic specification:

Theorem 5.6 (Nondeterministic Falsification). *For any OL instance based on the nondeterministic evaluation model (Definition 5.3) and outcome assertions (Definition 5.4), $\nvDash \langle\varphi\rangle\ C\ \langle\bigoplus_{i=1}^n Q_i\rangle$ iff:*

$$\exists\varphi' \Rightarrow \varphi.\ \mathsf{sat}(\varphi') \ \ \textit{and}\ \ \exists i.\ \vDash \langle\varphi'\rangle\ C\ \langle\overline{Q}_i\rangle \ \textit{or}\ \vDash^{\downarrow} \langle\varphi'\rangle\ C\ \langle\bigwedge_{i=1}^n \overline{Q}_i\rangle \ \textit{or}\ \vDash \langle\varphi'\rangle\ C\ \langle\top^\oplus\rangle$$

The type of bugs expressible in Incorrectness Logic are a special case of Theorem 5.6. Since IL is under-approximate, it can only express the second kind of bug (reachability of a bad outcome), not the first (non-reachability of a good outcome), or last (divergence). IL was motivated by its ability to disprove Hoare Triples—since Hoare Triples are a special case of OL (Theorem 4.5), Theorem 5.6 suggests that OL can disprove Hoare Triples as well. We make this correspondence explicit in the following Corollary where, compared to Theorem 5.6, the first two cases collapse since there is only a single outcome and the divergence case no longer represents a bug since the Hoare Triple is a partial correctness specification.

[5] Crucially, $\overline{Q}$ is not the same as $\neg Q$ (where $\neg$ is from BI) since $S \vDash \neg Q$ iff $S = \emptyset$ or $\exists\sigma \in S.\sigma \nvDash_\Sigma Q$ whereas $S \vDash \overline{Q}$ iff $S \neq \emptyset$ and $\forall\sigma \in S.\ \sigma \nvDash_\Sigma Q$.

Corollary 5.7 (Hoare Logic Falsification).

$$\nvDash \{P\}\ C\ \{Q\} \qquad \textit{iff} \qquad \exists\varphi \Rightarrow P.\quad \mathsf{sat}(\varphi) \quad \textit{and} \quad \vDash^{\downarrow} \langle\varphi\rangle\ C\ \langle\overline{Q}\rangle$$

So, although we do not show that OL *semantically* subsumes Incorrectness Logic, it does have the ability to express the same bugs as IL. OL can also disprove more complex correctness properties, such as that of the shuffle function that we saw in Section 2.2. As we will now see, another OL instance is capable of disproving probabilistic properties too.

5.2 Falsification in Probabilistic Programs

Before we can define falsification in a probabilistic setting, we must establish some preliminary definitions. Probabilistic programs use an execution model based on probability (sub)distributions. A (sub)distribution $\mu \in \mathcal{D}X$ over a set X is a function mapping elements $x \in X$ to probabilities in $[0,1] \subset \mathbb{R}$. The support of a distribution is the set of elements having nonzero probability $\mathsf{supp}(\mu) = \{x \mid \mu(x) > 0\}$ and the mass of a distribution is $|\mu| = \sum_{x\in\mathsf{supp}(\mu)} \mu(x)$. A valid distribution must have mass at most 1. The empty distribution $\varnothing$ maps everything to probability 0 and distributions can be summed pointwise $\mu_1 + \mu_2 = \lambda x.\mu_1(x) + \mu_2(x)$ if $|\mu_1| + |\mu_2| \le 1$. For any countable set X, $\langle \mathcal{D}X, +, \varnothing\rangle$ is a PCM. In addition, distributions can be weighted by scalars $p \cdot \mu = \lambda x.p \cdot \mu(x)$ if $p \cdot |\mu| \le 1$ (this is always defined if $p \le 1$). The Dirac distribution δ_x assigns probability 1 to x and 0 to everything else. We complete the definition of a probabilistic execution model:

Definition 5.8 (Probabilistic Evaluation Model). A probabilistic evaluation model based on program states Σ is defined as $\langle \mathcal{D}\Sigma, \mathsf{bind}, \mathsf{unit}, +, \varnothing\rangle$ where $\langle \mathcal{D}, \mathsf{bind}, \mathsf{unit}\rangle$ is the Giry [1982] monad:

$$\mathsf{bind}(\mu, k) = \sum_{x\in\mathsf{supp}(\mu)} \mu(x) \cdot k(x) \qquad\qquad \mathsf{unit}(x) = \delta_x$$

We can make our imperative language probabilistic by adding a command $x \xleftarrow{\$} \eta$ for sampling from finitely supported probability distributions $\eta \in \mathcal{D}\mathsf{Val}$ over program values. This command is intended to be added to an existing language such as GCL (Example 3.4) or mGCL (Section 6.4). The program semantics and atomic assertions are based on distributions over program states $\mu \in \mathcal{D}\Sigma$. The semantics for the sampling command is defined in terms of variable assignment. This allows us to abstract over the type of program states.

$$c ::= x \xleftarrow{\$} \eta \qquad\qquad [\![x \xleftarrow{\$} \eta]\!]\,(\sigma) = \mathsf{bind}(\eta, \lambda v.\,[\![x := v]\!]\,(\sigma))$$

Definition 5.9 (Probabilistic Outcome Assertions). Given some satisfaction relation on program states $\vDash_\Sigma\ \subseteq \Sigma \times \mathsf{Prop}$, we instantiate the outcome assertion logic (Definition 4.1) with the BI frame $\langle \mathcal{D}\Sigma, +, =, \varnothing\rangle$ such that atomic assertions have the form $\mathbb{P}[A] = p$ where $p \in [0,1]$, $A \in \mathsf{Prop}$, and:

$$\mu \vDash \mathbb{P}[A] = p \qquad \text{iff} \qquad |\mu| = p \quad \text{and} \quad \forall\sigma \in \mathsf{supp}(\mu).\ \sigma \vDash_\Sigma A$$

Intuitively, the assertion $\mathbb{P}[A] = p$ states that the outcome A occurs with probability p. As a shorthand for under-approximate assertions, we also define $\mathbb{P}[A] \ge p$ to be $(\mathbb{P}[A] = p) \oplus \top$ (see Lemma B.5 for a semantic justification).

We will now investigate falsification of probabilistic assertions of the form $\bigoplus_{i=1}^{n}(\mathbb{P}[A_i] = p_i)$. In general, any such sequence can be falsified by specifying the precise probabilities of all combinations of the outcomes A_i. In the special case where $n = 2$, $\mu \nvDash (\mathbb{P}[A] = p \oplus \mathbb{P}[B] = q)$ iff:

$$\mu \quad \vDash \quad \mathbb{P}[A \wedge B] = p_1 \quad \oplus \quad \mathbb{P}[A \wedge \neg B] = p_2 \quad \oplus \quad \mathbb{P}[\neg A \wedge B] = p_3 \quad \oplus \quad \mathbb{P}[\neg A \wedge \neg B] = p_4$$

Such that $p_4 > 0$ or $p_2 > p$ or $p_3 > q$ or $p_1 + p_2 + p_3 \ne p + q$. The more general version of this result shows that 2^n outcomes are needed to disprove an assertion with n outcomes [Zilberstein et al. 2023, Lemmas D.9 and D.12], which is infeasible for large n. However, there are several special cases that require many fewer outcomes. For example, if all the A_is are pairwise disjoint, then falsification can be achieved with just $n + 1$ outcomes. Below, we use $\vec{q}$ to denote a vector of probabilities $q_1, \ldots, q_n$.

