

Revealing Sources of (Memory) Errors via Backward Analysis

FLAVIO ASCARI, University of Pisa, Italy

ROBERTO BRUNI, University of Pisa, Italy

ROBERTA GORI, University of Pisa, Italy

FRANCESCO LOGOZZO, Meta Platforms, USA

Sound over-approximation methods are effective for proving the absence of errors, but inevitably produce false alarms that can hamper programmers. In contrast, under-approximation methods focus on bug detection and are free from false alarms. In this work, we present two novel proof systems designed to locate the source of errors via backward under-approximation, namely Sufficient Incorrectness Logic (SIL) and its specialization for handling memory errors, called Separation SIL. The SIL proof system is minimal, sound and complete for Lisbon triples, enabling a detailed comparison of triple-based program logics across various dimensions, including negation, approximation, execution order, and analysis objectives. More importantly, SIL lays the foundation for our main technical contribution, by distilling the inference rules of Separation SIL, a sound and (relatively) complete proof system for automated backward reasoning in programs involving pointers and dynamic memory allocation. The completeness result for Separation SIL relies on a careful crafting of both the assertion language and the rules for atomic commands.

CCS Concepts: • **Theory of computation** → **Logic and verification**; *Proof theory*; *Hoare logic*; **Separation logic**; *Programming logic*.

Additional Key Words and Phrases: Sufficient Incorrectness Logic, Incorrectness Logic, Outcome Logic

ACM Reference Format:

Flavio Ascari, Roberto Bruni, Roberta Gori, and Francesco Logozzo. 2025. Revealing Sources of (Memory) Errors via Backward Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 127 (April 2025), 28 pages. <https://doi.org/10.1145/3720486>

1 Introduction

Formal methods aim to automate the improvement of software reliability and security. Notable success stories are, e.g., the Astrée static analyzer [Blanchet et al. 2003], the SLAM model checker [Ball and Rajamani 2001], the certified C compiler CompCert [Leroy 2009], VCC for safety properties verification [Cohen et al. 2009], and the Frama-C platform for the integration of many C code analyses [Baudin et al. 2021]. Despite that, effective program correctness methods struggle to reach mainstream adoption, mostly because they exploit over-approximation to handle decidability issues and false positives are seen as a distraction by expert programmers. Being free from false positives is possibly the reason why *under-approximation* approaches for bug-finding, such as testing and bounded model checking, are preferred in industrial applications. Incorrectness Logic (IL) [O’Hearn 2020] is a new program logic for bug-finding: *any error state found in the post can be produced by some input states that satisfy the pre*. However, IL triples are not able to characterize precisely *the input states that are responsible for a given error*. This is possibly rooted in the *forward* flavor of the under-approximation, which follows the ordinary direction of code execution.

Authors’ Contact Information: Flavio Ascari, University of Pisa, Pisa, Italy, flavio.ascari@phd.unipi.it; Roberto Bruni, University of Pisa, Pisa, Italy, bruni@di.unipi.it; Roberta Gori, University of Pisa, Pisa, Italy, roberta.gori@unipi.it; Francesco Logozzo, Meta Platforms, Seattle, USA, logozzo@meta.com.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/4-ART127

<https://doi.org/10.1145/3720486>

Logic	HL Hoare	IL Incorrectness	NC Nec. Cond.	SIL Suff. Incorrectness	OL Outcome
Triples	$\{P\} r \{Q\}$	$[P] r [Q]$	$(P) r (Q)$	$\langle\!\langle P \rangle\!\rangle r \langle\!\langle Q \rangle\!\rangle$	$\langle P \rangle r \langle Q \rangle$

Fig. 1. The figure summarizes the syntax of all triples used in this paper, SIL being our novel contribution.

Backward Under-approximation. At POPL’19 in Lisbon, D. Dreyer and R. Jung suggested that P. O’Hearn should look at bug-finding in terms of a logic for proving the presence of faults (as reported in O’Hearn [2020]; Zilberstein et al. [2023]). Such “Lisbon” triples express that *for any initial state satisfying the pre, there exists some run leading to a final state satisfying the post*, likewise Hoare [1978]’s calculus of possible correctness, even if no form of approximation was considered there. Lisbon triples were then briefly discussed in Möller et al. [2021, §5] and Le et al. [2022, §3.2] under the name *backwards under-approximate triples*. More recently, the general framework of Outcome Logic (OL) [Zilberstein et al. 2023] has been shown able to account for both Lisbon triples and Hoare triples, recognizing the importance of tracking the sources of errors.

Ideally, given an incorrectness specification, the goal of backward under-approximation would be to unveil all dangerous input states that lead to bugs. However, all the above proposals are designed according to the forward semantics of programs and thus their rules are best suited to infer postconditions starting from any given precondition: in general, their backward analysis may require the instantiation of their consequence rules with some ingenious guess, because inference rules are not applicable to any post and there is no a priori best strategy guaranteed to succeed.

Main Contributions. Aiming to enhance program analysis with the ability to identify the source of incorrectness, we design a backward-oriented reasoning framework, where under-approximation is exploited to prevent false alarms. More precisely, our contribution is twofold.

First, on a more conceptual level, we present Sufficient Incorrectness Logic (SIL), a minimal, sound, and complete proof system for Lisbon triples. To some extent, SIL spells out, a posteriori, a formalization and generalization of the backward analysis step performed by industrial grade analysis tools for security oriented developed at Meta [Bleaney and Cepel 2019; Distefano et al. 2019; Gabi 2021]. Backward-orientation means that SIL rules are designed to be applicable to any post. The SIL rules are conceived with an emphasis on clarity and conciseness, allowing for a thorough comparison of various program logics by outlining their scopes, goals, relationships, and differences.

Second, we equip the SIL conceptual framework with the machinery of separation logic to detect inputs that witness memory errors: the resulting Separation SIL is our main technical contribution. We prove that Separation SIL is both sound and relatively complete, demonstrate how to automate its backward reasoning, and establish connections with other incorrectness logics. A prototype implementation of Separation SIL has been instrumental in developing all the proposed examples.

Below, we provide a more detailed description of our contributions.

How SIL fits in the Taxonomy of Program Logics. SIL focuses on *finding the sources of incorrectness rather than highlighting the presence of bugs*. Assuming the post Q defines some class of errors, i.e., it is an *incorrectness specification*, the SIL triple $\langle\!\langle P \rangle\!\rangle r \langle\!\langle Q \rangle\!\rangle$ means that “*all input states satisfying P have at least one execution of the program r leading to a state satisfying Q* ”. For deterministic programs, SIL guarantees that a state satisfying the pre *always* leads to an error. For nondeterminism programs, SIL guarantees that there *exists* an execution leading to an error.¹ SIL triples are highly valuable to programmers: by pointing out the sources of errors, they serve as a starting point to scope down debugging, fuzzing, and testing. Moreover, SIL can expose manifest

¹In other words, SIL follows the angelic resolution of nondeterminism, as opposed to Hoare logic’s demonic resolution.

	Forward		Backward
Over	HL: $\{P\} r \{Q\}$		NC: $(P) r (Q)$
	$\llbracket r \rrbracket P \subseteq Q$	$\xleftrightarrow[\text{(Prop. 3.12)}]{\cong}$	$\llbracket \ulcorner \rrbracket Q \subseteq P$
	$\forall \sigma \in P. \forall \sigma' \in \llbracket r \rrbracket \sigma. \sigma' \in Q$		$\forall \sigma' \in Q. \forall \sigma \in \llbracket \ulcorner \rrbracket \sigma'. \sigma \in P$
Under	IL: $[P] r [Q]$		SIL: $\langle\langle P \rangle\rangle r \langle\langle Q \rangle\rangle$
	$\llbracket r \rrbracket P \supseteq Q$	$\xleftrightarrow[\text{(Lemma 4.9)}]{\cong}$	$\llbracket \ulcorner \rrbracket Q \supseteq P$
	$\forall \sigma' \in Q. \exists \sigma \in P. \sigma' \in \llbracket r \rrbracket \sigma$		$\forall \sigma \in P. \exists \sigma' \in Q. \sigma \in \llbracket \ulcorner \rrbracket \sigma'$

Fig. 2. The taxonomy of validity conditions. Columns indicate if validity is based on forward or backward semantics and rows the verse of approximation. Arrows denote validity-preserving transformations.

errors [Le et al. 2022, §3.2]: an error Q is manifest iff the SIL triple $\langle\langle \text{true} \rangle\rangle r \langle\langle Q \rangle\rangle$ is valid. These are bugs that happen regardless of the context and are thus more likely to be fixed when reported.

Below, we give detailed insights into the intricate relations among SIL, Hoare Logic (HL) [Hoare 1969], Incorrectness Logic (IL) [O’Hearn 2020] and Necessary Conditions (NC) [Cousot et al. 2013, 2011] by constructing a taxonomy of program logics along several axes of comparison. We do not think that any one logic is superior to another; instead, we focus on clarifying which logic is best suited for achieving specific goals and how they can be effectively used in synergy.

HL is a well established logic for (partial) correctness. The HL triple $\{P\} r \{Q\}$ means that if P holds initially, any terminating execution of r ends in a state satisfying Q . IL is designed to detect true bugs, without false alarms. The IL triple $[P] r [Q]$ means that all the (error) states in Q are reachable from some states in P . NC provides a principled foundation for automatic precondition inference. A necessary precondition only rules out entry states that will inevitably lead to errors, *without* removing any good execution. The notation for the various triples is summarized in Fig. 1.

We realized that the comparisons among program logics can be better understood by arranging them in the grid in Fig. 2, where their validity conditions w.r.t. a program r with pre P and post Q are made explicit. Given a program r , we write $\llbracket r \rrbracket$ for its forward collecting semantics and $\llbracket \ulcorner \rrbracket$ for its backward one. The expression $\llbracket r \rrbracket P$ denotes the set of all possible output states of r when execution starts from a state in P (and r terminates). Vice versa, $\llbracket \ulcorner \rrbracket Q$ denotes the largest set of input states that can lead to some state in Q .² From the validity conditions we can, e.g., easily derive the corresponding consequence rules: in HL and SIL we can weaken Q and strengthen P , while in IL and NC we can do the opposite, i.e., consequence rules are aligned to the diagonals in Fig. 2. Moreover, this classification highlights connections, such as the isomorphisms between HL and NC via negation and the one between IL and SIL via the notion of opposite relation.

Separation SIL: A Sound and Complete Logic for Sources of Memory Errors. Separation SIL specializes SIL to handle dynamic memory allocation. Notably, Separation SIL is not only sound but also (relatively) complete. A relevant consequence of completeness is that weakest SIL preconditions can always be expressed in the assertion language. This depends on the level of expressiveness of the assertion language, which strikes a balance between the ability to express meaningful post but not those whose corresponding precondition cannot be represented in the language. Although we exclude negation and universal quantification from the assertion language, it remains expressive enough to include the properties expressible in industrial level tools based on Separation Logic,

²Although $\llbracket r \rrbracket$ and $\llbracket \ulcorner \rrbracket$ define opposite relations, it is not always the case that they are inverse to each other, because of nondeterminism and nontermination. Also please note that $\llbracket \ulcorner \rrbracket$ is not Dijkstra’s weakest liberal precondition $\text{wlp}[r]$.

```

// function r
x := nondet();
if (x=1) {
  if (y≤100) {
    rmc } }

// function rmc
while (x>0) {
  if (y>100) {
    y := y-10; x := x-1
  } else {
    y := y+11; x := x+1 } }

```

(a) A non deterministic program r .(b) r_{mc} for computing McCarthy's 91 function.

Fig. 3. A simple program used to compare the validity of triples in different program logics

Table 1. Different logics for different goals

	SIL	IL	HL	NC
$[\text{true}] r [y = 91 \wedge x \neq 1]$	✗	✓	✗	✓
$\langle\langle y \leq 100 \rangle\rangle r \langle\langle y = 91 \wedge x \neq 1 \rangle\rangle$	✓	✓	✗	✓
$\langle\langle y \leq 100 \rangle\rangle r \langle\langle y = 91 \rangle\rangle$	✓	✗	✗	✓
$\langle\langle y < 91 \rangle\rangle r \langle\langle y = 91 \rangle\rangle$	✓	✗	✗	✗

such as Meta Infer and Pulse. If we added negation and universal quantification to the assertion language, our proof system would remain sound but would no longer be complete. We illustrate Separation SIL on the motivating example of Incorrectness Separation Logic (ISL) [Raad et al. 2020] (Example 5.2). In comparison with ISL, we argue that Separation SIL triples can exhibit both a more succinct post capturing the error and provide a stronger guarantee, as *every* state in the pre is incorrect. With respect to Outcome Separation Logic (OSL) [Zilberstein et al. 2024], we show that a key difference resides in the memory allocation model. Therefore, not all valid Separation SIL triples can be derived in OSL. Finally, we show how to automate the backward analysis of Separation SIL, leading to a prototype tool that helped us to gain confidence in our results. Since our assertion language aligns with state-of-the-art tools, we anticipate potential integration opportunities.

A Small Running Example. To illustrate the validity conditions of different logics, we outline some comparison by considering the simple program r in Fig. 3, a non deterministic program computing McCarthy's 91 function [Manna 1974] under some circumstances.³ Assume that the incorrectness specification $Q_{91} \triangleq (y = 91)$ denotes the set of erroneous states.

The IL triple $[\text{true}] r [Q_{91} \wedge x \neq 1]$ proves that the program r is incorrect. It tells us that any state satisfying Q_{91} and $x \neq 1$ can be actually reached by some input state. Note that the constraint $x \neq 1$ is necessary in the post, because the state where $y = 91$ and $x = 1$ is not reachable. The same IL triple is also valid in NC, but neither in HL nor in SIL. For the validity in HL it is required that the post is satisfied by all possible final states; since the program is nondeterministic, the post should include all the possible initial values of y . For the validity in SIL instead it is needed that all the inputs satisfying the pre lead to an error, and this is not the case for example for $y = 101$ (the summary of this discussion is reported in Table 1, 1st line). Therefore, while the post of IL contains useful information, the pre true cannot guide the testing because, e.g., starting with $y = 101$ the program is correct. Of course the trivial pre true is extremely weak and a real-world implementation of IL, like tools in the Pulse family, would provide more detailed information about the path conditions leading to errors. However, by design, IL can always weaken preconditions introducing states that do not lead to any error, a fact that we show through this ad-hoc example.

³If $x = 1$ and $y \leq 100$, then r_{mc} will terminate with $x = 0$ and $y = 91$.