Theorem 5.10 (Disjoint Falsification). *First, let* $A_0 = \bigwedge_{i=1}^{n} \neg A_i$*. If all the events are disjoint (for all* $i \neq j$*,* $A_i \wedge A_j$ *iff* false*), then:*

$$\nvDash \langle \varphi \rangle\ C\ \langle \bigoplus_{i=1}^{n} (\mathbb{P}[A_i] = p_i) \rangle \qquad \textit{iff} \qquad \exists \vec{q}, \varphi' \Rightarrow \varphi. \quad \vDash \langle \varphi' \rangle\ C\ \langle \bigoplus_{i=0}^{n} (\mathbb{P}[A_i] = q_i) \rangle$$

Such that $\mathsf{sat}(\varphi')$ *and* $q_0 \neq 0$ *or for some* i $q_i \neq p_i$*.*

Many specifications fall into this disjointness case since the primary way in which proofs split into multiple probabilistic outcomes is via sampling, which always splits the postcondition into disjoint outcomes with the sampled variable x taking on a unique value.

The correctness of some probabilistic programs is specified using lower bounds. For example, we may want to specify that some good outcome occurs *with high probability*. These assertions can also be falsified using a lower bound.

Theorem 5.11 (Principle of Denial for Lower Bounds).

$$\textit{If} \quad \exists \varphi' \Rightarrow \varphi. \quad \mathsf{sat}(\varphi'), \quad \vDash \langle \varphi' \rangle\ C\ \langle \mathbb{P}[\neg A] \geq q \rangle \quad \textit{then} \quad \nvDash \langle \varphi \rangle\ C\ \langle \mathbb{P}[A] \geq p \rangle \quad (\textit{where } q > 1 - p)$$

Note that this implication only goes one way, since the original specification $\mathbb{P}[A] \geq p$ could be satisfied by a sub-distribution μ where $|\mu| < 1$ and therefore $(\mathbb{P}_\mu[A] \ngeq p) \nRightarrow (\mathbb{P}_\mu[\neg A] > 1 - p)$. There are many more special cases for probabilistic falsification, but the relevant cases for the purposes of this paper fall into the categories discussed above.

6 OUTCOME LOGIC FOR MEMORY ERRORS

In this section we specialize OL to prove the existence of memory errors in nondeterministic programs. The program logic is constructed in four layers. First, at its core, there is an assertion logic for describing heaps in the style of separation logic (Section 6.1). On top of that, we build an assertion logic with the capability of describing error states and multiple outcomes (Section 6.2). Then, we define the execution model using a monad combining both errors and nondeterminism (Section 6.3). Finally, we provide proof rules for this multi-layered logic (Section 6.4).

We use this logic in Section 6.5 to reason about memory errors in the style of Incorrectness Separation Logic [Raad et al. 2020]. We also discuss why the semantics of Outcome Logic is a good fit for this type of bug finding by examining manifest errors in more depth (Section 6.6).

6.1 Heap Assertions

First, we create a syntax of logical assertions to describe the heap in the style of Separation Logic [Reynolds 2002]. In order to describe why a program crashed, we need *negative heap assertions* in addition to the standard points-to predicates. These assertions, denoted $x \not\mapsto$, state that the pointer x is invalidated [Raad et al. 2020]. The syntax for the heap assertion logic is below.

$$p \in \mathsf{SL} ::= \mathbf{emp} \mid \exists x.p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p * q \mid p \mathrel{-\!\!*} q \mid e \mid e_1 \mapsto e_2 \mid e \mapsto - \mid e \not\mapsto \qquad (7)$$

In this syntax, $e \in \mathsf{Exp}$ is an expression which includes true and false. We add logical negation $\neg p$ as shorthand for $p \Rightarrow$ false. These assertions are satisfied by a stack and heap pair $(s, h) \in \mathcal{S} \times \mathcal{H}$. Stacks are defined as before (Example 3.4) and heaps are partial functions from positive natural numbers (addresses) to program values or bottom $\mathcal{H} = \{h \mid h \colon \mathbb{N}^+ \rightharpoonup \mathsf{Val} + \{\bot\}\}$[6]. The constant

[6] Note that $\ell \notin \mathsf{dom}(h)$ indicates that we have no information about the pointer ℓ whereas $h(\ell) = \bot$ indicates that ℓ is deallocated. This is why h is both partial *and* includes $\bot$ in the co-domain.

null is equal to 0, so it is not a valid address and therefore $\mathsf{null} \notin \mathsf{dom}(h)$ for any heap h. The semantics of SL is defined in Zilberstein et al. [2023, §E.1] and is similar to that of Raad et al. [2020].

6.2 Reasoning about Errors

While most formulations of Hoare Logic focus only on safe states, descriptions of error states are a fundamental part of Incorrectness Logic [O'Hearn 2019]. Reasoning about errors is built into the semantics of incorrectness triples and the underlying programming languages. In the style of Incorrectness Logic, we use $(\mathsf{ok} : p)$ and $(\mathsf{er} : p)$ to indicate whether or not the program terminated successfully. In our formulation, however, these are regular assertions rather than part of the triples themselves. This makes our assertion logic more expressive because we can describe programs that have multiple outcomes—some of which are successful and some erroneous—in a single triple. The semantics of programs that may crash is also encoded as a monadic effect.

Definition 6.1 (Assertion logic with errors). Given a set of error states E, a set of program states Σ, and the relations $\vDash_E \subseteq E \times \mathsf{Prop}_E$ and $\vDash_\Sigma \subseteq \Sigma \times \mathsf{Prop}_\Sigma$, we construct a new assertion logic with semantics $\vDash \subseteq (E + \Sigma) \times (\mathsf{Prop}_\Sigma \times \mathsf{Prop}_E)$ defined below:

$$\mathbb{i}_L(e) \vDash (p, q) \quad \text{iff} \quad e \vDash_E q \qquad\qquad \mathbb{i}_R(\sigma) \vDash (p, q) \quad \text{iff} \quad \sigma \vDash_\Sigma p$$

In the above, $\mathbb{i}_L : E \to E + \Sigma$ and $\mathbb{i}_R : \Sigma \to E + \Sigma$ are the left and right injections, respectively. We also add syntactic sugar $(\mathsf{ok} : p) \triangleq (p, \mathsf{false})$ and $(\mathsf{er} : q) \triangleq (\mathsf{false}, q)$, so in general the assertion (p, q) can be thought of as $(\mathsf{ok} : p) \vee (\mathsf{er} : q)$. Additional logical operations ($\neg$, $\wedge$, and $\vee$) are defined in Zilberstein et al. [2023, §E.2]. We now combine errors with separation logic as follows:

Definition 6.2 (Separation Logic with Errors). We define an assertion logic as follows:

- The syntax of basic assertions Prop is given in Definition 6.1 with $\mathsf{Prop}_E = \mathsf{Prop}_\Sigma = \mathsf{SL}$, the heap assertion logic (7). So, Prop has the syntax $(\mathsf{ok} : p)$ and $(\mathsf{er} : q)$ where $p, q \in \mathsf{SL}$.
- Σ, the set of program states, is given by $\mathcal{S} \times \mathcal{H}$.
- The satisfaction relation is also given in Definition 6.1 with $E = \Sigma$, so $\vDash \subseteq (\Sigma + \Sigma) \times \mathsf{Prop}$.

6.3 Execution Model

We will now create an execution model supports both nondeterminism and errors by combining the powerset monad (Definition 5.3) with an error monad. We begin by defining the error monad, which is based on taking a coproduct with a set E of errors. In order to use errors in conjunction with another effect (*i.e.,* nondeterminism), we define a monad transformer [Liang et al. 1995]. This is valid since the error monad composes with all other monads [Lüth and Ghani 2002].

Definition 6.3 (Execution model with errors). Given some execution model $\langle M, \mathsf{bind}_M, \mathsf{unit}_M, \diamond, \varnothing \rangle$, we define a new execution model $\langle M(E + -), \mathsf{bind}_{\mathsf{er}}, \mathsf{unit}_{\mathsf{er}}, \diamond, \varnothing \rangle$ such that:

$$\mathsf{bind}_{\mathsf{er}}(m, k) = \mathsf{bind}_M\left(m, \lambda x. \begin{cases} k(y) & \text{if } x = \mathbb{i}_R(y) \\ \mathsf{unit}_M(x) & \text{if } x = \mathbb{i}_L(y) \end{cases}\right) \qquad \mathsf{unit}_{\mathsf{er}}(x) = \mathsf{unit}_M(\mathbb{i}_R(x))$$

Note the monoid definition ($\diamond$ and $\varnothing$) remains the same as the original execution model. For example, if the outer monad is powerset, we still use set union and empty set in the same way—errors only exist within a single outcome.

Example 6.4 (Execution model for nondeterminism and errors). We are particularly interested in the above definition when M is the powerset monad, *i.e.,* $2^{(-)}$. This results in an execution model $\langle 2^{E+-}, \mathsf{bind}, \mathsf{unit}, \diamond, \varnothing \rangle$ where the operations are derived as follows:

$$\mathsf{bind}(S, k) = \{\mathbb{i}_L(x) \mid \mathbb{i}_L(x) \in S\} \cup \bigcup_{\mathbb{i}_R(x) \in S} k(x) \qquad \mathsf{unit}(x) = \{\mathbb{i}_R(x)\}$$

We now turn to defining atomic commands for manipulating the heap in a language called the Guarded Command Language with Memory (mGCL). The syntax for mGCL is given below and the semantics is in Zilberstein et al. [2023, §E.3]. Note that mGCL commands are deterministic and can therefore be interpreted in both nondeterministic and probabilistic evaluation models.

$$c \in \mathsf{mGCL} ::= \mathsf{assume}\ e \mid x := e \mid x := \mathsf{alloc}() \mid \mathsf{free}(e) \mid x \leftarrow [e] \mid [e_1] \leftarrow e_2 \mid \mathsf{error}()$$

Assume and assignment are the same as in GCL (Example 3.4). The usual heap operations for allocation (alloc), deallocation (free), loads ($x \leftarrow [e]$), and stores ($[e_1] \leftarrow e_2$) are also included along with an error command that immediately fails. We also define $x := \mathsf{malloc}()$ as syntactic sugar for $(x := \mathsf{alloc}()) + (x := \mathsf{null})$, which is valid in nondeterministic evaluation models.

Definition 6.5 (Outcome-Based Separation Logic). We instantiate OL (Definition 4.4) with:

(1) The execution model is from Example 6.4 with $E = \mathcal{S} \times \mathcal{H}$.
(2) The language of atomic commands is mGCL.
(3) The assertion logic is the one given in Definition 5.4 using Definition 6.2 for basic assertions.

Note that although the execution model has been augmented with errors, the nondeterministic falsification result (Theorem 5.6) still holds for Outcome-Based Separation Logic.

6.4 Proof Rules for Memory Errors

Now that we have defined the semantics of OL triples that can express properties about memory and errors, let us turn to the proof theory. In this section, we will define proof rules for Outcome-Based Separation Logic (Definition 6.5), which we will use in subsequent sections to prove that programs crash due to memory errors. We will define these proof rules in a way that is *generic* with respect to the execution model, leveraging the fact that the atomic mGCL commands are deterministic, and thus can be given specifications that hold good under multiple different execution models (*e.g.*, nondeterminism or probabilistic computation).

Concretely, let us observe that the semantics of mGCL is based on the composition of two monads: an outer monad M (*e.g.*, powerset), and the error monad $E + -$. Since the atomic commands of mGCL are deterministic, however, their semantics is agnostic to the choice of the outer monad M, and can be specified axiomatically without needing to talk explicitly about (multiple) outcomes. Hence, we define a new type of triple that is capable of making assertions about errors (using er and ok), but says nothing about outcomes (using $\oplus$):

Definition 6.6 (Liftable Triples). Consider an OL instance based on the composition of two monads $M = M_1 \circ M_2$, and so $[\![C]\!] : \Sigma \to (M_1 \circ M_2)\Sigma$ (note that any monad can be decomposed in this way, by taking $M_2 = \mathsf{Id}$). One such example is the execution model from Example 6.4 where $M_1 = 2^{(-)}$ and $M_2 = E + -$. The validity of an OL triple that is *liftable* into the monad M_1 is defined as follows:

$$\vDash_{M_1} \langle p \rangle\ C\ \langle q \rangle \quad \text{iff} \quad \forall \sigma \in M_2\Sigma. \quad \sigma \vDash p \;\Rightarrow\; \exists \tau \in M_2\Sigma.\ [\![C]\!]^\dagger(\mathsf{unit}_{M_1}(\sigma)) = \mathsf{unit}_{M_1}(\tau) \text{ and } \tau \vDash q$$

Intuitively, this triple says that C is deterministic; if we run it on any individual state that satisfies p, then the result will be an individual state satisfying q. In the case of Example 6.4, this means that p and q describe elements of $E + \Sigma$; they can describe error states (using er and ok), but cannot use $\oplus$. Similarly, we write $\vdash_{M_1} \langle p \rangle\ C\ \langle q \rangle$ to denote a liftable derivation, which is sound with respect to the above semantics and can be lifted into the monad M_1.

Figure 5 contains the proof rules for Outcome-Based Separation Logic (Definition 6.5). The first group of rules is very close to the standard separation logic proof system originally due to O'Hearn et al. [2001], with the addition of rules to reason about unsafe states inspired by Raad et al. [2020]. These rules are liftable into any monad M (since errors compose with all other monads). The

Separation Logic Small Axioms

$$\frac{}{\vdash_M \langle \mathsf{ok} : p \rangle\ \mathsf{error}()\ \langle \mathsf{er} : p \rangle}\ \textsc{Error} \qquad \frac{}{\vdash_M \langle \mathsf{ok} : x = v \land \mathsf{emp} \rangle\ x := \mathsf{alloc}()\ \langle \mathsf{ok} : x \mapsto - \rangle}\ \textsc{Alloc}$$

$$\frac{}{\vdash_M \langle \mathsf{ok} : e \mapsto - \rangle\ \mathsf{free}(e)\ \langle \mathsf{ok} : e \not\mapsto \rangle}\ \textsc{Free Ok} \qquad \frac{}{\vdash_M \langle \mathsf{ok} : e \not\mapsto \rangle\ \mathsf{free}(e)\ \langle \mathsf{er} : e \not\mapsto \rangle}\ \textsc{Free Er}$$

$$\frac{}{\vdash_M \langle \mathsf{ok} : e_1 \mapsto - \rangle\ [e_1] \leftarrow e_2\ \langle \mathsf{ok} : e_1 \mapsto e_2 \rangle}\ \textsc{Store Ok} \qquad \frac{}{\vdash_M \langle \mathsf{ok} : e_1 \not\mapsto \rangle\ [e_1] \leftarrow e_2\ \langle \mathsf{er} : e_1 \not\mapsto \rangle}\ \textsc{Store Er}$$

$$\frac{}{\vdash_M \langle \mathsf{ok} : x = m \land e \mapsto n \rangle\ x \leftarrow [e]\ \langle \mathsf{ok} : x = n \land e[m/x] \mapsto n \rangle}\ \textsc{Load Ok} \qquad \frac{}{\vdash_M \langle \mathsf{ok} : e \not\mapsto \rangle\ x \leftarrow [e]\ \langle \mathsf{er} : e \not\mapsto \rangle}\ \textsc{Load Er}$$

$$\frac{\vdash_M \langle \epsilon : p \rangle\ c\ \langle \epsilon' : q \rangle \qquad \mathsf{fv}(r) \cap \mathsf{mod}(c) = \emptyset}{\vdash_M \langle \epsilon : p * r \rangle\ c\ \langle \epsilon' : q * r \rangle}\ \textsc{Frame}$$

Monadic Rules

$$\frac{\vdash_{\mathbb{2}^{(-)}} \langle p \rangle\ C\ \langle q \rangle}{\langle p \rangle\ C\ \langle q \rangle}\ \textsc{Nondeterministic Lifting} \qquad \frac{}{\vdash_M \langle \mathsf{er} : p \rangle\ C\ \langle \mathsf{er} : p \rangle}\ \textsc{Error Propagation}$$

Fig. 5. Proof rules for Outcome-Based Separation Logic, the OL instantiation of Definition 6.5. The first group is inspired by O'Hearn et al.'s [2001] small axioms with additional rules added for unsafe states. The second group deal with the monadic execution model.