On the other hand, using SIL proof system in synergy with the post $Q_{91} \wedge x \neq 1$ detected by IL, we can derive the SIL triple $\langle y \leq 100 \rangle r \langle Q_{91} \wedge x \neq 1 \rangle$, which proves that any input state where $y \leq 100$ can cause the error. This triple is also valid in IL (see Table 1, 2nd line) but its derivation in IL would require the ingenious guess of the pre and would not guarantee that the error can be produced *whenever* $y \leq 100$. Moreover, since our incorrectness specification is Q_{91} , we could also start directly from such post and use SIL proof system to infer the triple $\langle y \leq 100 \rangle r \langle Q_{91} \rangle$, which is not valid in IL. Finally, to separate SIL from NC, we mention that the triple $\langle y < 91 \rangle r \langle Q_{91} \rangle$ is valid in SIL but not in NC, because NC triples must include in the pre *all* the incorrect inputs.

Structure of the paper. Related work is discussed in Section 2 and some basic concepts and notation are presented in Section 3. In Section 4 we introduce SIL and illustrate its role in the taxonomy of program analysis. In Section 5, we define Separation SIL over a suitable language of assertions. We conclude in Section 6 pointing out some future work.

2 Related Work

We firmly believe that, in judging between equally expressive rival theories, simplicity and ease-of-use should prove best rationales for scientists.

The origins of Lisbon triples have already been discussed in Section 1. In the literature, backward under-approximation triples were mentioned in Le et al. [2022]; Möller et al. [2021], but neither of them fully developed a corresponding program logic. Le et al. [2022, §3.2] also introduce the notion of *manifest errors*, a class of bugs particularly relevant in practice. Using SIL, a post Q is a manifest error iff the triple $\langle \text{true} \rangle r \langle Q \rangle$ is provable.

Zilberstein et al. [2023] already realized the importance of finding the sources of errors: it was one of the motivations leading to Outcome Logic. OL is able to express both SIL and HL triples, and characterize manifest errors. However, OL rules are better suited for forward inference, while SIL rules are designed for syntax-driven backward inference. While OL provides a framework that can be instantiated to many different computational models, the trade off for this generality can make OL instances less precise than specialized proof systems. We show an example of this for Outcome Separation Logic [Zilberstein et al. 2024] and Separation SIL in Section 5.9.

Raad et al. [2024] independently introduces a forward-oriented proof system with a core set of rules that are sound for both IL and SIL. It also becomes complete for IL, resp. SIL, when augmented with the corresponding consequence rule.

Exact Separation Logic (ESL) [Maksimovic et al. 2023] combines Separation Logic and Incorrectness Separation Logic to recover the exact semantics of a program and produce function summaries that can be used for both correctness and incorrectness. In the schema in Fig. 2, ESL would be placed between HL and IL. Following this idea, we could position the predicate transformer of possible correctness [Hoare 1978] between NC and SIL, as it is exact; OL along the diagonal between HL and SIL; and the forward-oriented core set of rules in Raad et al. [2024, Fig. 1] between IL and SIL.

Using Abstract Interpretation techniques, Cousot [2024] defines a general procedure to derive proof systems starting from the underlying triple validity conditions. All validity conditions considered here are also represented in Cousot's framework by combining a small set of abstractions in all possible ways, as summarized in Cousot [2024, Fig. 3]. While a proof system generated according to Cousot's recipe is sound and complete, it is not guaranteed to provide the most convenient inference rules from the applicability point of view. For example, in the case of heap manipulating programs, the locality principle and the Frame rule central to approached based on separation logics have not received yet a satisfactory treatment within Cousot's calculational framework.

Zhang and Kaminski [2022] propose a calculus for quantitative weakest pre and strongest post that subsumes the Boolean case. Using these, they propose a classification reminiscent of our

taxonomy. However, they neither develop proof systems, nor compare in detail SIL and IL with other logics. Moreover, contrary to SIL, their predicate transformers do not perform approximation during the computation. For instance, while SIL can under-approximate loops by finite unrolling, their calculus computes the semantics of the loop exactly. An analogous argument holds for other related predicate transformers, such as Hoare’s weakest possible precondition calculus $\mathbf{wpp}[r](S)$ [Hoare 1978, §5.3] and Verscht and Kaminski [2025]’s angelic weakest precondition $\mathbf{awp}[r](S)$.

Dynamic Logic (DL) [Harel et al. 2000] is an extension of modal logic which is able to describe program properties. All four logics in Fig. 2 can be encoded in DL, so it may be used to gain more insight on their connections. However, to the best of our knowledge, DL has never been used to compare program logics this way, nor to identify the source of incorrectness.

Underspecified Incorrectness Logic (UIL) [Verbeek et al. 2025] is a variant of IL that enables backward reasoning. While similar in goals to SIL, a UIL triple prescribes reachability of all final states (modulo underspecified variables) and imposes no guarantee on the initial states.

3 Background

3.1 Regular Commands.

Following O’Hearn [2020], we consider a simple language of regular commands $r \in \text{RCmd}$ over (integer) arithmetic expressions $a \in \text{AExp}$ and Boolean expressions $b \in \text{BExp}$, i.e., we let:

$$\begin{aligned} \text{ACmd} \ni c &:: \text{skip} \mid x := a \mid b? & \text{RCmd} \ni r &:: c \mid r; r \mid r \boxplus r \mid r^* & (1) \\ \text{AExp} \ni a &:: n \mid x \mid a \diamond a & \text{BExp} \ni b &:: \text{false} \mid \neg b \mid b \wedge b \mid a \asymp a \end{aligned}$$

where $n \in \mathbb{Z}$ is a natural number, x is a (integer) variable, $\diamond \in \{+, -, \cdot, \dots\}$ encodes standard arithmetic operations, and $\asymp \in \{=, \neq, \leq, <, \dots\}$ standard comparison operators.

Regular commands provide a general template which can be instantiated differently by changing the set ACmd of atomic commands c . The set RCmd determines the kind of operations allowed in the language. Here we list the inaction command skip , the standard deterministic assignment $x := a$ and the “assume” statement $b?$: if the input satisfies b it does nothing, otherwise it diverges. At times, we also use $x := \text{nondet}()$ to describe a nondeterministic assignment. Sequential composition is written $r; r$ and nondeterministic choice $r \boxplus r$. The Kleene star r^* denotes a nondeterministic iteration, where r can be executed any number of time (possibly none) before exiting.

This formulation can accommodate for a standard imperative programming language [Winskel 1993] by defining conditionals and while-loops as the macros below:

$$\text{if } (b) \{r_1\} \text{ else } \{r_2\} \triangleq (b?; r_1) \boxplus ((\neg b)?; r_2) \quad \text{while } (b) \{r\} \triangleq (b?; r)^*; (\neg b)?$$

Given our instantiation of ACmd , we consider a finite set of variables Var and the set of stores $\Sigma \triangleq \text{Var} \rightarrow \mathbb{Z}$ that are total functions σ from Var to integers. We tacitly assume that 0 is the default value of uninitialized variables. Given a store $\sigma \in \Sigma$, store update $\sigma[x \mapsto v]$ is defined as usual for $x \in \text{Var}$ and $v \in \mathbb{Z}$. We consider an inductively defined semantics $\llbracket \cdot \rrbracket$ for arithmetic and Boolean expressions such that $\llbracket a \rrbracket \sigma \in \mathbb{Z}$ and $\llbracket b \rrbracket \sigma \in \mathbb{B}$ for all $a \in \text{AExp}$, $b \in \text{BExp}$ and $\sigma \in \Sigma$. The semantics of atomic commands $\llbracket \cdot \rrbracket : \text{ACmd} \rightarrow \Sigma \rightarrow \wp(\Sigma)$ is defined below (where $\sigma \in \Sigma$):

$$\llbracket \text{skip} \rrbracket \sigma \triangleq \{\sigma\} \quad \llbracket x := a \rrbracket \sigma \triangleq \{\sigma[x \mapsto \llbracket a \rrbracket \sigma]\} \quad \llbracket b? \rrbracket \sigma \triangleq \begin{cases} \{\sigma\} & \text{if } \llbracket b \rrbracket \sigma = \mathbf{tt} \\ \{\} & \text{otherwise} \end{cases}$$

from which we derive the (forward) semantics of regular commands $\llbracket \cdot \rrbracket : \text{RCmd} \rightarrow \Sigma \rightarrow \wp(\Sigma)$ ⁴

$$\llbracket c \rrbracket \triangleq \llbracket c \rrbracket \quad \llbracket r_1; r_2 \rrbracket \sigma \triangleq \llbracket r_2 \rrbracket (\llbracket r_1 \rrbracket \sigma) \quad \llbracket r_1 \boxplus r_2 \rrbracket \sigma \triangleq \llbracket r_1 \rrbracket \sigma \cup \llbracket r_2 \rrbracket \sigma \quad \llbracket r^* \rrbracket \sigma \triangleq \bigcup_{n \geq 0} \llbracket r \rrbracket^n \sigma \quad (2)$$

⁴Since in general r is possibly nondeterministic and nonterminating, $\llbracket r \rrbracket \sigma$ may contain none, one or many states.

where, in the previous equation and in the rest of the paper, we overload the symbol $\llbracket \cdot \rrbracket$ to denote also its additive lifting to sets of states $\llbracket \cdot \rrbracket : \text{ACmd} \rightarrow \wp(\Sigma) \rightarrow \wp(\Sigma)$, defined by $\llbracket r \rrbracket S \triangleq \bigcup_{\sigma \in S} \llbracket r \rrbracket \sigma$. Roughly, $\llbracket r \rrbracket S$ is the set of output states reachable from the input states in S , i.e., it coincides with the usual strongest postcondition calculus $\text{sp}[r](S)$ and it is also known under the name *collecting semantics*. $\llbracket r \rrbracket$ can also be seen as a binary relation over Σ , such that $(\sigma, \sigma') \in \llbracket r \rrbracket$ if and only if $\sigma' \in \llbracket r \rrbracket \sigma$. This allows us to define the backward semantics $\llbracket \overleftarrow{r} \rrbracket : \text{RCmd} \rightarrow \Sigma \rightarrow \wp(\Sigma)$ as the function yielding the opposite relation of the forward semantics (also written $\llbracket \cdot \rrbracket^{\text{op}}$), that is

$$\sigma \in \llbracket \overleftarrow{r} \rrbracket \sigma' \iff \sigma' \in \llbracket r \rrbracket \sigma \quad \text{or, equivalently,} \quad \llbracket \overleftarrow{r} \rrbracket \sigma' \triangleq \{\sigma \mid \sigma' \in \llbracket r \rrbracket \sigma\}. \quad (3)$$

As before, we additively lift the definition of backward semantics to set of states by union. Roughly, $\llbracket \overleftarrow{r} \rrbracket S$ is the set of input states that can lead to some output states in S , i.e., it coincides with Hoare's weakest possible precondition calculus $\text{wpp}[r](S)$ [Hoare 1978, §5.3]. The backward semantics can also be characterized compositionally, similarly to the forward one:

LEMMA 3.1. *For any $r, r_1, r_2 \in \text{RCmd}$ and $S \in \wp(\Sigma)$, all of the following equalities hold:*

$$\llbracket \overleftarrow{r_1}; r_2 \rrbracket = \llbracket \overleftarrow{r_1} \rrbracket \circ \llbracket \overleftarrow{r_2} \rrbracket \quad \llbracket \overleftarrow{r_1 \boxplus r_2} \rrbracket S = \llbracket \overleftarrow{r_1} \rrbracket S \cup \llbracket \overleftarrow{r_2} \rrbracket S \quad \llbracket \overleftarrow{r^*} \rrbracket S = \bigcup_{n \geq 0} \llbracket \overleftarrow{r} \rrbracket^n S$$

REMARK 3.2. *The notation $\llbracket \overleftarrow{r} \rrbracket$ may suggest that the backward semantics is the same as applying the forward semantics to some sort of reversed program $(r)^{\text{op}}$, defined inductively as*

$$(r_1; r_2)^{\text{op}} \triangleq (r_2)^{\text{op}}; (r_1)^{\text{op}} \quad (r_1 \boxplus r_2)^{\text{op}} \triangleq (r_1)^{\text{op}} \boxplus (r_2)^{\text{op}} \quad (r^*)^{\text{op}} \triangleq ((r)^{\text{op}})^*$$

However, it is important to remark that the reversal $(c)^{\text{op}}$ of some generic atomic command c is not necessarily expressible in the program syntax. Particularly, while one can correctly define $(\text{skip})^{\text{op}} \triangleq \text{skip}$ and $(b?)^{\text{op}} \triangleq b?$, to reverse the assignment $(x := a)^{\text{op}}$ in the language of regular commands we would need to extend the syntax with some sort of “revoking” assignment $a =: x$ that, given a state σ , returns a state $\sigma[x \mapsto v]$ for some value v such that $(\llbracket a \rrbracket \sigma)[x \mapsto v] = \sigma(x)$, if such v exists.

REMARK 3.3 (ASSERTION LANGUAGE). *The properties of triple-based program logics depend on the expressiveness of the formulae (the assertion language) allowed in pre and post [Blass and Gurevich 2001]. As an example consider HL. When the assertion language is first-order logic, HL is sound but not complete, because first-order logic is not able to represent all the properties needed to prove completeness, notably loop invariants [Apt 1981, §2.7]. If the assertion language is not close under conjunctions and disjunctions HL may be unsound [Cousot et al. 2012, Ex. 5]. To overcome the aforementioned problems, following Cousot et al. [2012] and until Section 5, we assume P and Q to be set of states instead of logic formulae—we therefore write, e.g., $\sigma \in P$ to say that the state σ satisfies the precondition P .*

3.2 Hoare Logic

Hoare logic (HL) [Hoare 1969] is a well known triple-based logic for proving properties about programs. The HL triple $\{P\} r \{Q\}$ means that, whenever the execution of r begins in a state σ satisfying P and it ends in a state σ' , then σ' satisfies Q . When Q is a correctness specification, then any HL triple $\{P\} r \{Q\}$ provides a *sufficient* condition P for the so called partial correctness of the program r . Formally, this is described by the over-approximation property of postconditions

$$\llbracket r \rrbracket P \subseteq Q, \quad (\text{HL})$$

so we say that a triple satisfying this inclusion is *valid* in HL. In its original formulation, HL was given for a deterministic while-language and assuming P and Q were formulae in a given logic. Subsequent work generalized it to many other settings, such as nondeterminism [Apt 1984] and regular commands [Möller et al. 2021], whose proof system is summarized in Fig. 4 (3rd column).