Lifting proof rule states that if some triple is liftable into the powerset monad (where p and q are satisfied by individual states), then we can obtain a new triple where p and q are satisfied by sets of states (as in Definition 5.4). All of the small axioms above can be lifted in this way.

In order to use the proof rules for conditionals and assignment from Figure 4, we also define expression entailment and substitution. Both operations are only defined for ok assertions.

$$(\mathsf{ok} : p) \vDash e \quad \text{iff} \quad p \Rightarrow e \qquad\qquad (\mathsf{ok} : p)[e/x] \triangleq \mathsf{ok} : p[e/x]$$

This means that, for example, the Assign rule only allows us to prove $\langle \mathsf{ok} : p[e/x] \rangle\ x := e\ \langle \mathsf{ok} : p \rangle$. If an error has occurred, we instead use the Error Propagation rule to propagate the error forward through the proof (*i.e.*, $\langle \mathsf{er} : p \rangle\ x := e\ \langle \mathsf{er} : p \rangle$), since the program will never recover from the crash.

6.5 Proof of a Bug

We now demonstrate that the OL proof system shown in Figure 5 is effective for bug-finding. The program in Figure 6 has a possible use-after-free error. This program first appeared as a motivating example for ISL [Raad et al. 2020]. It models a common error in C++ when using the `std::vector` library. A call to push_back can reallocate the vector's underlying memory buffer, in which case pointers to that buffer become invalid.

As in Raad et al. [2020], we model the vector as a single heap location, and the push_back function nondeterministically chooses to either reallocate the buffer or do nothing. A subsequent memory access may then fail, as seen in the main function. Since our language does not have procedures, we model these as macros and prove the existence of the bug with all the code inlined. The proof mostly makes use of standard separation logic proof rules and is quite similar to the ISL version [Raad et al. 2020] especially in the use of negative heap assertion after the call to free. Under-approximation is achieved using the rule of consequence to drop one of the outcomes.

Correctness for this program would be given by the postcondition $(\mathsf{ok} : v \mapsto x * x \mapsto 1)$. As Theorem 5.6 showed, we can disprove it by showing that an undesirable outcome will sometimes

$$
\begin{array}{l}
\mathsf{main}():\\
\quad x \leftarrow [v] \fatsemi\\
\quad \mathsf{push\_back}(v) \fatsemi\\
\quad [x] \leftarrow 1\\
\\
\mathsf{push\_back}(v):\\
\quad \left(\begin{array}{l} y \leftarrow [v] \fatsemi\\ \mathsf{free}(y) \fatsemi\\ y := \mathsf{alloc}() \fatsemi\\ [v] \leftarrow y \end{array}\right) + \mathsf{skip}
\end{array}
\qquad\Bigg|\qquad
\begin{array}{l}
\langle \mathsf{ok} : v \mapsto a * a \mapsto - \rangle\\
x \leftarrow [v] \fatsemi\\
\langle \mathsf{ok} : v \mapsto x * x \mapsto - \rangle\\
\left(\begin{array}{l}
\langle \mathsf{ok} : v \mapsto x * x \mapsto - \rangle\\
y \leftarrow [v] \fatsemi\\
\langle \mathsf{ok} : v \mapsto x * x \mapsto - \land y = x \rangle\\
\mathsf{free}(y) \fatsemi\\
\langle \mathsf{ok} : v \mapsto x * x \not\mapsto \land y = x \rangle\\
y := \mathsf{alloc}() \fatsemi\\
\langle \mathsf{ok} : v \mapsto x * x \not\mapsto * y \mapsto - \rangle\\
[v] \leftarrow y\\
\langle \mathsf{ok} : v \mapsto y * x \not\mapsto * y \mapsto - \rangle
\end{array}\right)
+
\begin{array}{l}
\langle \mathsf{ok} : v \mapsto x * x \mapsto - \rangle\\
\mathsf{skip}\\
\langle \mathsf{ok} : v \mapsto x * x \mapsto - \rangle
\end{array}
\fatsemi\\
\langle (\mathsf{ok} : v \mapsto y * x \not\mapsto * y \mapsto -) \oplus (\mathsf{ok} : v \mapsto x * x \mapsto -) \rangle\\
[x] \leftarrow 1\\
\langle (\mathsf{er} : v \mapsto y * x \not\mapsto * y \mapsto -) \oplus (\mathsf{ok} : v \mapsto x * x \mapsto 1) \rangle\\
\implies \langle (\mathsf{er} : x \not\mapsto * \mathsf{true}) \oplus \top \rangle
\end{array}
$$

Fig. 6. Program with a possible use-after-free error (left) and proof sketch (right)

occur. In this case, that undesirable outcome is $(\mathsf{er} : x \not\mapsto * \mathsf{true})$. Clearly, $(\mathsf{er} : x \not\mapsto * \mathsf{true}) \oplus \top$ implies $\neg(\mathsf{ok} : v \mapsto x * x \mapsto 1)$, so the specification in Figure 6 disproves the correctness specification.

6.6 Manifest Errors

Le et al. [2022] showed empirically that the fix rates of bug-finding tools can be improved by reporting only those bugs that occur regardless of context. These errors are known as *manifest errors*, as demonstrated in the examples below.

$$\langle \mathsf{ok} : x \not\mapsto \rangle\ [x] \leftarrow 1\ \langle \mathsf{er} : x \not\mapsto \rangle \qquad \langle \mathsf{ok} : \mathsf{true} \rangle\ x := \mathsf{malloc}() \fatsemi [x] \leftarrow 1\ \langle (\mathsf{er} : x = \mathsf{null}) \oplus \top \rangle$$

The left program has a *latent* error since it is only triggered if the pointer is already deallocated, therefore it would not be reported. The right program has a *manifest* error since it is possible no matter the context in which the program is invoked. Le et al. [2022, Def. 3.2] give the following definition for manifest errors:

$$\vDash [\mathsf{ok} : p]\ C\ [\mathsf{er} : q] \quad \text{is a manifest error} \qquad \text{iff} \qquad \forall \sigma. \quad \exists \tau \in [\![C]\!](\sigma). \quad \tau \vDash (\mathsf{er} : q * \mathsf{true})$$

First note that the precondition p does not appear in the formal definition. This indicates that IL preconditions do not meaningfully describe the conditions *sufficient* to reach an end state. In addition, the universal quantification over the precondition resembles Hoare Logic more closely than Incorrectness Logic (which quantifies over the postcondition). As stated in the following lemma, the formal definition of a manifest error can be expressed as an OL triple.

Lemma 6.7 (Manifest Error Characterization).

$$\vDash [p]\ C\ [\mathsf{er} : q] \quad \textit{is a manifest error} \qquad \textit{iff} \qquad \vDash^{\downarrow} \langle \mathsf{ok} : \mathsf{true} \rangle\ C\ \langle \mathsf{er} : q * \mathsf{true} \rangle$$

Following from this result, determining whether a program has a manifest error is equivalent to proving an OL triple of the form above. Characterizing a manifest error using IL is much harder. Le et al. [2022] provide an algorithm to do so, which involves several satisfiability checks (which are NP-hard). The difficulty in characterizing manifest errors suggests that under-approximation in IL is too powerful. To see this, we compare the standard If rule from OL to One-Sided If—a hallmark of IL which allows the analysis to only consider one branch of an if statement.

$$\frac{\langle \mathsf{ok} : p \land e \rangle\ C_1\ \langle \epsilon : q \rangle \qquad \langle \mathsf{ok} : p \land \neg e \rangle\ C_2\ \langle \epsilon : q \rangle}{\langle \mathsf{ok} : p \rangle\ \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \langle \epsilon : q \rangle}\,\textsc{If} \qquad \frac{[p \land e]\ C_1\ [\epsilon : q]}{[p]\ \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ [\epsilon : q]}\,\textsc{One-Sided If}$$

$$\frac{\vdash_{\mathcal{D}} \langle A \rangle\ C\ \langle B \rangle}{\langle \mathbb{P}[A] = p \rangle\ C\ \langle \mathbb{P}[B] = p \rangle}\textsc{Lifting} \qquad \frac{\forall v \in \mathrm{supp}(\eta).\ \langle A \rangle\ x := v\ \langle B_v \rangle}{\langle \mathbb{P}[A] = p \rangle\ x \xleftarrow{\$} \eta\ \langle \bigoplus_{v \in \mathrm{supp}(\eta)} (\mathbb{P}[B_v] = p \cdot \eta(v)) \rangle}\textsc{Sample}$$

Fig. 7. Probabilistic proof rules.