THEOREM 3.4 (HL IS SOUND AND COMPLETE [COOK 1978]). *A HL triple is provable using the rules in Fig. 4 (3rd column, top quadrant) if and only if it is valid.*

REMARK 3.5 (ON THE MEANING OF “FORWARD”). *Although the validity condition (HL) is expressed in terms of the forward semantics and thus classifies HL as a forward over-approximation in our taxonomy, the reader should not be misled to think the adjective forward refers to direction of the deduction process, which is totally independent of our classification. For example, it is well known that HL is related to Dijkstra’s weakest liberal precondition [Dijkstra 1975] for a given post Q : a triple $\{P\} r \{Q\}$ is valid if and only if $P \subseteq \mathbf{wlp}[r](Q)$. In this case, given the correctness specification Q , HL triples can be used for backward program analysis to find the weakest pre for program correctness.*

Example 3.6. Consider the program r of Example 3 and the correctness specification $\neg Q_{91} \triangleq (y \neq 91)$. The valid HL triple $\{y > 100\} r \{\neg Q_{91}\}$ assures us that if our input satisfies the precondition our program will be correct. The same result could be obtained using Dijkstra’s weakest liberal precondition, indeed, $\mathbf{wlp}[r](\neg Q_{91}) = (y > 100)$. Note that if we considered the incorrectness specification instead, that is Q_{91} , we would obtain $\mathbf{wlp}[r](Q_{91}) = (y = 91)$. \square

3.3 Incorrectness Logic

Incorrectness Logic (IL) [O’Hearn 2020] was introduced as a formalism for finding true bugs in the code via under-approximation. The IL triple $[P] r [Q]$ means that all the states in Q are reachable from states in P , i.e., that for any state $\sigma' \in Q$ there is a state $\sigma \in P$ such that σ' is reachable from σ . Therefore, if $\sigma' \in Q$ is an “error” state, the triple guarantees it is a true alarm of the analysis. This is described by the under-approximation property

$$\llbracket r \rrbracket P \supseteq Q, \quad (\text{IL})$$

which characterizes *valid* IL triples. The rules, inspired by previous work on reverse Hoare logic [de Vries and Koutavas 2011] and here simplified to not separate correct and erroneous termination states, are shown in Fig. 4 (2nd column).

THEOREM 3.7 (IL IS SOUND AND COMPLETE [O’HEARN 2020]). *An IL triple is provable using the rules in Fig. 4 (2nd column, top quadrant) if and only if it is valid.*

Example 3.8 (HL vs IL). The valid IL triple $[\text{true}] r [y = 91 \wedge x \neq 1]$ in Table 1 is not valid in HL. Vice versa, the valid HL triple $\{y > 100\} r \{y \neq 91\}$ from Example 3.6 is not valid in IL. \square

REMARK 3.9 (OK/ER FLAGS). *One distinguishing feature of proof systems for incorrectness [O’Hearn 2020; Raad et al. 2020, 2022, 2023] is to tag post, but not pre, with either the flag *ok* or *er* to separate successful and erroneous computations. An immediate consequence is that the number of proof rules increases to deal with such tags and that the definition of the semantics becomes longer and more complex. Striving for simplicity, we follow the alternative exploited in Bruni et al. [2023], where the whole concrete domain is extended with such flags, i.e., $\wp(\{\text{ok}, \text{er}\} \times \Sigma)$, but it is also tacitly assumed that error states are preserved by all transfer functions, i.e., that $\llbracket r \rrbracket(\text{er} : \sigma) = \text{er} : \sigma$, for any $r \in \text{RCmd}$ and $\sigma \in \Sigma$. This way, we accommodate for both pre and post that are tagged, but the treatment of tags is transparent to the rules of the logic. Therefore, in this extended setting, when we write P and Q we leave implicit that they may contain both kinds of tagged states. Moreover, this allows for a smoother treatment of reversed programs, whose execution can lead to *ok* states from *er* states, contrary to the original presentation of IL where this is never allowed. The distinction between successful and erroneous outputs will be made explicit in the analysis presented in Section 5.6.*

3.4 Necessary Conditions

Cousot et al. [2013, 2011] introduced the notion of Necessary Conditions (NC) for contract inference. Recently, the same concept has been applied to the context of security [Mackay et al. 2022]. The goal was to relax the burden on programmers by preventing the invocation of a function with arguments that will *inevitably* lead to some error. Given a correctness post Q , the NC triple $(P) r (Q)$ means that any state $\sigma \in P$ admits at least one non-erroneous execution of the program r , i.e., an execution that either terminates in Q or does not terminate at all. In other words, a necessary precondition *rules out no good run*: if violated by the initial state, the program has only erroneous executions.

Example 3.10. Consider r of Fig. 3 and the correctness post $Q_{91} \triangleq (y = 91)$. An input state $\sigma \in Q_{91}$ always leads to the post Q_{91} . Instead, if $\sigma(y) \leq 100 \wedge \sigma(y) \neq 91$ we can have both correct derivations and wrong ones, leading to output states that satisfy $\neg Q_{91}$. Finally, if $\sigma(y) > 100$ we will surely reach an error state satisfying $\neg Q_{91}$. Thus, e.g., $P = (y \leq 100)$ is a necessary precondition for Q_{91} , while $(y < 100)$ is not, because it excludes some good runs. \square

The validity of HL and IL correspond to $\llbracket r \rrbracket P \subseteq Q$ and $\llbracket r \rrbracket P \supseteq Q$, respectively, which differ for the direction (over / under) of approximation. We can distil a similar inequality for NC by noting that necessary preconditions over-approximate the backward semantics, as outlined in Fig. 2 (Zhang and Kaminski [2022, §6.3] state a similar observation in terms of weakest precondition).

PROPOSITION 3.11 (NC AS BACKWARD OVER-APPROXIMATION). *Given a correctness postcondition Q for the program r , any possible necessary precondition P for Q satisfies:*

$$\llbracket \overleftarrow{r} \rrbracket Q \subseteq P. \quad (\text{NC})$$

In general, a necessary precondition for a given post Q has no relationship with $\mathbf{wlp}[r](Q)$. However, if we switch the concept of “erroneous” and “correct” executions by considering the post $\neg Q$ instead of Q , σ admits some non-erroneous executions iff $\sigma \in \mathbf{wlp}[r](\neg Q)$, from which we derive $\neg P \subseteq \mathbf{wlp}[r](\neg Q)$. This establishes a strong connection between (NC) and (HL), namely that a necessary precondition is just the negation of a sufficient precondition for the negated post. This was already observed in Zhang and Kaminski [2022, Th. 5.4] and Cousot [2024, §I.3.14.1].

PROPOSITION 3.12 (BIJECTION BETWEEN NC AND HL). *For any $r \in \text{RCmd}$ and $P, Q \subseteq \Sigma$ we have that the HL triple $\{P\} r \{Q\}$ is valid iff the NC triple $(\neg P) r (\neg Q)$ is valid, i.e.:*

$$\llbracket r \rrbracket P \subseteq Q \iff \llbracket \overleftarrow{r} \rrbracket (\neg Q) \subseteq \neg P.$$

Example 3.13 (NC vs HL). We build on Example 3.6. Consider the correctness specification $\neg Q_{91} \triangleq (y \neq 91)$. If initially $y \neq 91$, x can be assigned a value different from 1 and y is not updated, therefore $\mathbf{wlp}[r](\neg Q_{91}) = \mathbf{wlp}[r](Q_{91}) = Q_{91}$. Thus, a condition P is necessary iff it is implied by $\neg \mathbf{wlp}[r](\neg Q_{91}) = \neg Q_{91}$. For instance, $(y \neq 91 \vee x = 2)$ is necessary, while $(y > 100)$ is not. \square

4 Sufficient Incorrectness Logic

The main motivation for studying SIL is to enhance program analysis frameworks with the ability to identify the source of incorrectness and provide programmers with evidence of some inputs that can cause the errors. This is different from IL, which can determine the presence of bugs, but cannot precisely identify their sources. SIL is also different in spirit from HL and NC that aim, respectively, to prove the absence of bugs or prevent them. Roughly, the SIL triple $\langle P \rangle r \langle Q \rangle$ asserts that “all states in P have at least one run leading to a state in Q ”, which amounts to the validity condition

$$\llbracket \overleftarrow{r} \rrbracket Q \supseteq P. \quad (\text{SIL})$$

Note that, in the presence of nondeterminism, states in P are required to have one execution leading to Q , not necessarily all of them. We make this equivalence formal with the following proposition.

Rule	SIL	IL	HL
atom	$\overline{\langle\langle \overline{\tau} \rangle\rangle c \langle Q \rangle}$	$\overline{[P] c \llbracket [c] P \rrbracket}$	$\overline{\{P\} c \{\llbracket [c] P \rrbracket\}}$
cons	$\frac{P \subseteq P' \quad \langle P' \rangle r \langle Q' \rangle \quad Q' \subseteq Q}{\langle P \rangle r \langle Q \rangle}$	$\frac{P \supseteq P' \quad [P'] r [Q'] \quad Q' \supseteq Q}{[P] r [Q]}$	$\frac{P \subseteq P' \quad \{P'\} r \{Q'\} \quad Q' \subseteq Q}{\{P\} r \{Q\}}$
seq	$\frac{\langle P \rangle r_1 \langle R \rangle \quad \langle R \rangle r_2 \langle Q \rangle}{\langle P \rangle r_{1; r_2} \langle Q \rangle}$	$\frac{[P] r_1 [R] \quad [R] r_2 [Q]}{[P] r_{1; r_2} [Q]}$	$\frac{\{P\} r_1 \{R\} \quad \{R\} r_2 \{Q\}}{\{P\} r_{1; r_2} \{Q\}}$
choice	$\frac{\forall i \in \{1, 2\} \quad \langle P_i \rangle r_i \langle Q \rangle}{\langle P_1 \cup P_2 \rangle r_1 \boxplus r_2 \langle Q \rangle}$	$\frac{\forall i \in \{1, 2\} \quad [P] r_i [Q_i]}{[P] r_1 \boxplus r_2 [Q_1 \cup Q_2]}$	$\frac{\forall i \in \{1, 2\} \quad \{P\} r_i \{Q\}}{\{P\} r_1 \boxplus r_2 \{Q\}}$
iter	$\frac{\forall n \geq 0. \langle Q_{n+1} \rangle r \langle Q_n \rangle}{\langle \bigcup_{n \geq 0} Q_n \rangle r^* \langle Q_0 \rangle}$	$\frac{\forall n \geq 0. [P_n] r [P_{n+1}]}{[P_0] r^* [\bigcup_{n \geq 0} P_n]}$	$\frac{\{P\} r \{P\}}{\{P\} r^* \{P\}}$
empty	$\overline{\langle \emptyset \rangle r \langle Q \rangle}$	$\overline{[P] r [\emptyset]}$	$\overline{\{\emptyset\} r \{Q\}}$
disj	$\frac{\langle P_1 \rangle r \langle Q_1 \rangle \quad \langle P_2 \rangle r \langle Q_2 \rangle}{\langle P_1 \cup P_2 \rangle r \langle Q_1 \cup Q_2 \rangle}$	$\frac{[P_1] r [Q_1] \quad [P_2] r [Q_2]}{[P_1 \cup P_2] r [Q_1 \cup Q_2]}$	$\frac{\{P_1\} r \{Q_1\} \quad \{P_2\} r \{Q_2\}}{\{P_1 \cup P_2\} r \{Q_1 \cup Q_2\}}$
iter0	$\overline{\langle Q \rangle r^* \langle Q \rangle}$	$\overline{[P] r^* [P]}$	unsound
unroll	$\frac{\langle P \rangle r^*; r \langle Q \rangle}{\langle P \rangle r^* \langle Q \rangle}$	$\frac{[P] r^*; r [Q]}{[P] r^* [Q]}$	unsound
conj	unsound	unsound	$\frac{\{P_1\} r \{Q_1\} \quad \{P_2\} r \{Q_2\}}{\{P_1 \cap P_2\} r \{Q_1 \cap Q_2\}}$

Fig. 4. Rules of SIL, IL and HL. HL/IL rules are adapted from Möller et al. [2021]. Identical rules are in purple. Minimal sound and complete sets of rules are in the top quadrant, while auxiliary rules are in the bottom one.

PROPOSITION 4.1 (CHARACTERIZATION OF SIL VALIDITY). *For any $r \in \text{RCmd}$, $P, Q \subseteq \Sigma$ we have*

$$\llbracket \overline{\tau} \rrbracket Q \supseteq P \iff \forall \sigma \in P. \exists \sigma' \in Q. \sigma' \in \llbracket r \rrbracket \sigma$$

According to Prop. 4.1, we can fit SIL in the quadrant of Fig. 2: it is opposite to NC w.r.t. the direction of approximation and opposite to IL w.r.t. the direction of analysis.

By the correspondence between $\llbracket \overline{\tau} \rrbracket$ and $\mathbf{wpp}[r]$ drawn in Section 3.1, it follows that SIL validity condition coincides with an under-approximation of the possible correctness introduced by Hoare, in the sense that $\llbracket \overline{\tau} \rrbracket Q \supseteq P \iff P \subseteq \mathbf{wpp}[r](Q)$. However, Hoare did not discuss under-approximation, and SIL differs from the corresponding calculus of predicate transformers because it allows to introduce approximations at any step of deduction, e.g., by unrolling a loop, without having to compute the exact weakest possible precondition. The objectives of SIL and \mathbf{wpp} are also different: identifying sources of incorrectness vs ensuring some correct outcome is reachable.

A convenient way to exploit SIL is to assume that the analysis takes as input the incorrectness specification Q , which represents the set of erroneous final states. Such post can, e.g., be determined by exploiting IL to detect reachable error states. Then, any valid SIL triple $\langle P \rangle r \langle Q \rangle$ yields a precondition which surely captures erroneous executions.

4.1 Proof System

We present the inference rules for SIL in the 1st column of Fig. 4. Note that all the rules can be applied to an arbitrary post Q to infer a corresponding pre, in the same way as all the rules of

IL can be applied to an arbitrary pre P to infer a corresponding post. This is a key feature of SIL proof system, which facilitates backward reasoning. The five rules in the top quadrant form a minimal, sound and complete proof system. The remaining four rules are auxiliary ones that can ease program analysis and are discussed later.

A Minimal, Sound and Complete Proof System. Rules $\langle\langle\text{atom}\rangle\rangle$ of HL and IL exploit the forward semantics, while SIL rule $\langle\langle\text{atom}\rangle\rangle$ uses the backward one, matching the direction of the analysis and summarizing cases for skip, assignments and Boolean guards. In Sections 4.2 and 5.3 we account for the instances of $\langle\langle\text{atom}\rangle\rangle$ using pre / post defined by formulae instead of sets.

The consequence rule is key in all the logics because it allows to generalize a proof by weakening/strengthening the two conditions P and Q involved. It can readily be derived from the validity condition of each logic: the direction of rules $\langle\langle\text{cons}\rangle\rangle$ of SIL and $\{\text{cons}\}$ of HL is the same and it is exactly the opposite of rule $[\text{cons}]$ of IL and NC. So the consequence rules follow the diagonals of Fig. 2. Moreover, $\langle\langle\text{cons}\rangle\rangle$ allows SIL to drop disjuncts in the pre, just as $[\text{cons}]$ allows IL do it in the post. This feature is crucial in both SIL and IL to increase scalability of tools.

Rule $\langle\langle\text{seq}\rangle\rangle$ is standard: SIL triples are composed sequentially just like in all other logics.

SIL rule $\langle\langle\text{choice}\rangle\rangle$ is symmetric to IL rule $[\text{choice}]$. It states that if all states in P_1 (resp. P_2) have an execution of r_1 (resp. r_2) ending in Q , they also have an execution of $r_1 \boxplus r_2$ ending in Q since the semantics of $r_1 \boxplus r_2$ is a superset of that of r_1 (resp. r_2), cf. (2). Rule $\langle\langle\text{choice}\rangle\rangle$ is reminiscent of the equation for conditionals in the calculus of possible correctness [Hoare 1978].

Rule $\{\text{iter}\}$ of HL allows to use any invariant, not necessarily the minimal one, because HL relies on over-approximation. Instead, rules $\langle\langle\text{iter}\rangle\rangle$ and $[\text{iter}]$ compute under-approximations so they are based on finite unrolling. But while $[\text{iter}]$ is designed for forward inference, the direction of $\langle\langle\text{iter}\rangle\rangle$ is clearly backward. Indeed, the precondition for r^* is the union of the preconditions of exactly n iteration steps backward. We remark that a similar rule first appeared in Möller et al. [2021, §5].

These five rules form a sound and complete proof system for SIL. Soundness can be proved by straightforward induction on derivation trees. To prove completeness, we rely on the fact that rules other than $\langle\langle\text{cons}\rangle\rangle$ are exact, that is, if their premises satisfy the equality $\llbracket\langle\langle\text{r}\rangle\rangle Q\rrbracket = P$, their conclusion does as well. Using this, we can derive the triple $\langle\langle\llbracket\langle\langle\text{r}\rangle\rangle Q\rrbracket r \langle\langle Q \rangle\rangle$ for any r and Q . Then we conclude using $\langle\langle\text{cons}\rangle\rangle$ to get a proof of $\langle\langle P \rangle\rangle r \langle\langle Q \rangle\rangle$ for any $P \subseteq \llbracket\langle\langle\text{r}\rangle\rangle Q\rrbracket$.

THEOREM 4.2 (SIL IS SOUND AND COMPLETE). *A SIL triple is provable using the rules in Fig. 4 (1st column, top quadrant) if and only if it is valid.*

Additional Rules for Program Analysis. The set of topmost five rules in Fig. 4 is deliberately minimal: if we remove any rule it is no longer complete. Of course, many other sound rules can be useful in practice but cannot be derived from those five: some are reported in Fig. 4, at the bottom.

Rule $\langle\langle\text{empty}\rangle\rangle$ is the same as $\{\text{empty}\}$ and it is used to drop paths backward, just like IL can drop them forward. Particularly, this allows to ignore one of the branches of $\langle\langle\text{choice}\rangle\rangle$, or to stop the backward iteration of $\langle\langle\text{iter}\rangle\rangle$ without covering all the iterations. An example of such an application is the derived rule $\langle\langle\text{iter0}\rangle\rangle$, which corresponds to not entering the iteration at all. It can be proved from rules $\langle\langle\text{iter}\rangle\rangle$ and $\langle\langle\text{empty}\rangle\rangle$ by taking $Q_0 = Q$ and $Q_n = \emptyset$ for $n \geq 1$. It subsumes HL's rule $\{\text{iter}\}$, which is based on loop invariants: those are a correct but not complete reasoning tool for under-approximation [O'Hearn 2020]. Rules iter0 and unroll are a prerogative of under-approximation: they are the same for SIL and IL, but they are unsound for HL. Rule unroll allows to unroll a loop once. Subsequent applications of this rule allow to simulate (backward in SIL, forward in IL) a finite number of iterations, and then rule iter0 can be used to ignore the remaining ones. As already observed for IL [O'Hearn 2020], finite unrolling is a sound under-approximation technique.

Example 4.3. Let us consider the program “loop0” from O'Hearn [2020, §6.1]:

```
x:=0; n:=nondet(); while (n>0) { x:=x+n; n:=nondet(); } // assert(x != 2000000)
```

In the syntax of regular commands, we let $\text{rloop0} \triangleq x:=0; n:=\text{nondet}(); (r_w)^*; (n \leq 0)?$, where $r_w \triangleq (n > 0)?; x:=x+n; n:=\text{nondet}()$. Final error states are those in $Q_{2M} \triangleq (x = 2000000)$. To prove a triple for rloop0 , we have to perform at least one iteration, and we do so using $\langle\langle \text{unroll} \rangle\rangle$. We let $R_{2M} \triangleq (x = 2000000 \wedge n \leq 0)$ and $T_{2M} \triangleq (x + n = 2000000 \wedge n > 0)$. It is straightforward to prove $\langle\langle T_{2M} \rangle\rangle r_w \langle\langle R_{2M} \rangle\rangle$ via $\langle\langle \text{seq} \rangle\rangle$ and $\langle\langle \text{atom} \rangle\rangle$. Given this triple, we can unroll the loop once using $\langle\langle \text{unroll} \rangle\rangle$ and ignore other iterations with $\langle\langle \text{iter0} \rangle\rangle$, proving the same triple for r_w^* : $\langle\langle T_{2M} \rangle\rangle r_w^* \langle\langle R_{2M} \rangle\rangle$. This is a property of under-approximation: a non-deterministic number of iterations can be under-approximated by a single iteration [O’Hearn 2020, §6.1]. With this triple for the loop, we can prove for the whole program the triple $\langle\langle \text{true} \rangle\rangle \text{rloop0} \langle\langle Q_{2M} \rangle\rangle$ using $\langle\langle \text{seq} \rangle\rangle$ and $\langle\langle \text{atom} \rangle\rangle$. Differently than IL, this triple highlights that any initial state can lead to an error: instead of just reporting the presence of a bug, we can report some sources of the bug. \square

Rule $\langle\langle \text{disj} \rangle\rangle$ allows to split the analysis and join the results, just like HL and IL. The validity of such a rule relies on the fact that collecting semantics (both forward and backward) are additive. However they are not co-additive, so the corresponding rule $\langle\langle \text{conj} \rangle\rangle$ which perform intersection of pre/post pairs is sound for HL but not for IL and SIL. We discuss this point further in Section 4.2.

4.2 Comparison Between Program Logics

In this section we enhance the informal relation between SIL and other program logics drawn in the introduction, providing additional technical insights. We start with a point-wise comparison between SIL and HL, IL, NC and OL and then focus on the existence of weakest/strongest conditions and on the relation between termination and reachability aspects. Some of these relations were independently discovered and discussed in Verscht and Kaminski [2025] and in Zhang et al. [2024, §5.4], albeit formulated quite differently.

SIL vs HL. In general, HL and SIL address different goals. However, they coincide whenever the program r is deterministic and terminates for every input. This is neatly spelled out below in terms of their validity conditions.

PROPOSITION 4.4 (SIL vs HL). *For any $r \in \text{RCmd}$ and $P, Q \subseteq \Sigma$ we have:*

- if r is deterministic, $\llbracket \overleftarrow{r} \rrbracket Q \supseteq P \implies \llbracket r \rrbracket P \subseteq Q$ (i.e., SIL implies HL),
- if r is terminating, $\llbracket r \rrbracket P \subseteq Q \implies \llbracket \overleftarrow{r} \rrbracket Q \supseteq P$ (i.e., HL implies SIL).

Example 4.5. From Example 3.6, we know that the HL triple $\{y > 100\} r \{-Q_{91}\}$ is valid for the program r of Fig. 3. Moreover, it is known that McCarthy’s 91 function always terminates for every input, so that r is terminating. Therefore, according to Proposition 4.4 $\{y > 100\} r \{-Q_{91}\}$ is a valid SIL triple. Indeed, whenever $y > 100$ at the beginning, the execution does not enter the if statement, so the value of y is unchanged in the output. \square

SIL and IL vs NC. It follows immediately from the validity conditions of (SIL) and (NC) that the only triples they have in common are the exact ones, similarly to (IL) and (HL). However, this similarity sparks another question: just like (SIL) and (HL) (which forms one of diagonals of Fig. 2) are related under some program hypothesis, the same may be true for (IL) and (NC). It turns out that whenever r is reversible (i.e., $\llbracket r \rrbracket$ is injective) any valid IL triple is also a valid NC triple. This fact, as well as the difference between (NC) and (IL), is easier to see on quantified definitions:

$$\forall \sigma' \in Q. \forall \sigma \in \llbracket \overleftarrow{r} \rrbracket \sigma'. \sigma \in P \quad (\text{NC}^\forall) \qquad \forall \sigma' \in Q. \exists \sigma \in \llbracket \overleftarrow{r} \rrbracket \sigma'. \sigma \in P \quad (\text{IL}^\exists)$$

While (NC^\forall) universally quantifies on initial states, (IL^\exists) existentially quantifies on them. However, in general NC and IL are not related:

Example 4.6 (NC vs IL). Consider once again the program r of Fig. 3, with $Q_{91} \triangleq (y = 91)$. In Table 1, we already observed that $(y \leq 100) \ r \ (Q_{91})$ was a valid NC triple but that $[y \leq 100] \ r \ [Q_{91}]$ was not valid in IL. On the other hand, $[y = 100] \ r \ [Q_{91} \wedge x = 0]$ is a valid IL triple but it is not a valid NC triple. This is because the pre in NC have to include all the inputs leading to the post. \square

SIL vs IL. While Proposition 3.12 shows that over-approximation logics (HL) and (NC) are isomorphic through the use of negation, the next counterexample shows that the connection does not extend to under-approximation logics (IL) and (SIL).

Example 4.7. Since IL and SIL enjoy different consequence rules, neither of the two can imply the other under the same pre and post. For the use of negation, consider the assignment $r1 \triangleq x := 1$ and the properties $P_{\geq 0} \triangleq (x \geq 0)$ and $Q_1 \triangleq (x = 1)$. Both the SIL triple $\langle\langle P_{\geq 0} \rangle\rangle \ r1 \ \langle\langle Q_1 \rangle\rangle$ and the IL triple $[P_{\geq 0}] \ r1 \ [Q_1]$ are valid, but neither $[-P_{\geq 0}] \ r1 \ [-Q_1]$ nor $\langle\langle -P_{\geq 0} \rangle\rangle \ r1 \ \langle\langle -Q_1 \rangle\rangle$ are valid. \square

To gain some insights on why (SIL) and (IL) are not equivalent through negation, given a regular command r we define the set of states that only diverges D_r and the set of unreachable states U_r :

$$D_r \triangleq \{\sigma \mid \llbracket r \rrbracket \sigma = \emptyset\} \quad U_r \triangleq \{\sigma' \mid \sigma' \notin \llbracket r \rrbracket \Sigma\} = \{\sigma' \mid \llbracket \overleftarrow{r} \rrbracket \sigma' = \emptyset\}.$$

These two sets are dual w.r.t. the direction of execution: U_r is the set of states which “diverge” going backward. In fact, if we consider $\llbracket \overleftarrow{r} \rrbracket$ instead of $\llbracket r \rrbracket$, the roles of D_r and U_r are swapped.

LEMMA 4.8. *For any regular command $r \in \text{RCmd}$ and sets of states $P, Q \subseteq \Sigma$ it holds that*

$$\llbracket \overleftarrow{r} \rrbracket \llbracket r \rrbracket P \supseteq P \setminus D_r \quad \text{and} \quad \llbracket r \rrbracket \llbracket \overleftarrow{r} \rrbracket Q \supseteq Q \setminus U_r.$$

Lemma 4.8 showcases the asymmetry between over and under-approximation: the composition of a function with its inverse is increasing (but for non-terminating states). This hints at why (HL) and (NC) are related while (IL) and (SIL) are not: on the over-approximating side, $P \setminus D_r \subseteq \llbracket \overleftarrow{r} \rrbracket \llbracket r \rrbracket P$ can be further exploited if we know $\llbracket r \rrbracket P \subseteq Q$ via (HL), but it cannot when $\llbracket r \rrbracket P \supseteq Q$ via (IL).

Rather than using negation, a correspondence between validity conditions of SIL and IL for a program r can be drawn by exploiting the properties of opposite relations, through the use of a “reversed” program $(r)^{\text{op}}$, when expressible. In fact, since forward and backward semantics are defined as opposite relations, i.e., $\llbracket r \rrbracket^{\text{op}} = \llbracket \overleftarrow{r} \rrbracket$, one can rephrase SIL validity condition $\llbracket \overleftarrow{r} \rrbracket Q \supseteq P$ by writing $\llbracket r \rrbracket^{\text{op}} Q \supseteq P$. This implies that whenever $(r)^{\text{op}}$ such that $\llbracket (r)^{\text{op}} \rrbracket = \llbracket r \rrbracket^{\text{op}}$ is expressible in the language, we get an isomorphism between SIL triples $\langle\langle P \rangle\rangle \ r \ \langle\langle Q \rangle\rangle$ and IL triples $[Q] \ (r)^{\text{op}} \ [P]$. HL triples $\{P\} \ r \ \{Q\}$ and NC triples $(Q) \ (r)^{\text{op}} \ (P)$ are related in the same way.

LEMMA 4.9. *For any $P, Q \subseteq \Sigma$ and $r \in \text{RCmd}$ such that $(r)^{\text{op}} \in \text{RCmd}$, it holds:*

$$\llbracket \overleftarrow{r} \rrbracket Q \supseteq P \iff \llbracket (r)^{\text{op}} \rrbracket Q \supseteq P.$$

In retrospect, the above isomorphism and the compositionality properties stated in Remark 3.2 suggest a method to distill sound SIL proof rules from IL ones (or vice versa). This is the case for all rules in Fig. 4 except for $\langle\langle \text{unroll} \rangle\rangle$, whose premise should involve the opposite command $r; r^*$ instead of (the equivalent) $r^*; r$, and for $\langle\langle \text{atom} \rangle\rangle$, because the opposite of an atomic command is not necessarily available, as in the case of assignments (see Remark 3.2). This means SIL rule for assignment cannot be derived from the IL one. More in detail, let us consider two well known HL axioms: Hoare’s backward substitution [Hoare 1969] and Floyd’s forward inference [Floyd 1967]:

$$\frac{}{\{q[a/x]\} \ x \ := \ a \ \{q\}} \ \{\text{Hoare}\} \quad \frac{}{\{p\} \ x \ := \ a \ \{\exists x'. p[x'/x] \wedge x = a[x'/x]\}} \ \{\text{Floyd}\}$$

where $q[a/x]$ denotes the capture-avoiding substitution of all free occurrences of x in q with a . Floyd’s forward axiom is also valid in IL, but Hoare’s axiom is not [O’Hearn 2020, §4]. In contrast, both Hoare’s backward and Floyd’s forward axiom are valid in SIL. Moreover, this witnesses that

forward and backward under-approximation analysis behave differently. This is possibly rooted in the properties of arithmetic expressions: they are defined for every input but not necessarily surjective. In the terminology of Lemma 4.8 above, $D_{x:=a}$ is always empty but $U_{x:=a}$ may not be.