One-Sided If generates imprecise preconditions since the precondition of the premise ($p \land e$) is stronger than the precondition of the conclusion (p). OL, on the other hand, requires the precondition to be precise enough to force the execution down a specific logical path, otherwise both paths must be considered as seen in the If rule. As such, OL enables under-approximation in *just the right ways*; only outcomes that result from nondeterministic choice can be dropped.

Le et al.'s [2022] discussion of manifest errors suggests that *sufficient* preconditions are important; we need to know what happens when we run the program on *any* state satisfying the precondition. Interestingly, there is no analogous motivation for covering the whole postcondition (as IL does). Reachability is important, but we only have to reach *some* error state, not *all* of them. In fact, as we will see in our exploration of probabilistic programming, covering the entire post is often infeasible.

7 PROBABILISTIC INCORRECTNESS

Randomization is a powerful tool that is seeing increased adoption in mainstream software development as it is essential for machine learning and security applications. The study of probabilistic programming has a rich history [Kozen 1979, 1983], but there is little prior work on proving that probabilistic programs are *incorrect*. In Section 5.2, we gave a theoretical result showing that probabilistic specifications in OL can be disproven. In this section, we provide a proof system for probabilistic OL and use it to prove incorrectness in a particular example program.

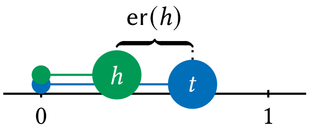
We work with the probabilistic OL instance using the evaluation model from Definition 5.8 and the outcome assertions in Definition 5.9. The basic commands are assignment and assume from GCL (Example 3.4) with probabilistic sampling added ($x \xleftarrow{\$} \eta$). There are only two proof rules for the probabilistic language, given in Figure 7. The Lifting rule allows us to lift a derivation (e.g. for variable assignment) into a probabilistic setting. This is sound, since every state in the support must transition from A to B, thus $\mathbb{P}[A]$ before running C is equal to $\mathbb{P}[B]$ after. The Sample rule splits the postcondition into a separate outcome for each value in the support of η.

The rules for conditional branching in Figure 4 can be used in probabilistic proofs by defining expression entailment $(\mathbb{P}[A] = p) \vDash e$ iff $A \vDash e$. Assign can also be used; substitution propagates inside the probabilistic assertion $(\mathbb{P}[A] = p)[e/x] = (\mathbb{P}[A[e/x]] = p)$. Note that the conditional rules require us to know the probability that the guard is true or false upfront. This is standard for probabilistic Hoare Logics [Barthe et al. 2018; den Hartog 2002].

Absent are rules for while loops. Looping rules in probabilistic languages are complex since invariants cannot be used when probabilities change across iterations. Such proof rules are certainly expressible in our model, but are out of scope for this paper. For examples of how this is done, see Barthe et al. [2018]; den Hartog [2002].

7.1 Error Bounds for Machine Learning

Randomization is often used in approximation algorithms where computing the exact solution to a problem is difficult. In these applications, some amount of error is acceptable as long as it is likely to be small. One such application is supervised learning algorithms, which produce a *hypothesis* from a set of labelled examples. The examples are members of some set X and are drawn randomly from some probability distribution $\eta \in \mathcal{D}X$. The hypothesis is a function $h : X \to \mathbb{B}$ which guesses whether new data points are positive or negative examples.



```
$h := -1 \fatsemi$
for $N$ do
  $x \xleftarrow{\$} [0,1] \fatsemi$
  if $\mathcal{L}(x) \land x > h$ then
    $h := x$
  else
    skip
```

```
$\langle \mathbb{P}[\mathsf{true}] = 1 \rangle \implies \langle \mathbb{P}[\mathsf{er}(-1) > \varepsilon] \geq (1-\varepsilon)^0 \rangle$
 $h := -1 \fatsemi$
$\langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (1-\varepsilon)^0 \rangle$
 for $N$ do
   $\langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (1-\varepsilon)^i \rangle$
    $x \xleftarrow{\$} [0,1] \fatsemi$
   $\langle \bigoplus_{q \in [0,1]} \mathbb{P}[\mathsf{er}(h) > \varepsilon \land x = q] \geq \Delta \cdot (1-\varepsilon)^i \rangle$
    if $\mathcal{L}(x) \land x > h$ then
      $\langle \bigoplus_{q \in (h,t]} \mathbb{P}[\mathsf{er}(h) > \varepsilon \land x = q] \geq \Delta \cdot (1-\varepsilon)^i \rangle$
       $\implies \langle \bigoplus_{x \in (h,t-\varepsilon]} \mathbb{P}[\mathsf{er}(x) > \varepsilon] \geq \Delta \cdot (1-\varepsilon)^i \rangle$
       $\implies \langle \mathbb{P}[\mathsf{er}(x) > \varepsilon] \geq (t - \varepsilon - h) \cdot (1-\varepsilon)^i \rangle$
       $h := x$
      $\langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (t - \varepsilon - h) \cdot (1-\varepsilon)^i \rangle$
    else
      $\langle \bigoplus_{q \in [0,h] \cup (t,1]} \mathbb{P}[\mathsf{er}(h) > \varepsilon \land x = q] \geq \Delta \cdot (1-\varepsilon)^i \rangle$
       skip$\fatsemi$
      $\langle \bigoplus_{q \in [0,h] \cup (t,1]} \mathbb{P}[\mathsf{er}(h) > \varepsilon \land x = q] \geq \Delta \cdot (1-\varepsilon)^i \rangle$
       $\implies \langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (h + 1 - t) \cdot (1-\varepsilon)^i \rangle$
   $\langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (t - \varepsilon - h) \cdot (1-\varepsilon)^i \oplus \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (h + 1 - t) \cdot (1-\varepsilon)^i \rangle$
    $\implies \langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (1-\varepsilon)^{i+1} \rangle$
$\langle \mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (1-\varepsilon)^N \rangle$
```

Fig. 8. The interval learning problem: a diagram of the learning problem (top left), a program implementing interval learning (bottom left), and a proof sketch (right).

Consider the simple learning problem in which we want to learn a point $t \in [0,1] \subset \mathbb{R}$ on the unit interval. Since we require distributions used in programs to be finite, we can approximate $[0,1]$ as $\{k \cdot \Delta \mid 0 \leq k \leq \frac{1}{\Delta}\}$ for some finite *step size* Δ. Anything in the interval $[0,t]$ is considered a positive example, and anything greater than t is a negative example. This concept is illustrated at the top of Figure 8 and the program below—expressed in a probabilistic extension of GCL—learns this concept by repeatedly sampling examples and refining the hypothesis h after each round. The resulting hypothesis is always equal to the largest positive example that the algorithm has seen. Therefore it will always classify negative examples correctly and only make mistakes on positive examples between h and t.

The *labelling* oracle $\mathcal{L}(x) = x \leq t$ gives the true label of any point on the interval. Let $\mathsf{er}(h) = t - h$ be the error of the hypothesis (the total probability mass between h and t). The goal is to determine the probability that h has error greater than ε after N iterations. Practically speaking, this simulates training the model on a dataset of size N. Intuitively, the error will be less than ε if the algorithm ever samples an example in the interval $[t - \varepsilon, t]$. The chance of getting a hit in this range increases greatly with the number of examples seen. While this problem may seem contrived, it is a 1-dimensional version of the *Rectangle Learning Problem* which is known to have practical applications and the proof ideas are extensible to other learnable concepts [Kearns and Vazirani 1994].

To prove that this program is *correct*, we want to say that the resulting hypothesis has small error with high probability. Choosing an error bound ε and a confidence parameter δ, we say that the program is correct if at the end $\mathbb{P}[\mathsf{er}(h) \leq \varepsilon] \geq 1 - \delta$. Now, we can look to Theorem 5.11 to determine how to *disprove* the correctness specification. We need to show that the probability of the opposite happening ($\mathsf{er}(h) > \varepsilon$) is higher than δ. Based on the derivation in Figure 8, we conclude that the program is incorrect if $(1 - \varepsilon)^N > \delta$. Suppose we had a dataset of size $N = 100$ and desired at most 1% error ($\varepsilon = 0.01$) with 90% likelihood ($\delta = 0.1$). Then the postcondition tells us that the error is higher than 1% with probability at least 37%. Clearly 37% $> \delta$, so the program is incorrect; we need a larger dataset in order to get a better result.

7.2 Probabilistic Incorrectness Logic

It is natural to ask whether a similar result could be achieved using a probabilistic variant of Incorrectness Logic. However, such a program logic is cumbersome and produces poor characterizations of errors. To show this, we begin by examining the semantics of a probabilistic IL triple.

$$\vDash [P]\ C\ [Q] \qquad \text{iff} \qquad \forall \mu \vDash Q.\quad \exists \mu'.\quad \mu \sqsubseteq [\![C]\!]^{\dagger}(\mu') \quad \text{and} \quad \mu' \vDash P$$

This definition differs from standard Incorrectness Logic in two ways. First, assertions are satisfied by *distributions* over program states $\mu \in \mathcal{D}\Sigma$ rather than individual program states $\sigma \in \Sigma$. This is necessary in order to make the assertion logic quantitative. Second, under-approximation is achieved using the sub-distribution relation $\sqsubseteq$ instead of set inclusion[7]. As is typical with Incorrectness Logic, this definition stipulates that *any* subdistribution satisfying the postcondition must be reachable by an execution of the program. While in non-probabilistic cases it can already be hard to fully characterize a valid end-state, even more information is needed in the probabilistic case.

To demonstrate this, consider the interval learning program from Figure 8. The postcondition of this triple is $\mathbb{P}[\mathsf{er}(h) > \varepsilon] \geq (1-\varepsilon)^N$, which is not a valid postcondition for an incorrectness triple because it does not adequately describe the final distribution. That is, there are many distributions satisfying this assertion that could not result from running the program. In one such distribution, $h = -1$ with probability 1. So, lower bounds are not suitable for use in Incorrectness Logic because a distribution can be invented where the probability is arbitrarily large, rendering it unreachable. But changing the inequality to an equality to obtain $\mathbb{P}[\mathsf{er}(h) > \varepsilon] = (1-\varepsilon)^N$ does not solve the problem. This assertion can be satisfied by a distribution where $h = -1$ with probability $(1-\varepsilon)^N$, which is also unreachable. In order for an assertion to properly characterize the output distribution, it has to specify all the possible values of h. Such an assertion is given below:

$$\bigoplus_{x=0}^{\varepsilon} \left(\mathbb{P}[\mathsf{er}(h) = x] = (1-x)^N - (1-(x+\Delta))^N \right)$$

The original assertion was easy to understand; we immediately knew the probability of having a large error. By contrast, the added information needed for IL actually obscures the result. It is not useful to know the probability of each value of h, we only care about bounding the probability that $\mathsf{er}(h) > \varepsilon$. In general, Probabilistic Incorrectness Logic requires us to specify the *entire joint distribution* over all the program variables which is certainly undesirable and often infeasible.

Many techniques in probabilistic program analysis summarize the output distribution in alternative ways. This includes using expected values [Kaminski 2019; Morgan et al. 1996] and probabilistic independence [Barthe et al. 2019]. If those techniques are used to express correctness, it makes sense that similar ideas would be desirable for incorrectness. However, techniques that *summarize* a distribution are incompatible with Incorrectness Logic since they do not specify the output distribution in a sufficient level of detail. Based on these findings, we conclude that developing probabilistic variants of Incorrectness Logic is not a promising research direction. In fact, the differences between correctness and incorrectness are often quite blurred in probabilistic examples. Since some amount of error is typically expected, it is not possible to reason about correctness *without* reasoning about incorrectness. It is therefore sensible that a unified theory captures both.

8 RELATED WORK

Incorrectness reasoning and program analysis. In motivating Incorrectness Logic (IL), O'Hearn [2019] posed the twin challenges of *sound* and *scalable* incorrectness reasoning: program logics for incorrectness must guarantee *true positive* bugs, while also supporting *under-approximation*

[7] This order is defined pointwise: $\mu_1 \sqsubseteq \mu_2$ iff $\forall x.\mu_1(x) \leq \mu_2(x)$.

in order to scale to large codebases. Outcome Logic (OL) takes inspiration from those challenges, but offers a solution that is closer to traditional Hoare Logic [Hoare 1969] and, as such, is also compatible with correctness reasoning.

Outcome Logic was also inspired in part by *Lisbon triples*, which were first described in a published article by Möller et al. [2021, §5] under the name *backwards under-approximate triples*.[8] The semantics of Lisbon triples is based on Hoare's [1978] calculus of *possible correctness*: for any initial state satisfying the precondition, there exists *some* trace of execution leading to a final state satisfying the postcondition. As such, Lisbon triples describe true positives (behaviors that are witnessed by an actual trace, assuming the pre is satisfiable). As recounted by O'Hearn [2019, §7], Lisbon triples predate Incorrectness Logic; Derek Dreyer and Ralf Jung suggested them as a foundation for incorrectness reasoning during a discussion with Peter O'Hearn and Jules Villard that took place at POPL'19 in Lisbon (hence the name "Lisbon Triples").

Shortly thereafter, O'Hearn developed the semantics of IL triples. His major motivation for developing IL (instead of further exploring Lisbon triples) was the goal of finding a logical foundation for *scalable* bug-catching static analysis tools (such as Pulse-X [Le et al. 2022]), and one key to scalability is the ability to discard program paths (aka "drop disjuncts") during analysis. More concretely, the analysis accumulates a disjunction of assertions which symbolically represents the set of possible states at each program point. If this set gets too large, then it is important to be able to drop some of the disjuncts in order to save memory and computation time. Thanks to its reverse rule of consequence—which supports *strengthening* of the postcondition—IL provides a sound logical foundation for dropping disjuncts, whereas Lisbon triples do not.

One can see OL as a generalization of Lisbon triples which supports discarding of program paths in a different way than IL does: namely, via the *outcome conjunction* connective, which enables reasoning about multiple executions at the same time.[9] Specifically, "disjuncts" arise in a program analysis when the program makes a *choice* to branch based on either a logic condition (*e.g.*, an if statement or while loop) or a computational effect (*e.g.*, nondeterminism or randomization). In IL, both types of choice are encoded by standard disjunction. In OL, on the other hand, we distinguish these two forms of choice by using disjunction ($\vee$) for the former and outcome conjunction ($\oplus$) for the latter. This leads to a different approach for supporting discarding of program paths, but one which we believe can serve as an alternative logical foundation for practical static analyses.

Let us first consider the case of choices arising from computational effects. Incorrectness Logic includes a Choice rule that allows analyses to drop one branch of a nondeterministic choice. An analogous derived rule is also sound in OL [Zilberstein et al. 2023, §B.2]; both are shown below.

$$\frac{[P]\ C_1\ [Q]}{[P]\ C_1 + C_2\ [Q]}\ \textsc{Choice (IL)} \qquad\qquad \frac{\langle P\rangle\ C_1\ \langle Q\rangle}{\langle P\rangle\ C_1 + C_2\ \langle Q \oplus \top\rangle}\ \textsc{Under-Approx (OL)}$$

Given that nondeterministic variants of OL provide reachability guarantees, it may appear surprising that a conclusion about $C_1 + C_2$ can be made without showing that C_2 terminates. However, the assertion $\top$ encompasses all outcomes (including nontermination), so this inference is valid. Note that there are also symmetric versions of these rules where the C_2 branch is instead taken.