Furthermore, some extra care is necessary when handling IL rules with distinct termination modalities, like **ok** and **er**, because they are present only in postconditions, under the assumption that $\llbracket r \rrbracket(er : \sigma) = er : \sigma$ for any command r . In general, this property does not hold for the opposite relation: $\llbracket r \rrbracket^{op}(er : \sigma)$ may not contain $er : \sigma$ only, and therefore the IL treatment of reversed programs would require new, ad hoc technical developments.

SIL vs OL. The OL framework is parametric in a monoid of outcomes, which makes it applicable to a variety of contexts, including probabilistic systems. We compare SIL with OL nondeterministic instance based on the powerset monoid, denoted OL_{nd} . As outlined in the Introduction, OL already recognized the importance of locating the source of errors and can encode Lisbon triples: any OL_{nd} triple of the form⁵ $\langle P \rangle r \langle Q \oplus \top \rangle$ is a valid Lisbon triple $\langle\langle P \rangle\rangle r \langle\langle Q \rangle\rangle$. The correspondence between OL_{nd} and Lisbon/SIL triples has already been discussed in Zilberstein et al. [2023]. Although OL_{nd} and SIL share similar goals, we highlight below some key differences in their usage.

First, the OL rules presented in Zilberstein et al. [2023] are mostly oriented for forward analysis, while SIL rules are explicitly tailored to backward analysis (see Section 5.9).

Second, we observe that OL_{nd} uses a higher-level assertions language which describes sets of subsets of Σ , whereas SIL assertions are just subsets of Σ . This is a direct consequence of the generality of OL framework. In the jargon of OL, SIL assertions are called *atomic assertions*, which are then combined with higher level operators to denote sets of sets of states. Take for instance the atomic assertion $P_x \triangleq (x = 0)$. In SIL, P_x is satisfied by states σ such that $\sigma(x) = 0$, while in OL_{nd} P_x is satisfied by *sets of states* m such that $\forall \sigma \in m. \sigma(x) = 0$. Consequently, two kinds of disjunctive predicates coexist in OL, named \vee and \oplus . Neither of them correspond directly to the union \cup used in SIL, even though the equivalence $P_1 \cup P_2 = P_1 \vee P_2 \vee (P_1 \oplus P_2)$ holds. Higher-level assertions make the application of OL_{nd} rules to prove Lisbon triples sometimes more involved, possibly with the need to derive specialized rules (such as, for instance, those presented in the unpublished manuscript Zilberstein [2024]).

Third, OL is extremely general: its rules accommodate for all possible execution models and can thus be reused for, e.g., probabilistic programs. On the contrary, SIL is crafted specifically for nondeterminism, making its rules smoother and compact and their application straight and syntax-directed. While the difference is often negligible, there are some cases where it becomes pivotal. We refer the reader to Section 5.9 for an example that shows how the model underlying SIL can provide a more precise analysis, because it addresses the nuances of nondeterministic systems.

Weakest/Strongest Conditions. Depending on the way in which program analysis is conducted, one can be interested in deriving either the most general or most specific hypotheses under which certain phenomena can take place. For instance, given a correctness specification Q , the weakest liberal preconditions for Q satisfied fixes the minimal constraint on the input that guarantees program correctness. Conversely, to infer necessary conditions we can be interested in devising the strongest hypotheses under which some correct run is possible.

To investigate the existence of weakest/strongest pre and post, we take into account the consequence rules of the four kinds of triples, which explains how pre and postconditions can be weakened/strengthened. The concrete semantics is trivially a strongest (HL and NC) or weakest (IL and SIL) condition for the “target” property (P computing backward and Q forward). It turns out that having a strongest/weakest condition on the “source” property is a prerogative of over-approximation, i.e., that over and under-approximation behave quite differently in this respect.

⁵Where \oplus is the monoid composition and \top represents the trivial property satisfied by any outcome.

PROPOSITION 4.10 (WEAKEST AND STRONGEST CONDITIONS). *For any command $r \in \text{RCmd}$:*

- *for every Q , there exists a weakest P such that $\llbracket r \rrbracket P \subseteq Q$ (weakest pre for HL);*
- *for every P , there exists a weakest Q such that $\llbracket \overleftarrow{r} \rrbracket Q \subseteq P$ (weakest post for NC);*
- *for some Q , there is no strongest P such that $\llbracket r \rrbracket P \supseteq Q$ (no strongest pre for IL);*
- *for some P , there is no strongest Q such that $\llbracket \overleftarrow{r} \rrbracket Q \supseteq P$ (no strongest post for SIL).*

The reason why strongest conditions may not exist for IL and SIL is that the collecting semantics (both forward and backward) is additive but not co-additive. In other words, rule {disj} is sound for all triples, while rule {conj} is valid for HL and NC but neither for IL nor SIL, as shown in Fig. 4. This means that given two IL triples $[P_1] r [Q]$ and $[P_2] r [Q]$, in general $\llbracket r \rrbracket (P_1 \cap P_2) \not\supseteq Q$ in which case $[P_1 \cap P_2] r [Q]$ is not valid, as shown in the following example.

Example 4.11. Consider the program r of Fig. 3. We can prove the two IL triples $[y = 93] r [Q_{91} \wedge x = 0]$ and $[y = 94] r [Q_{91} \wedge x = 0]$, but their “conjunctive triple” $[\emptyset] r [Q_{91} \wedge x = 0]$ is no longer a valid IL triple. Similarly for SIL, we can prove both $\langle\langle y = 93 \rangle\rangle r \langle\langle y = 91 \rangle\rangle$ and $\langle\langle y = 93 \rangle\rangle r \langle\langle y = 93 \rangle\rangle$, and both are minimal because \emptyset would not be a valid post for the pre ($y = 93$). \square

This can also be observed using the theory of adjunction. It is well known that left adjoints are additive while right adjoints are co-additive [Davey and Priestley 2002]. The weakest precondition \mathbf{wlp} for HL is characterized by the adjunctive property $P \subseteq \mathbf{wlp}[r](Q)$ iff $\llbracket r \rrbracket P \subseteq Q$ (weakest postconditions for NC are defined analogously). Since the forward (resp. backward) semantics is additive we get the existence of its right adjoint, that is exactly HL weakest precondition (resp. NC weakest postcondition). However, a strongest precondition \mathbf{sp} for IL would satisfy the adjunctive property $\mathbf{sp}[r](Q) \subseteq P$ iff $Q \subseteq \llbracket r \rrbracket P$, making the non co-additive forward semantics a right adjoint. Similarly, a strongest postcondition for SIL would be a left adjoint of the backward semantics.

Termination and Reachability. Termination and reachability are two sides of the same coin when switching from forward to backward reasoning, but over- and under-approximation behave quite differently in their respect.

For HL, given the definition of collecting semantics, we can only set apart a precondition which always causes divergence: if Q is empty, all states in the precondition must always diverge. However, if just one state in P has one terminating computation, its final state must be in Q , so we do not know whether states in P diverge or not. Moreover, because of the over-approximation, a non empty Q does not mean there truly are finite executions, as those may be introduced by the approximation. Dually, NC cannot say much about reachability of states in Q unless P is empty, in which case all states in Q are clearly unreachable.

On the contrary, under-approximation has much stronger guarantees on divergence/reachability. Any IL triple $[P] r [Q]$ ensures that all states in Q are reachable (in particular, from states in P). Conversely, a SIL triple $\langle\langle P \rangle\rangle r \langle\langle Q \rangle\rangle$ means that all states in P have a convergent computation (which ends in a state in Q). This motivates the choice of a forward (resp. backward) rule for iteration in IL (resp. SIL): a backward (resp. forward) rule would need to prove reachability (resp. termination) of all points in the postcondition (resp. precondition). Instead, the forward rule of IL (resp. backward rule of SIL) ensures reachability (resp. termination) by construction, as it builds Q (resp. P) only with points which are known to be reachable (resp. terminating) by executing the loop.

5 Separation Sufficient Incorrectness Logic

We instantiate SIL to handle pointers and dynamic memory allocation, introducing Separation SIL, which constitutes our main technical contribution. The goal of Separation SIL is to identify the causes of memory errors: it combines the backward under-approximation principles of SIL with the ability to deal with pointers from Separation Logic (SL) [O’Hearn et al. 2001; Reynolds 2002].

5.1 Heap Regular Commands

We denote by HRCmd the set of all heap (regular) commands that is obtained by plugging in (1) the following definition of heap atomic commands HACmd in place of ACmd :

$$\text{HACmd} \ni c ::= \text{skip} \mid x := a \mid b? \mid x := \text{alloc}() \mid \text{free}(x) \mid x := [y] \mid [x] := y$$

where, without loss of generality, we assume that x and y are syntactically distinct variables. The new primitives are highlighted in blue. The primitive $\text{alloc}()$ allocates a new memory location containing a nondeterministic value, free deallocates memory and $[\cdot]$ is the dereferencing operator. The syntax only allows to allocate, free and dereference single variables. To use a value from the heap in a compound expression, it must be loaded in a variable beforehand.

Given a heap command $r \in \text{HRCmd}$, we let $\text{fv}(r) \subseteq \text{Var}$ as the set of (free) variables appearing in r . We let $\text{mod}(r) \subseteq \text{Var}$ be the set of variables modified by r ; the definition is standard but we remark that $\text{free}(x)$ and $[x] := y$ do not modify x : this is because they only modify the value pointed by x , not the actual value of x (the memory address itself).

5.2 Assertion Language

Our assertion language is derived from both SL and Incorrectness Separation Logic (ISL):

$$\text{Asl} \ni p, q ::= \text{false} \mid \text{true} \mid p \wedge q \mid p \vee q \mid \exists x. p \mid a \asymp a \mid \mathbf{emp} \mid x \mapsto a \mid x \not\mapsto \mid p * q$$

where $\asymp \in \{=, \neq, \leq, <, \dots\}$ replaces standard comparison operators, $x \in \text{Var}$ is a variable and $a \in \text{AExp}$ is an arithmetic expression. The first six constructs describe coherent logic [Bezemer and Coquand 2005], a fragment of first-order logic. The others describe heaps and come from Separation Logic, with the exception of $x \not\mapsto$, which was introduced by Raad et al. [2020] in ISL. The constant \mathbf{emp} denotes an empty heap. The assertion $x \mapsto a$ represents an heap with a single memory cell pointed by x and whose content is a , while $x \not\mapsto$ describes that x points to a deallocated memory cell. The separating conjunction $p * q$ describes a heap which can be divided in two disjoint sub-heaps, one satisfying p and the other q . We let $x \mapsto - \triangleq \exists v. x \mapsto v$ describe that x is allocated but we do not care about its exact value. Given a formula $p \in \text{Asl}$, we call $\text{fv}(p) \subseteq \text{Var}$ the set of its free variables. Note that our assertion language lacks negation. This choice was done to enable completeness and automation of the proof system: in fact, in the presence of negation, Lemma 5.5 would not hold anymore. This Lemma is a key ingredient for both the proof of the completeness Theorem 5.7 and the algorithm described in Section 5.5. Moreover, we do not believe this choice to be restrictive for applications because our assertion language (strictly) contains the one used in bi-abduction [Berdine et al. 2006; Calcagno et al. 2009] and in SL-based tools. From this perspective, our choice to not include negation is a trade-off between expressivity of the assertion language and precision of the proof system that hits a soft spot, justified on the theoretical side by completeness and on the practical side by tools not using negation.

5.3 Proof System

We present the rules of Separation SIL in Fig. 5, where we recall that $q[a/x]$ is the capture-avoiding substitution. For the sake of presentation, we focus on rules without explicit error management (see Remark 3.9). However, the extension is straightforward: in Section 5.6 we will present the error rule for store and its use in Example 5.6.

We split the rules of Separation SIL in three groups (Fig. 5). The first group gives the rules for atomic commands $c \in \text{HACmd}$, i.e., all instances of the SIL rule $\langle\langle \text{atom} \rangle\rangle$. The second one includes rules borrowed from SL, especially the frame rule. The third one rephrases the rules of SIL in Fig. 4 in our language of assertions.

$\frac{}{\langle\mathbf{emp}\rangle \text{ skip } \langle\mathbf{emp}\rangle} \langle\text{skip}\rangle$	$\frac{}{\langle q[a/x] \rangle x := a \langle q \rangle} \langle\text{assign}\rangle$
$\frac{}{\langle\mathbf{emp}\rangle x := \text{alloc}() \langle x \mapsto v \rangle} \langle\text{alloc1}\rangle$	$\frac{}{\langle q \wedge b \rangle b? \langle q \rangle} \langle\text{assume}\rangle$
$\frac{}{\langle \beta \mapsto - \rangle x := \text{alloc}() \langle x = \beta \wedge x \mapsto v \rangle} \langle\text{alloc2}\rangle$	$\frac{}{\langle x \mapsto - \rangle \text{free}(x) \langle x \mapsto - \rangle} \langle\text{free}\rangle$
$\frac{x \notin \text{fv}(a)}{\langle y \mapsto a * q[a/x] \rangle x := [y] \langle y \mapsto a * q \rangle} \langle\text{load}\rangle$	$\frac{}{\langle x \mapsto - \rangle [x] := y \langle x \mapsto y \rangle} \langle\text{store}\rangle$
$\frac{\langle p \rangle r \langle q \rangle \quad \text{fv}(t) \cap \text{mod}(r) = \emptyset}{\langle p * t \rangle r \langle q * t \rangle} \langle\text{frame}\rangle$	$\frac{\langle p \rangle r \langle q \rangle \quad x \notin \text{fv}(r)}{\langle \exists x. p \rangle r \langle \exists x. q \rangle} \langle\text{exists}\rangle$
$\frac{p \Rightarrow p' \quad \langle p' \rangle r \langle q' \rangle \quad q' \Rightarrow q}{\langle p \rangle r \langle q \rangle} \langle\text{cons}\rangle$	$\frac{\langle p \rangle r_1 \langle t \rangle \quad \langle t \rangle r_2 \langle q \rangle}{\langle p \rangle r_1; r_2 \langle q \rangle} \langle\text{seq}\rangle$
$\frac{\langle p_1 \rangle r_1 \langle q \rangle \quad \langle p_2 \rangle r_2 \langle q \rangle}{\langle p_1 \vee p_2 \rangle r_1 \boxplus r_2 \langle q \rangle} \langle\text{choice}\rangle$	$\frac{\forall n \geq 0 \langle q(n+1) \rangle r \langle q(n) \rangle}{\langle \exists n. q(n) \rangle r^* \langle q(0) \rangle} \langle\text{iter}\rangle$
$\frac{}{\langle \text{false} \rangle r \langle q \rangle} \langle\text{empty}\rangle$	$\frac{\langle p_1 \rangle r \langle q_1 \rangle \quad \langle p_2 \rangle r \langle q_2 \rangle}{\langle p_1 \vee p_2 \rangle r \langle q_1 \vee q_2 \rangle} \langle\text{disj}\rangle$
$\frac{}{\langle q \rangle r^* \langle q \rangle} \langle\text{iter0}\rangle$	$\frac{\langle p \rangle r^*; r \langle q \rangle}{\langle p \rangle r^* \langle q \rangle} \langle\text{unroll}\rangle$

Fig. 5. Proof rules for Separation SIL. The first group replaces SIL rule $\langle\text{atom}\rangle$, the second includes rules peculiar of SL, the third includes rule from SIL, both its core set and auxiliary ones.