Let us now consider the case of choices arising from logical conditions, where the differences between OL and IL are more pronounced. Consider the following program, which will only fail in the case that b is true.

$$\langle \mathsf{ok} : x \mapsto -\rangle\ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{free}(x)\ \mathsf{else}\ \mathsf{skip} \fatsemi [x] \leftarrow 1\ \langle (\mathsf{ok} : x \mapsto 1) \vee (\mathsf{er} : x \not\mapsto)\rangle$$

[8] Though Le et al. [2022, §3.2] also mention backwards under-approximate triples, their potential has gone largely unexplored.

[9] In Zilberstein et al. [2023, §C], we show that Lisbon triples are in fact a special case of OL.

The semantics of OL does not permit us to simply drop one of the disjuncts in the postcondition. If we want to only explore the program path in which the error occurs, then we need to push information about the logical condition b backwards into the precondition.

$$\langle \mathsf{ok} : x \mapsto - \land b\rangle \ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{free}(x)\ \mathsf{else}\ \mathsf{skip} \fatsemi [x] \leftarrow 1 \ \langle \mathsf{er} : x \not\mapsto\rangle$$

This is in contrast to Incorrectness Logic, in which we *can* drop disjuncts, *but in return* we need to ensure that every state described by the postcondition is reachable. More precisely, $(\mathsf{er} : x \not\mapsto)$ is not a strong enough IL postcondition for the aforementioned program because it includes the unreachable state in which $x \not\mapsto$, but b is false. In IL, one must therefore specify the bug as follows:

$$[x \mapsto -] \ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{free}(x)\ \mathsf{else}\ \mathsf{skip} \fatsemi [x] \leftarrow 1 \ [\mathsf{er} : x \not\mapsto \land b]$$

So, in either case we must record the same amount of information about the logical condition b. The difference is whether this information appears in the pre- or postcondition. As we discussed in Section 6.6, there are advantages to having a more precise precondition (as OL does): it enables us to easily determine how to trigger a bug and characterize manifest errors. Conversely, the precise postconditions required by IL make it difficult to design abstract domains, suggesting that IL is not compatible with popular analysis techniques like abstract interpretation [Ascari et al. 2022].

Furthermore, in order to generate more useful bug reports and error traces for the user, practical static analysis tools like Pulse-X [Le et al. 2022] *do* in any case push logical conditions backwards to the pre-condition using a technique called bi-abduction [Calcagno et al. 2009, 2011]. This suggests that while the theories of OL and IL differ substantially, it may be possible to build practical static analysis tools atop OL in a similar manner to IL-based tools like Pulse-X. We plan to investigate this further in future work.

Unifying correctness and incorrectness. Parallel efforts have been made to unify correctness and incorrectness reasoning within a single program logic. Bruni et al. [2021, 2023] introduced Local Completeness Logic (LCL), which is based on Incorrectness Logic, but with limits on the rule of consequence such that an over-approximation of the reachable states can always be recovered from the postcondition. Similarly, Exact Separation Logic (ESL) [Maksimović et al. 2022] combines the semantics of IL and Hoare Logic in triples that exactly describe the reachable states.

Both of these logics are capable of proving correctness properties as well as finding true bugs. But they achieve this by compromising the ability to use the rule of consequence, which is crucial to scalable analysis algorithms. Analyses based on Hoare Logic use consequences to abstract the postcondition, reducing the information overhead and aiding in finding loop invariants. Analyses based on IL use consequences to drop disjuncts and consider fewer program paths. Since neither type of consequence is valid in LCL and ESL, it remains unclear whether those theories can feasibly serve as the foundation of practical tools. By contrast, Outcome Logic enjoys the full power of the (forward) rule of consequence and can also drop nondeterministic paths.

There has also been work to connect the theories of correctness and incorrectness algebraically using Kleene Algebra with Tests (KAT) [Kozen 1997], an equational theory for reasoning about program equivalence. Möller et al. [2021]; Zhang et al. [2022] showed that both Hoare Logic and IL can be embedded in variants of KAT and used this insight to formalize connections between the two types of specifications. While this provides an algebraic theory powerful enough to capture Hoare Logic and IL, this connection does not go as deep as the unification offered by OL and does not provide a clear path to shared analyses for both program verification and bug finding.

Since our paper was conditionally accepted to OOPSLA, a closely related paper has appeared on arXiv, which presents a program logic, called Hyper Hoare Logic, for proving and disproving program hyper-properties (properties relating multiple program traces) [Dardinier and Müller 2023]. It achieves this using the same underlying semantics as Outcome Logic instantiated to the powerset monad. Their work shows the applicability of the OL model beyond the usage scenarios that we envisioned in this paper.

Separation logic and Iris. While both separation logic [O'Hearn et al. 2001; Reynolds 2002] and Outcome Logic employ Bunched Implications [O'Hearn and Pym 1999] as a fundamental part of their metatheories, the way in which BI is used in each case is substantially different.

In separation logic and its extensions such as Iris [Jung et al. 2018, 2015], the value of the BI resource monoid is neatly demonstrated by the Frame Rule, which enables local reasoning by adding assertions about unused resources to the pre- and postconditions of some smaller proof derivation. In this way, framing allows us to talk about the same program execution with additional (unused) resources. By contrast, the outcome conjunction deals with assertions about *different* program executions.

The Frame Rule is in general unsound with respect to the outcome conjunction. To demonstrate this, we use the same counterexample that Reynolds [2002] used to demonstrate that the Rule of Constancy is unsound in separation logic:

$$\frac{\langle x \mapsto - \rangle\ [x] \leftarrow 4\ \langle x \mapsto 4 \rangle}{\langle x \mapsto - \oplus y \mapsto 3 \rangle\ [x] \leftarrow 4\ \langle x \mapsto 4 \oplus y \mapsto 3 \rangle}\textsc{Frame}$$

It is easy to see that this is an invalid inference. The outcome conjunction does not preclude that x and y are aliased, in which case it must be that $y \mapsto 4$ in the postcondition. Instead, we have the Split rule (Figure 4), which allows us to analyze a program separately for each outcome in the precondition and then compose the resulting outcomes in the postcondition.

This example shows that, although both separation logic and OL use BI, the two logics are modeling two very different aspects of the program (resource usage vs. program outcomes, respectively), and the resulting program logics are therefore different.

OL and separation logic are not mutually exclusive. In Section 6, we saw how separation logic can be embedded in OL. In addition, we believe that combining OL with Iris is a very interesting direction for future research: Iris offers advanced mechanisms to reason modularly about concurrency, and OL offers a way to extend Hoare Logic to be amenable to both correctness and incorrectness reasoning. Combining the two would result in a program logic capable of proving the existence of bugs in concurrent programs (while a concurrent version of Incorrectness Logic already exists [Raad et al. 2022], it is not built atop Iris and does not support the full capabilities offered by Iris).

In a concurrent version of Outcome Logic, outcomes would model possible interleavings of concurrent branches. In an assertion of the form $P \oplus \top$, the predicate P could describe an undesirable outcome that occurs in *some* of those interleavings (*i.e.,* a bug), which is not currently possible to express in Iris.

Probabilistic and quantitative program analysis. Probabilistic variants of Hoare Logic [Barthe et al. 2018; den Hartog 2002; Rand and Zdancewic 2015; Tassarotti and Harper 2019] were a major source of inspiration for the design of Outcome Logic. Whereas pre- and postconditions of standard Hoare Logic describe individual program states, probabilistic variants of Hoare Logic use assertions that describe *distributions* over program states. These logics also include connectives similar to the outcome conjunction, but specialized to probability distributions. In Outcome Logic, we generalize from probability distributions to support a wider variety of PCMs.

Starting with the seminal work of Kozen [1979, 1983], expected values have been a favorite choice for probabilistic program analysis. Morgan et al. [1996]'s weakest-pre-expectation (wpe) calculus computes expected values of program expressions with an approach similar to Dijkstra's [1976] Weakest Precondition calculus. Many extensions to wpe have arisen, including to handle nondeterminism, runtimes [Kaminski 2019], and Separation Logic [Batz et al. 2019]. This line of work has not intersected with Incorrectness Logic since the semantics of weakest-pre is incompatible with IL, although Batz et al. [2019] hinted at the nuanced interaction between correctness and incorrectness in quantitative settings with their "faulty garbage collector" example. We hope that our new perspective—using Hoare Logic for incorrectness—will encourage the use of wpe calculi for bug-finding.

Zhang and Kaminski [2022] developed a Quantitative Strongest Post (QSP) calculus and noted its connections to IL, which was originally characterized by O'Hearn [2019] in terms of Dijkstra's [1976] strongest-post. QSP is an interesting foundation for studying the Galois Connections between types of quantitative program specifications, although the goals are somewhat orthogonal to our own in that we sought to *unify* correctness and incorrectness rather than explore dualities.

9 CONCLUSION

Formal methods for incorrectness remain a young field. The foundational work of O'Hearn [2019] has already led to several program logics for proving the existence of bugs such as memory errors, memory leaks, data races, and deadlocks [Le et al. 2022; Raad et al. 2020, 2022]. However, as with any new field there are growing pains—manifest errors and probabilistic programs are an awkward fit in the original formulation of IL. This has inspired us to pursue a new theory incorporating O'Hearn's [2019] core tenets of incorrectness—true positives and under-approximation—while also accounting for more evaluation models and different types of incorrectness. Outcome Logic achieves just that, with the added benefit of unifying the theories of correctness and incorrectness in a single program logic. Our Falsification Theorem (Theorem 5.1) shows that any OL triple can be disproven within the logic. So, any bug invalidating a correctness specification can be expressed. OL also offers a cleaner characterization of manifest errors, suggesting it may be semantically closer to the way that programmers reason about bugs.

In this paper, we introduced OL as a theoretical basis for incorrectness reasoning, but in the future we plan to further explore its practical potential as well. Incorrectness Logic has been shown to scale well as an underlying theory for bug-finding in large part due to its ability to *drop disjuncts* [Le et al. 2022; Raad et al. 2020]; analysis algorithms accumulate a disjunction of possible outcomes as they move forward through a program, and due to the semantics of IL, these disjuncts can be soundly pruned to keep the search space small. Hoare Logics (including OL) cannot drop disjuncts. However, as we saw in Section 2 and Section 4, OL *can* drop *outcomes*, which we believe is sufficient to make the algorithm scale to large codebases (although this remains to be demonstrated). Furthermore, since OL triples can be used both for correctness and incorrectness reasoning, we plan to develop a bi-abductive [Calcagno et al. 2009, 2011] algorithm to infer procedure summaries that can be used by *both* correctness verification and bug-finding analyses.

When O'Hearn [2019] remarked that "program correctness and incorrectness are two sides of the same coin," he was expressing that just as programmers spend significant mental energy debugging (reasoning about *incorrectness*), we in the formal methods community must invent sound reasoning principles for incorrectness. We take this idea one step further, suggesting that program correctness and incorrectness are two *usages* of the same *program logic*. We hope that this unifying perspective will continue to invigorate the field of incorrectness reasoning and invite the reuse of tools and techniques that have already been successfully deployed for correctness reasoning.

ACKNOWLEDGMENTS

We thank Peter O'Hearn, Josh Berdine, Azalea Raad, Jules Villard, Quang Loc Le, and Julien Vanegue for their helpful feedback. This work has been supported in part by the Defense Advanced Research Projects Agency under Contract HR001120C0107.

REFERENCES

Krzysztof R. Apt. 1981. Ten Years of Hoare's Logic: A Survey–Part I. *ACM Trans. Program. Lang. Syst.* 3, 4 (oct 1981), 431–483. https://doi.org/10.1145/357146.357150

Flavio Ascari, Roberto Bruni, and Roberta Gori. 2022. Limits and difficulties in the design of under-approximation abstract domains. In *Foundations of Software Science and Computation Structures*, Patricia Bouyer and Lutz Schröder (Eds.). Springer International Publishing, Cham, 21–39. https://doi.org/10.1007/978-3-030-99253-8_2

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 117–144. https://doi.org/10.1007/978-3-319-89884-1_5

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

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

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

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* (feb 2023). https://doi.org/10.1145/3582267 Just Accepted.

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

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

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

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

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

Jerry den Hartog. 2002. *Probabilistic Extensions of Semantical Models.* Ph.D. Dissertation. Vrije Universiteit Amsterdam. https://core.ac.uk/reader/15452110

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

Edsger W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall. I–XVII, 1–217 pages.

Simon Docherty. 2019. *Bunched logics: a uniform approach.* Ph.D. Dissertation. University College London. https://discovery.ucl.ac.uk/id/eprint/10073115/

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

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

C. A. R. Hoare. 1978. Some Properties of Predicate Transformers. *J. ACM* 25, 3 (Jul 1978), 461–480. https://doi.org/10.1145/322077.322088

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000151

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

Benjamin Lucien Kaminski. 2019. *Advanced weakest precondition calculi for probabilistic programs.* Dissertation. RWTH Aachen University, Aachen. https://doi.org/10.18154/RWTH-2019-01829 Veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Dissertation, RWTH Aachen University, 2019.

Michael J. Kearns and Umesh V. Vazirani. 1994. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA.

Dexter Kozen. 1979. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (SFCS '79)*. 101–114. https://doi.org/10.1109/SFCS.1979.38

Dexter Kozen. 1983. A Probabilistic PDL. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. Association for Computing Machinery, New York, NY, USA, 291–297. https://doi.org/10.1145/800061.808758

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195

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

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

Christoph Lüth and Neil Ghani. 2002. Composing Monads Using Coproducts. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/581478.581492

Petar Maksimović, Caroline Cronjäger, Julian Sutherland, Andreas Lööw, Sacha-Élie Ayoun, and Philippa Gardner. 2022. Exact Separation Logic. https://doi.org/10.48550/ARXIV.2208.07200

Bernhard Möller, Peter O'Hearn, and Tony Hoare. 2021. On Algebra of Program Correctness and & Incorrectness. In *Relational and Algebraic Methods in Computer Science: 19th International Conference, RAMiCS 2021, Marseille, France, November 2–5, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 325–343. https://doi.org/10.1007/978-3-030-88701-8_20

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

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

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. http://www.jstor.org/stable/421090

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

Benjamin C. Pierce. 1991. *Basic Category Theory for Computer Scientists*. MIT Press. https://doi.org/10.7551/mitpress/1524.001.0001

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

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

Robert Rand and Steve Zdancewic. 2015. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs. In *Electronic Notes in Theoretical Computer Science*, Vol. 319. 351–367. https://doi.org/10.1016/j.entcs.2015.12.021 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

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

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

Hongseok Yang. 2001. *Local Reasoning for Stateful Programs*. Ph.D. Dissertation. USA. Advisor(s) Reddy, Uday S. https://dl.acm.org/doi/10.5555/933728

Cheng Zhang, Arthur Azevedo de Amorim, and Marco Gaboardi. 2022. On Incorrectness Logic and Kleene Algebra with Top and Tests. *Proc. ACM Program. Lang.* 6, POPL, Article 29 (jan 2022), 30 pages. https://doi.org/10.1145/3498690

Linpeng Zhang and Benjamin Lucien Kaminski. 2022. Quantitative Strongest Post: A Calculus for Reasoning about the Flow of Quantitative Information. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 87 (apr 2022), 29 pages. https://doi.org/10.1145/3527331

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation of Correctness and Incorrectness Reasoning (Full Version). https://doi.org/10.48550/arXiv.2303.03111

Received 2022-10-28; accepted 2023-02-25