REMARK 5.1 (LOCALITY PRINCIPLE). *Differently from SIL, the post of Separation SIL rules for atomic commands is not a generic assertion q . We formulate rules this way inspired by the “locality principle” of separation logics: each axiom only focuses on the relevant part of the heap manipulated by the atomic command. Larger heaps are dealt with using a suitable “frame” rule. We will show in Section 5.5 how it is always possible to algorithmically rewrite any assertion q in a shape to which these rules can be applied, thus allowing for (automated) backward reasoning in Separation SIL.*

Rule $\langle\text{skip}\rangle$ does not specify anything about its pre and postconditions, because whatever is true before and after the skip can be added with $\langle\text{frame}\rangle$. Rule $\langle\text{assign}\rangle$ is Hoare’s backward assignment rule [Hoare 1969]. While also Floyd’s forward axiom [Floyd 1967] is sound for SIL (see Section 4.2), we opt for Hoare’s rule because it fits better with the backward analysis of SIL. Rule $\langle\text{assume}\rangle$ conjoins, in the precondition, the assertion b to the postcondition, because only states satisfying the Boolean guard can reach the post. Rules $\langle\text{alloc1}\rangle$ and $\langle\text{alloc2}\rangle$ allocate a new memory location for x : in the former the location is new, in the latter it reuses the previously deallocated location β . Rule $\langle\text{free}\rangle$ requires x to be allocated before freeing it. Rule $\langle\text{load}\rangle$ is similar to rule $\langle\text{assign}\rangle$, with the addition of the (disjoint) $y \mapsto a$ to make sure that y is allocated. Rule $\langle\text{store}\rangle$ requires that x is allocated, and updates the value it points to.

The rule $\langle\langle\text{exists}\rangle\rangle$ allows to “hide” local variables. The rule $\langle\langle\text{frame}\rangle\rangle$ is a hallmark of separation logics [Raad et al. 2020; Reynolds 2002]: it allows to add a frame around a derivation, plugging the proof for a small portion of a program inside a larger heap.

In the third group, the only notable difference w.r.t. Fig. 4 is rule $\langle\langle\text{iter}\rangle\rangle$. As a logical replacement for the infinite union used in SIL rule, Separation SIL uses a predicate $q(n)$ parametrized by the natural number $n \in \mathbb{N}$ and the precondition $\exists n.q(n)$ in the conclusion of the rule.

Pre and post of the rules for atomic commands are carefully crafted and are one of the distinguishing features of Separation SIL. This is because enforcing locality in the axioms of a sound and complete proof system depends on the subtle interactions between the rules and is not necessarily a property of each rule in isolation (see, e.g., Lemma 5.3, where a stronger condition than SIL validity is required). For instance, the symmetry between SIL and IL suggest that Separation SIL atomic rules can be obtained by ISL ones. To see the issues with this approach, consider the command $\text{free}(x)$ and the two ISL rules

$$\frac{}{[x \mapsto e] \text{ free}(x) [x \not\mapsto]} \text{ [free]} \quad \frac{}{[x = x' * y \not\mapsto] x := \text{alloc}() [x = y * y \mapsto -]} \text{ [alloc2]}$$

Separation SIL $\langle\langle\text{free}\rangle\rangle$ is different from ISL [free]: since $(x \mapsto e) \Rightarrow (x \mapsto -)$, each rule is stronger in its own proof system thanks to the consequence rule, while the other rule would be incomplete. Inspired by Lemma 4.9, we could also try reversing ISL [alloc2], but since free and alloc are not actually inverse operations this would yield an unsound rule.

The next example shows how to infer preconditions ensuring that a provided error can happen.

Example 5.2. Consider the motivating example of Raad et al. [2020], encoding a use-after-free bug involving C++ vector `push_back` function:

```
// program rclient                                push_back(v) {
x := *v;                                           if (nondet()) {
push_back(v);                                       free(*v);
*x := 1;                                           *v := alloc(); } }
```

We encode the above program as a regular command by letting:

$$\text{rclient} \triangleq x := [v]; (r_b \boxplus \text{skip}) \quad r_b \triangleq y := [v]; \text{free}(y); y := \text{alloc}(); [v] := y$$

Since our syntax does not include functions, we inline `push_back`. We cannot free and allocate $*v$ directly, whence the auxiliary variable y . For simplicity, we do not include the last assignment $*x := 1$ in `rclient`: whenever the postcondition $x \not\mapsto$ holds, an error occurs after `rclient`.

We prove the Separation SIL triple

$$\langle\langle v \mapsto z * z \mapsto - * \text{true} \rangle\rangle \text{rclient} \langle\langle x \not\mapsto * \text{true} \rangle\rangle$$

which ensures that *every* state in the precondition reaches the error, thus giving (many) actual witnesses for testing and debugging purposes. Moreover, this example explains how Separation SIL proof system guides the crafting of the precondition by backward analysis once the error postcondition is provided. Let us fix the following assertions:

$$p \triangleq (v \mapsto z * z \mapsto - * \text{true}), \quad q \triangleq (x \not\mapsto * \text{true}), \quad t \triangleq (v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto) * \text{true}).$$

To prove the Separation SIL triple $\langle\langle p \rangle\rangle \text{rclient} \langle\langle q \rangle\rangle$, we first prove the triple $\langle\langle t \rangle\rangle r_b \langle\langle q \rangle\rangle$, whose derivation is in Fig. 6b. Derivations are best read bottom-up: we start from the post and, for every atomic command, we find a suitable pre to apply the rule. In all cases, we strengthen the post to be able to apply the right rule: this usually means adding some constraint on the shape of the heap. Particularly, to apply the rule $\langle\langle\text{free}\rangle\rangle$ we need y to be deallocated, and this can happen in two different ways: either if $y = x$, since x is deallocated; or if y is a new name. This is captured by the

$$\begin{array}{l}
\langle\langle p : v \mapsto z * z \mapsto - * \text{true} \rangle\rangle \\
\langle\langle v \mapsto z * (z = z \vee z \not\mapsto) * z \mapsto - * \text{true} \rangle\rangle \\
x := [v] \\
\langle\langle v \mapsto z * (x = z \vee x \not\mapsto) * z \mapsto - * \text{true} \rangle\rangle \\
\langle\langle (\text{true} * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto)) \vee (x \not\mapsto * \text{true}) \rangle\rangle \\
\left(\begin{array}{l} \langle\langle t \rangle\rangle \\ y := [v]; \\ \text{free}(y); \\ y := \text{alloc}(); \\ [v] := y \\ \langle\langle q \rangle\rangle \end{array} \right) \boxplus \begin{array}{l} \langle\langle x \not\mapsto * \text{true} \rangle\rangle \\ \text{skip} \\ \langle\langle x \not\mapsto * \text{true} \rangle\rangle \end{array} \\
\langle\langle q : x \not\mapsto * \text{true} \rangle\rangle
\end{array}
\quad
\begin{array}{l}
\langle\langle t : \text{true} * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto) \rangle\rangle \\
y := [v]; \\
\langle\langle \text{true} * v \mapsto z * y \mapsto - * (x = y \vee x \not\mapsto) \rangle\rangle \\
\langle\langle \text{true} * v \mapsto - * y \mapsto - * (x = y \vee x \not\mapsto) \rangle\rangle \\
\text{free}(y); \\
\langle\langle \text{true} * v \mapsto - * \underline{y \not\mapsto} * (x = y \vee x \not\mapsto) \rangle\rangle \\
\langle\langle x \not\mapsto * v \mapsto - * \text{emp} * \text{true} \rangle\rangle \\
y := \text{alloc}(); \\
\langle\langle x \not\mapsto * v \mapsto - * \underline{y \mapsto y'} * \text{true} \rangle\rangle \\
\langle\langle x \not\mapsto * v \mapsto - * \text{true} \rangle\rangle \\
[v] := y \\
\langle\langle x \not\mapsto * v \mapsto \underline{y} * \text{true} \rangle\rangle \\
\langle\langle q : x \not\mapsto * \text{true} \rangle\rangle
\end{array}$$

(a) Linearized derivation of the Separation SIL triple $\langle\langle p \rangle\rangle$ rclient $\langle\langle q \rangle\rangle$. The omitted sub-derivation is in Fig. 6b.

(b) Linearized derivation of the Separation SIL triple $\langle\langle t \rangle\rangle$ r_b $\langle\langle q \rangle\rangle$.

Fig. 6. The full derivation of $\langle\langle p \rangle\rangle$ rclient $\langle\langle q \rangle\rangle$, split in two parts. We write in grey the strengthened conditions obtained using $\langle\langle \text{cons} \rangle\rangle$, and underline the postcondition of the rule for the current atomic command. Everything else is a frame shared between pre and post, using $\langle\langle \text{frame} \rangle\rangle$.

disjunction $x = y \vee x \not\mapsto$. For the sake of presentation, here this strengthening is done by human intervention, but we describe in Section 5.5 how it can be automated.

Using the derivation in Fig. 6b, we complete the proof as shown in Fig. 6a. In the derivation, using rule $\langle\langle \text{load} \rangle\rangle$ for the first assignment $x := [v]$, we get the pre $(v \mapsto z * z \mapsto - * (z = z \vee z \not\mapsto) * \text{true})$, but since $z \mapsto - * z \not\mapsto$ is not satisfiable we remove that disjunct and find p .

Note the use of rule $\langle\langle \text{cons} \rangle\rangle$ in the pre of the nondeterministic choice to remove the disjunct $(x \not\mapsto * \text{true})$, effectively dropping the analysis of that program path. This mirrors IL's ability to drop paths going forward. \square

In the example, we use as error postcondition $x \not\mapsto * \text{true}$. It is necessary to include $(* \text{true})$ because, in final reachable states, x is not the only variable allocated (there are also v and y), so the final heap should talk about them as well. Adding $(* \text{true})$ is a convenient way to focus only on the part of the heap that describes the error, that is $x \not\mapsto$, and just leave everything else unspecified.

5.4 Soundness

To prove soundness of Separation SIL, we give a semantic model for heap regular commands. Fixed a finite set Var of variables and an infinite set Loc of memory locations, we define the set of values as $\text{Val} \triangleq \mathbb{Z} \uplus \text{Loc}$ (\uplus is disjoint union). Stores $s \in \text{Store}$ are (total) functions $s : \text{Var} \rightarrow \text{Val}$; heaps $h \in \text{Heap}$ are partial functions $h : \text{Loc} \rightarrow \text{Val} \uplus \{\delta\}$. If $h(l) = v \in \text{Val}$, location l is allocated and holds value v , if $l \notin \text{dom}(h)$ then it is not allocated. The special value δ describes a deallocated memory location: if $h(l) = \delta$, that location was previously allocated and then deallocated. As notation, we use $s[x \mapsto v]$ for function update, $[\]$ for the empty heap and $[l \mapsto v]$ for the heap defined only on l and associating value v to it. We say two heaps are disjoint, written $h_1 \perp h_2$, when $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$, and in that case we define the \bullet operation as the merge of the two: $h_1 \bullet h_2$ coincides with h_1 on $\text{dom}(h_1)$, with h_2 on $\text{dom}(h_2)$ and it is undefined everywhere else.

$$\begin{aligned}
\llbracket x := a \rrbracket(s, h) &\triangleq \{(s[x \mapsto \llbracket a \rrbracket s], h)\} \\
\llbracket x := \text{alloc}() \rrbracket(s, h) &\triangleq \{(s[x \mapsto l], h[l \mapsto v]) \mid v \in \text{Val}, \text{avail}_h(l)\} \\
\llbracket \text{free}(x) \rrbracket(s, h) &\triangleq \{(s, h[s(x) \mapsto \delta])\} \quad \text{if } h(s(x)) \in \text{Val} \\
\llbracket x := [y] \rrbracket(s, h) &\triangleq \{(s[x \mapsto h(s(y))], h)\} \quad \text{if } h(s(y)) \in \text{Val}
\end{aligned}$$

Fig. 7. Semantics of heap atomic commands (excerpt), where $\text{avail}_h(l) \triangleq (l \notin \text{dom}(h) \vee h(l) = \delta)$. We let $\llbracket c \rrbracket \sigma \triangleq \{\mathbf{err}\}$ unless differently stated.

$$\begin{aligned}
\llbracket a_1 \asymp a_2 \rrbracket &\triangleq \{(s, h) \mid \llbracket a_1 \rrbracket s \asymp \llbracket a_2 \rrbracket s\} & \llbracket \text{false} \rrbracket &\triangleq \emptyset \\
\llbracket \exists x. p \rrbracket &\triangleq \{(s, h) \mid \exists v \in \text{Val}. (s[x \mapsto v], h) \in \llbracket p \rrbracket\} & \llbracket \text{true} \rrbracket &\triangleq \Sigma \\
\llbracket x \not\mapsto \rrbracket &\triangleq \{(s, [s(x) \mapsto \delta])\} & \llbracket p \vee q \rrbracket &\triangleq \llbracket p \rrbracket \cup \llbracket q \rrbracket \\
\llbracket x \mapsto a \rrbracket &\triangleq \{(s, [s(x) \mapsto \llbracket a \rrbracket s])\} & \llbracket p \wedge q \rrbracket &\triangleq \llbracket p \rrbracket \cap \llbracket q \rrbracket \\
\llbracket p * q \rrbracket &\triangleq \{(s, h_p \bullet h_q) \mid (s, h_p) \in \llbracket p \rrbracket, (s, h_q) \in \llbracket q \rrbracket, h_p \perp h_q\} & \llbracket \mathbf{emp} \rrbracket &\triangleq \{(s, [])\}
\end{aligned}$$

Fig. 8. Semantics of the assertion language.

Let $\Sigma = \text{Store} \times \text{Heap}$, and $\Sigma_e = \Sigma \uplus \{\mathbf{err}\}$: states $\sigma \in \Sigma_e$ are either a pair store/heap or the error state \mathbf{err} . A meaningful excerpt of the denotational semantics of heap atomic commands $\llbracket \cdot \rrbracket : \text{HACmd} \rightarrow \wp(\Sigma_e) \rightarrow \wp(\Sigma_e)$ is in Fig. 7. To ease the presentation, we define it as $\llbracket \cdot \rrbracket : \text{HACmd} \rightarrow \Sigma \rightarrow \wp(\Sigma_e)$, let $\llbracket c \rrbracket \mathbf{err} = \{\mathbf{err}\}$, and lift $\llbracket c \rrbracket$ to set of states by union. Please note that evaluation of both arithmetic and Boolean expressions only depends on the store, because they cannot dereference variables. The forward collecting semantics of heap commands $\llbracket \cdot \rrbracket : \text{HRCmd} \rightarrow \wp(\Sigma_e) \rightarrow \wp(\Sigma_e)$ is defined as in (2), but on top of that of heap atomic commands.

The semantics $\llbracket \cdot \rrbracket$ of a formula $p \in \text{Asl}$ is a set of states in Σ , and is defined in Fig. 8.

We say a triple $\llbracket p \rrbracket r \llbracket q \rrbracket$ is *valid* if $\llbracket \overleftarrow{r} \rrbracket \llbracket q \rrbracket \supseteq \llbracket p \rrbracket$. To prove soundness of Separation SIL, we rely on a stronger lemma, whose proof is by induction on the derivation tree. Then, by taking $t = \mathbf{emp}$ and using $p * \mathbf{emp} \equiv p$, we get the soundness of the proof system.

LEMMA 5.3. *Let $p, q, t \in \text{Asl}$ and $r \in \text{HRCmd}$. If $\llbracket p \rrbracket r \llbracket q \rrbracket$ is provable and $\text{fv}(t) \cap \text{mod}(r) = \emptyset$ then:*

$$\llbracket \overleftarrow{r} \rrbracket \llbracket q * t \rrbracket \supseteq \llbracket p * t \rrbracket.$$

COROLLARY 5.4 (SEPARATION SIL IS SOUND). *If a Separation SIL triple is provable then it is valid.*

Interestingly, Lemma 5.3 means that Separation SIL proof system can only prove triples that are valid for every frame. This is one solution for the non-local semantics of allocation [Zilberstein et al. 2024, §7], and is realized by not allowing $\llbracket \text{alloc} \rrbracket$ to talk about the specific memory location allocated. In Section 5.7 we will discuss how to change this to make the proof system complete.

5.5 Automating Separation SIL

The main issue with automation of Separation SIL is the need to reconcile a generic assertion with the post of axioms for heap-manipulating atomic commands. To do so, we need two observations.

First, for heap-manipulating atomic commands, all the reachable states share the (de)allocation of a given variable. Consider for instance $\text{free}(x)$: every state reachable after its execution (i.e., any state in $\llbracket \text{free}(x) \rrbracket(\Sigma_e)$) is either \mathbf{err} (which doesn't satisfy any assertion) or a pair (s, h) such that $h(s(x)) = \delta$. This means (s, h) satisfies $(x \not\mapsto * \text{true})$. Therefore, whenever the postcondition

for $\text{free}(x)$ is q , we can strengthen it via $\llbracket \text{cons} \rrbracket$ to $q \wedge (x \not\mapsto * \text{true})$ without losing any reachable state and thus any initial state leading to q . Analogous arguments hold for all the atomic commands.

Second, when the postcondition gets strengthened this way, we rewrite it in the shape $\exists \vec{x}. q * t$, where the axiom can be applied to q and then extended to the whole formula using $\llbracket \text{frame} \rrbracket$ and $\llbracket \text{exists} \rrbracket$. The first step is to lift existential quantifiers to the top level of the assertion (possibly by renaming bound variables). Then, we use the following rewriting:

LEMMA 5.5. *Let $q \in \text{Asl}$ be a formula without \exists , and let x' be a fresh variable. Then,*

- (1) *there exists a \bar{q} such that $q \wedge (x \mapsto x' * \text{true}) \equiv x \mapsto x' * \bar{q}$*
- (2) *there exists a \bar{q} such that $q \wedge (x \not\mapsto * \text{true}) \equiv x \not\mapsto * \bar{q}$*

Particularly, \bar{q} is computed inductively on the structure of q as shown in the following table (the two columns are for points (1) and (2) of the statement, respectively):

q	$\bar{q} \quad (x \mapsto x')$	$\bar{q} \quad (x \not\mapsto)$
$z \mapsto z'$	$z' = x' \wedge z = x \wedge \mathbf{emp}$	false
$z \not\mapsto$	false	$z = x \wedge \mathbf{emp}$
$q_1 * q_2$	$\bar{q}_1 * \bar{q}_2 \vee q_1 * \bar{q}_2$	
\mathbf{emp}	false	
$q_1 \wedge q_2$	$\bar{q}_1 \wedge \bar{q}_2$	
$q_1 \vee q_2$	$\bar{q}_1 \vee \bar{q}_2$	
false	false	
true	true	
$a_1 \simeq a_2$	$a_1 \simeq a_2$	

In Example 5.2 this algorithm finds precisely the disjunction $x = y \vee x \not\mapsto$ in the post of $\text{free}(y)$.

The above Lemma 5.5 is exploited in our prototype implementation for the inference of preconditions. If negation and universal quantification were added to the assertion language, Lemma 5.5 would no longer hold with non-trivial impact on our implementation. Moreover, the procedure derived from Lemma 5.5 is novel and only depends on the assertion language, so it may be applicable to separation logics other than Separation SIL.

5.6 Exit Conditions in Separation SIL

In Example 5.2 the error postcondition $(x \not\mapsto * \text{true})$ is assumed to be given as input. While this assumption is not unreasonable (e.g., the error post was found by a previous analysis, possibly IL-based), in this section we present a possible solution to find it during the analysis by briefly discussing how to adapt Separation SIL to handle different exit conditions. Following O'Hearn [2020], we consider ok and er , denoting correct and erroneous termination respectively. As discussed in Remark 3.9, this correspond to use $\{\text{ok}, \text{er}\} \times \Sigma$ as set of states instead of Σ_e . The denotational semantics of regular commands then acts as described for normal states (returning the error version of the current correct state instead of the generic \mathbf{err}) and as the identity on error states.

The proof system changes accordingly: all the current rules for atoms are still valid with the ok exit condition in both pre and post. For errors, the proof system is augmented with a rule $\llbracket \text{er-id} \rrbracket$ for error propagation together with rules for the atomic commands. We report as an example the error rule for store:

$$\frac{}{\llbracket \text{er} : q \rrbracket \text{ r } \llbracket \text{er} : q \rrbracket} \llbracket \text{er-id} \rrbracket \qquad \frac{}{\llbracket \text{ok} : x \not\mapsto \rrbracket [x] := y \llbracket \text{er} : x \not\mapsto \rrbracket} \llbracket \text{store-er} \rrbracket$$

Example 5.6. Consider a refinement of the program in Example 5.2: here we assume that len and cap are two variables associated to the vector v describing its current length and capacity, respectively. We can then use them to decide the behavior of push_back : the vector gets reallocated

```

 $\langle\langle ok : len \geq cap \wedge len > 7 \wedge (v \mapsto z * z \mapsto - * true) \rangle\rangle$ 
x := [v];       $\langle\langle ok : len \geq cap \wedge len > 7 \wedge (true * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto)) \rangle\rangle$ 
(len  $\geq$  cap)?;  $\langle\langle ok : len > 7 \wedge (true * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto)) \rangle\rangle$ 
y := [v];     //
free(y);      // see Fig. 6b
y := alloc(); //
[v] := y;      $\langle\langle ok : len + 1 > 8 \wedge (x \not\mapsto * true) \rangle\rangle$ 
len := len + 1;  $\langle\langle ok : len > 8 \wedge (x \not\mapsto * true) \rangle\rangle$ 
cap := cap * 2;  $\langle\langle ok : len > 8 \wedge (x \not\mapsto * true) \rangle\rangle$ 
(len > 8)?;    $\langle\langle ok : x \not\mapsto * true \rangle\rangle$ 
[x] := 10      $\langle\langle er : true \rangle\rangle$ 

```

Fig. 9. Sketch of the derivation of the triple for `rclient2` in Example 5.6. Postconditions to each statement are written on the right of the statement itself.

only if the length after the insertion would exceed the current capacity. Moreover, we assume that `x` is used to access the element in position 8 of the vector, and therefore its use after the `push_back` is guarded by a check that the vector is long enough. Therefore, the code becomes

$$\begin{aligned}
r_{client2} &\triangleq x := [v]; \text{ if } (len < cap) \{ r_{b2} \} \text{ else } \{ len := len + 1 \}; r_{use} \\
r_{b2} &\triangleq y := [v]; \text{ free}(y); y := \text{alloc}(); [v] := y; len := len + 1; cap := cap * 2 \\
r_{use} &\triangleq \text{if } (len > 8) \{ [x] := 10 \}
\end{aligned}$$

To analyze this program, we use the error postcondition $\langle\langle er : true \rangle\rangle$. For space constraints, we only consider the code path that goes through the then-branches of both if statements. The proof, linearized, is in Fig. 9. The omitted part is analogous to the derivation in Fig. 6b.

We highlight the use of rule $\langle\langle store-er \rangle\rangle$ to infer the precondition $\langle\langle ok : x \not\mapsto * true \rangle\rangle$ from the error postcondition $\langle\langle er : true \rangle\rangle$ and $\langle\langle assume \rangle\rangle$ to conjoin the boolean conditions in the preconditions of the guards. Moreover, the backward substitution of `len := len + 1` performed by $\langle\langle assign \rangle\rangle$ naturally propagates backward the constraint `len > 8` to the value preceding the assignment, obtaining `len + 1 > 8`. This way, in the precondition of the whole command we explicitly find the constraints `len \geq cap \wedge len > 7` on the initial values of `len` and `cap` that lead to the error. \square

5.7 Relative completeness of Separation SIL

The proof system in Section 5.3 is complete for all atomic commands except `alloc` because $\langle\langle alloc1 \rangle\rangle$ misses the ability to refer to the specific memory location that was allocated. The naive solution to add a constraint `x = β` in the post of $\langle\langle alloc1 \rangle\rangle$ makes the frame rule unsound: for instance, the following triple is not valid:

$$\langle\langle emp * \beta \mapsto - \rangle\rangle x := \text{alloc}() \langle\langle (x \mapsto - \wedge x = \beta) * \beta \mapsto - \rangle\rangle.$$

To recover the frame rule, just like ISL needs the deallocated assertion in the post [Raad et al. 2020, §3], Separation SIL needs a “will be allocate” assertion in the pre. To this end, we use the already available $\not\mapsto$ assertion and change the semantic model to only allocate a memory location that is explicitly δ . We formalize this by letting $avail_h(l) \triangleq (h(l) = \delta)$ in Fig. 7, and removing rule $\langle\langle alloc1 \rangle\rangle$: the only valid rule for allocation in this semantics is $\langle\langle alloc2 \rangle\rangle$. Moreover, we can prove relative completeness [Apt and Olderog 2019, §4.3] for loop-free programs:

THEOREM 5.7 (RELATIVE COMPLETENESS FOR LOOP-FREE PROGRAMS). *Suppose to have an oracle to prove implications between formulae in Asl. Let $r \in \text{HRCmd}$ be a regular command without $*$ and $p, q \in \text{Asl}$ such that $\llbracket \overleftarrow{r} \rrbracket \{q\} \supseteq \{p\}$. Then the triple $\langle p \rangle r \langle q \rangle$ is provable.*

The proof follows very closely the process described in Section 5.5 to automate Separation SIL backward inference, and in fact it was the inspiration for it. The proof uses this process to compute an assertion t whose semantics is precisely the weakest possible pre $\llbracket \overleftarrow{r} \rrbracket \{q\}$ and prove the triple $\langle t \rangle r \langle q \rangle$. Then, using the oracle and $\langle \text{cons} \rangle$, the theorem follows for any p that implies t . Notably, this theorem shows that the weakest (possible) precondition $\llbracket \overleftarrow{r} \rrbracket \{q\}$ of loop-free programs is always expressible as an assertion $t \in \text{Asl}$, namely $\{t\} = \llbracket \overleftarrow{r} \rrbracket \{q\}$. This result is far from trivial, as it depends on the expressiveness of the assertion language: for instance, if we add negation to the assertion language, the same result does not hold anymore because it introduces some new posts for which the precondition is not in the assertion language.

We have several ways to extend the result to loops. Standard techniques include assuming expressivity of the assertion language [Cook 1978] (i.e., assuming the existence of formulae describing weakest preconditions), or extending the assertion language with a least fixed-point operator [Blass and Gurevich 1987]—both solutions work for Separation SIL. Another option is to focus on single states. Note that, differently than other techniques, the following result does not need the oracle: it is always possible to craft a p whose proof only needs decidable implications.

THEOREM 5.8 (STATE-WISE COMPLETENESS). *Given a heap program $r \in \text{HRCmd}$ and two states σ, σ' such that $\sigma \in \llbracket \overleftarrow{r} \rrbracket \sigma'$, for every assertion q such that $\sigma' \in \{q\}$ there exists an assertion p such that $\sigma \in \{p\}$ and $\langle p \rangle r \langle q \rangle$ is provable in Separation SIL.*

5.8 Implementations

To gain confidence in the mechanizability of our results, we developed a proof-of-concept implementation of Separation SIL in OCaml, available on GitHub.⁶ While it is not meant to scale up to real programs, it can derive all the examples in this section and validates our intuition on the automated backward inference of Section 5.5. The code exploits a routine implementing the process described in that section, which is analogous to the proof of relative completeness (Theorem 5.7).

Moreover, the ideas that lead to SIL represent *a posteriori* formalization and theoretical justification of the parallel, modular, and compositional static analysis implemented in industrial grade static analyzers for security developed and used at Meta, such as Zoncolan [Distefano et al. 2019], Mariana Trench [Gabi 2021], and Pysa [Bleaney and Cepel 2019]. Those tools automatically find more than 50% of the security bugs in the Meta family of apps and many SEVs (“severe bugs” in the Meta jargon) [Distefano et al. 2019, Fig. 5].

In order to scale up to hundreds of millions of lines of code, static analyses need to be parallelizable and henceforth modular and compositional. Modularity implies that the analysis can infer *meaningful* information without full knowledge of the global program. Compositionality means that the *results* of analyzing modules are good enough that one does not lose information in using the inferred triple instead of inlining the code.

The analysis implemented in the aforementioned tools is a modular backward analysis that determines which input states for a function will lead to a security error, likewise the SIL rules described in Fig. 4. In particular, the analysis infers sufficient incorrectness preconditions (modularity) for callees that can be used by the callers (compositionality) to generate their incorrectness preconditions. When the propagation of the inferred sufficient precondition reaches an attacker-controlled input, the analyzers check if that input is included in the propagated error condition. If it is the

⁶<https://github.com/flavio-a/Failure-SSIL-Analyser/tree/oopsla25>

// attacker setup	$\langle \text{true} \rangle$
$x := \text{alloc}();$	$\langle x \mapsto - \rangle$
$\text{free}(x);$	$\langle \langle x \not\mapsto \rangle \vee (x \mapsto \text{secret} * \text{true}) \rangle$
// user code reading the secret	
$y := \text{alloc}();$	$\langle \langle y \mapsto - * x = y \rangle \vee (x \mapsto \text{secret} * y \mapsto - * \text{true}) \rangle$
$[y] := \text{secret};$	$\langle \langle x \mapsto \text{secret} * x = y \rangle \vee (x \mapsto \text{secret} * y \mapsto \text{secret} * \text{true}) \rangle$
// attacker code leaking the secret	
$\text{leak} := [x]$	$\langle \text{leak} = \text{secret} * x \mapsto - \rangle$
	$\langle \text{leak} = \text{secret} \rangle$

Fig. 10. Sketch of the derivation for Example 5.9.

case, then it emits an error. The function analyses are parallelized and a strategy similar to the iteration rule of Fig. 4 is used to compute the fixpoint in presence of mutually recursive functions. Specifically, the automation of deriving sufficient incorrectness preconditions opens up new avenues for revisiting the handling of recursive functions and complex contexts, suggesting potential optimizations in dependency analysis between modules and in fixpoint computation.

5.9 Comparisons

There are other logics that address memory errors with an “incorrectness” perspective. Here we compare Separation SIL to ISL and Outcome Separation Logic (OSL) [Zilberstein et al. 2024].

The key difference between Separation SIL and ISL is the same between IL and SIL: ISL is more effective for forward analysis and ensures reachability of all final states; Separation SIL is thought for backward analysis and guarantees that all initial states can reach a final state. For instance, the ISL triple in Raad et al. [2020] for the Example 5.2 is $[v \mapsto z * z \mapsto -] \text{rclient } [v \mapsto y * y \mapsto - * x \not\mapsto]$, which proves that those faulty states are reachable, but tells nothing about which input states actually lead to them. On the other hand, the Separation SIL triple has a more succinct post capturing the error and exposes faulty initial states, but cannot describe which errors are reachable.

The comparison with OSL is more convoluted. We already compared SIL and OL in Section 4.2, but we remark that, while in Zilberstein et al. [2023, Fig. 6] outcome-based separation logic proves essentially the same triple as Example 5.2, the deduction process is quite different. OL reasoning is forward oriented, as witnessed by the implication that concludes the proof and by the triple for the skip branch, whereas Separation SIL rules naturally guides the backward inference. OSL is more closely related to Separation SIL because it takes pointer programs at its core. For OSL, the authors propose an abduction-based algorithm that produces similar results to Separation SIL. However, the two algorithms differ in their details and work in different settings. To tackle the issue of *non-locality* of allocation, Zilberstein et al. [2024, §2.2] prescribe that an OSL triple should be valid for *every* possible allocation semantics. This prevents triples from talking too precisely about the result of an allocation, and is available even when the execution model doesn’t include nondeterminism. Thus, OSL frame rule works across different execution models, e.g., for probabilistic programs. On the contrary, Separation SIL model is tailored to nondeterminism and can’t be used for other execution models. However, this allows Separation SIL to be more precise about nondeterminism: differently than OSL, Separation SIL is complete and its triples can talk about a *specific* allocation semantics. Thus Separation SIL can prove some special kinds of triples that are not valid in OSL model.

Example 5.9. C standard allocator reuses previously freed locations, and this can cause security leaks. Consider the code in Fig. 10: here an attacker sets up a memory address by allocating and

then freeing it, so that the next user allocation gets exactly that location ℓ . The user then stores a secret in ℓ , that the attacker can read. Separation SIL witnesses the error: in the post of $y := \text{alloc}()$ it can be seen that the leaky behaviour is caused by $(x = y)$. In contrast, OSL cannot witness this error: since it occurs only for a specific allocation semantics this triple is not valid. \square

This simple example shows that, for some kind of errors, a more specific semantics is needed, thus the usefulness of different techniques for both more general and more specific settings.

6 Conclusion and Future Work

We have formalized the SIL proof system for backward under-approximation of programs with the aim to unveil the causes of errors. Our choice of rules allows us to define a taxonomy of program logics and to compare them in a clear and concise way. The SIL proof system also leads to our main technical contribution: we combine SIL and Separation Logic to define Separation SIL, a proof system to detect inputs leading to memory errors. Using a carefully crafted assertion language, we proved that Separation SIL is relatively complete. Simple examples show that in some cases Separation SIL can be more advantageous than ISL and OSL. A proof-of-concept implementation shows that our process is fully mechanizable and helped us double checking our examples.

The relationships highlighted by our taxonomy can be summarized as follows:

HL vs SIL: Although HL and SIL share the same consequence rule, they are generally not comparable. However, for terminating programs, HL implies SIL, while for deterministic programs, SIL implies HL.

NC vs IL: Despite sharing the same consequence rule, NC and IL are not comparable unless the program is reversible, in which case IL implies NC.

NC vs SIL: The only triples common to NC and SIL are the exact ones.

IL vs SIL: Unlike the relationship between HL and NC, IL and SIL triples cannot be related using negation. As they provide complementary analyses, the best way to use them synergistically is to identify errors using IL and then trace their causes with SIL, or vice versa

OL vs SIL: Both OL's generality and SIL's specificity have value. On the one hand, OL can be instantiated for various execution models and prove both correctness and incorrectness. On the other hand, (Separation) SIL facilitates a syntax-directed, straightforward backward algorithm and captures a wider range of nondeterministic behaviors.

In conclusion, each logic has its own goals and strengths, and we think they should be used for complementary and more informative analyses, rather than being perceived in competition.

Future Work. It seems interesting to explore a synergic use of different logics for practical issues. For instance, a main application of SIL that we envisage is in strict combination with IL: once an error is detected with IL, its IL proof tree will be exploited to track back the actual source of error by a clever application of SIL rules. To this aim, it could be useful to extend the taxonomy of HL, NC, IL and SIL by incorporating some other approaches [Hoare 1978; Maksimovic et al. 2023; Raad et al. 2022, 2024; Zilberstein et al. 2023], as discussed in Section 2. Within this context, it would be interesting to study the interplay between more expressive assertion languages for ISL and Separation SIL and the consequences at both the theoretical and practical level.

Another possible application of SIL is in combination with white-box fuzzy testing [Bounimova et al. 2013; Ganesh et al. 2009]: given some program states, a SIL analysis can find preconditions leading to them, thus reducing the input state space to randomly sample with fuzzing.

Lastly, we plan to compare the handcrafted SIL proof system with the one obtained following Cousot's recipe for its calculational design [Cousot 2024] to gain some insights on auxiliary rules and induction principles that can emerge.

7 Data-Availability Statement

The prototype implementation of Separation SIL we describe in Section 5.8 is available on GitHub.⁷ It has been exploited to validate the mechanization process described in Section 5.5. The prototype was developed as a proof-of-concept student project. It has not been submitted as an artefact, because it was not meant to overcome other tools in terms of performance, precision or usability.

Acknowledgments

We sincerely appreciate the time and effort of the anonymous reviewers, whose valuable comments and constructive suggestions have helped improve the quality of our work. In particular, the correspondence outlined in Lemma 4.9 has been pointed out by the reviewers. We also thank Noam Zilberstein for the insightful discussion on Separation SIL.

This research has been partially supported by the Italian Ministero dell'Università e della Ricerca under Grant No. P2022HXNSC, PRIN 2022 PNRR – *Resource Awareness in Programming: Algebra, Rewriting, and Analysis (RAP)*, by the project *Security and Rights In the CyberSpace (SERICS)*, code PE00000014 - CUP H73C2200089001, under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU, by the INDAM-GNCS Project, code CUP_-E53C22001930001, *Reversibilità In Sistemi Concorrenti: analisi Quantitative e Funzionali (RISICO)*, and by the University of Pisa4PRA_2022_99 *Formal methods for the healthcare domain based on spatial information (FM4HD)*

References

- Krzysztof R. Apt. 1981. Ten Years of Hoare's Logic: A Survey—Part I. *ACM Trans. Program. Lang. Syst.* 3, 4 (10 1981), 431–483. <https://doi.org/10.1145/357146.357150>
- Krzysztof R. Apt. 1984. Ten Years of Hoare's Logic: A Survey Part II: Nondeterminism. *Theor. Comput. Sci.* 28 (1984), 83–109. [https://doi.org/10.1016/0304-3975\(83\)90066-X](https://doi.org/10.1016/0304-3975(83)90066-X)
- Krzysztof R. Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Formal Aspects Comput.* 31, 6 (2019), 751–807. <https://doi.org/10.1007/S00165-019-00501-3>
- Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 260–264. https://doi.org/10.1007/3-540-44585-4_25
- Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM* 64, 8 (2021), 56–68. <https://doi.org/10.1145/3470569>
- Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects - 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer, 115–137. https://doi.org/10.1007/11804192_6
- Marc Bezem and Thierry Coquand. 2005. Automating Coherent Logic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3835)*, Geoff Sutcliffe and Andrei Voronkov (Eds.). Springer, 246–260. https://doi.org/10.1007/11591191_18
- Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 196–207. <https://doi.org/10.1145/781131.781153>
- Andreas Blass and Yuri Gurevich. 1987. Existential Fixed-Point Logic. In *Computation Theory and Logic, In Memory of Dieter Rödding (Lecture Notes in Computer Science, Vol. 270)*, Egon Börger (Ed.). Springer, 20–36. https://doi.org/10.1007/3-540-18170-9_151
- Andreas Blass and Yuri Gurevich. 2001. The Underlying Logic of Hoare Logic. In *Current Trends in Theoretical Computer Science, Entering the 21st Century*, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa (Eds.). World Scientific, 409–436.

⁷<https://github.com/flavio-a/Failure-SSIL-Analyser/tree/oopsla25>

- Graham Bleaney and Sinan Cepel. 2019. Pysa: An Open Source Static Analysis Tool to Detect and Prevent Security Issues in Python Code. <https://engineering.fb.com/2021/09/29/security/mariana-trench/>.
- Ella Bounimova, Patrice Godefroid, and David A. Molnar. 2013. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 122–131. <https://doi.org/10.1109/ICSE.2013.6606558>
- Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* 70, 2 (2023), 15:1–15:45. <https://doi.org/10.1145/3582267>
- Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 289–300. <https://doi.org/10.1145/1480881.1480917>
- Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2
- Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (1978), 70–90. <https://doi.org/10.1137/0207005>
- Patrick Cousot. 2024. Calculational Design of [In]Correctness Transformational Program Logics by Abstract Interpretation. *Proc. ACM Program. Lang.* 8, POPL, Article 7 (jan 2024), 34 pages. <https://doi.org/10.1145/3632849>
- Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 128–148. https://doi.org/10.1007/978-3-642-35873-9_10
- Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 150–168. https://doi.org/10.1007/978-3-642-18275-4_12
- Patrick Cousot, Radhia Cousot, Francesco Logozzo, and Michael Barnett. 2012. An Abstract Interpretation Framework for Refactoring with Application to Extract Methods with Contracts. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 213–232. <https://doi.org/10.1145/2384616.2384633>
- Brian A. Davey and Hilary A. Priestley. 2002. *Introduction to Lattices and Order*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511809088>
- 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, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12
- Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. <https://doi.org/10.1145/360933.360975>
- Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. <https://doi.org/10.1145/3338112>
- Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. <http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf>
- Dominik Gabi. 2021. Open-sourcing Mariana Trench: Analyzing Android and Java App Security in Depth. <https://engineering.fb.com/2021/09/29/security/mariana-trench/>.
- Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 474–484. <https://doi.org/10.1109/ICSE.2009.5070546>
- David Harel, Dexter Kozen, and Jerzy Tiuryn. 2000. *Dynamic logic* (1st ed.). MIT Press.
- 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 (1978), 461–480. <https://doi.org/10.1145/322077.322088>
- 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 (2022), 1–27. <https://doi.org/10.1145/3527325>

- Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. <https://doi.org/10.1145/1538788.1538814>
- Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity Specifications for Robustness. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 811–840. <https://doi.org/10.1145/3563317>
- Petar Maksimovic, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPICs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:27. <https://doi.org/10.4230/LIPICs.ECOOP.2023.19>
- Zohar Manna. 1974. *Mathematical Theory of Computation* (4th ed. ed.). McGraw-Hill.
- Bernhard Möller, Peter W. 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 (Lecture Notes in Computer Science, Vol. 13027)*, Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter (Eds.). Springer, 325–343. https://doi.org/10.1007/978-3-030-88701-8_20
- Peter W. O’Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. <https://doi.org/10.1145/3371078>
- Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1
- Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O’Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 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 (2022), 1–29. <https://doi.org/10.1145/3498695>
- Azalea Raad, Julien Vanegue, Josh Berdine, and Peter W. O’Hearn. 2023. A General Approach to Under-Approximate Reasoning About Concurrent Programs. In *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium (LIPICs, Vol. 279)*, Guillermo A. Pérez and Jean-François Raskin (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:17. <https://doi.org/10.4230/LIPICs.CONCUR.2023.25>
- Azalea Raad, Julien Vanegue, and Peter W. O’Hearn. 2024. Non-termination Proving at Scale. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 246–274. <https://doi.org/10.1145/3689720>
- John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. <https://doi.org/10.1109/LICS.2002.1029817>
- Freek Verbeek, Md Syadus Sefat, Zhoulai Fu, and Binoy Ravindran. 2025. On Extending Incorrectness Logic with Backwards Reasoning. *Proc. ACM Program. Lang.* 9, POPL, Article 14 (1 2025), 25 pages. <https://doi.org/10.1145/3704850>
- Lena Verscht and Benjamin Lucien Kaminski. 2025. A Taxonomy of Hoare-Like Logics: Towards a Holistic View using Predicate Transformers and Kleene Algebras with Top and Tests. *Proc. ACM Program. Lang.* 9, POPL, Article 60 (1 2025), 30 pages. <https://doi.org/10.1145/3704896>
- G. Winskel. 1993. *The Formal Semantics of Programming Languages: an Introduction*. MIT press.
- 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 (2022), 1–29. <https://doi.org/10.1145/3527331>
- Linpeng Zhang, Noam Zilberstein, Benjamin Lucien Kaminski, and Alexandra Silva. 2024. Quantitative Weakest Hyper Pre: Unifying Correctness and Incorrectness Hyperproperties via Predicate Transformers. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 817–845. <https://doi.org/10.1145/3689740>
- Noam Zilberstein. 2024. A Relatively Complete Program Logic for Effectful Branching. arXiv:2401.04594 [cs.LO] Preprint.
- Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 522–550. <https://doi.org/10.1145/3586045>
- Noam Zilberstein, Angelina Salliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 276–304. <https://doi.org/10.1145/3649821>

Received 2024-10-16; accepted 2025-02-18