U-Turn: Enhancing Incorrectness Analysis by Reversing Direction

FLAVIO ASCARI, University of Konstanz, Germany
ROBERTO BRUNI, University of Pisa, Italy
ROBERTA GORI, University of Pisa, Italy
AZALEA RAAD, Imperial College London, United Kingdom

O'Hearn's Incorrectness Logic (IL) has sparked renewed interest in static analyses that aim to detect program errors rather than prove their absence, thereby avoiding false alarms—a critical factor for practical adoption in industrial settings. As new incorrectness logics emerge to capture diverse error-related properties, a key question arises: *can combining correctness and incorrectness techniques enhance precision, expressiveness, automation, or scalability?* Notable frameworks, such as outcome logic, UNTer, local completeness logic, and exact separation logic, unify multiple analyses within a single proof system. In this work, we adopt a complementary strategy. Rather than designing a unified logic, we combine IL, which identifies reachable error states, with Sufficient Incorrectness Logic (SIL), which finds input states potentially leading to those errors. As a result, we get a more informative and effective analysis than either logic in isolation. Rather than sequencing them, our key innovation is reusing heuristic choices from the first analysis to steer the second. In fact, both IL and SIL rely on under-approximation and thus their automation legitimizes heuristics that avoid exhaustive path enumeration (e.g., selective disjunct pruning, loop unrolling). Concretely, we instrument the proof rules of the second logic with derivations from the first to inductively guide rule selection and application. To our knowledge, this is the first rule format enabling such inter-analysis instrumentation. This combined analysis aids debugging and testing by revealing both reachable errors and their causes, and opens new avenues for embedding incorrectness insights into scalable, expressive, automated code contracts.

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

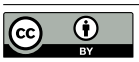
Additional Key Words and Phrases: Sufficient Incorrectness Logic, Incorrectness Logic

ACM Reference Format:
Flavio Ascari, Roberto Bruni, Roberta Gori, and Azalea Raad. 2026. U-Turn: Enhancing Incorrectness Analysis by Reversing Direction. *Proc. ACM Program. Lang.* 10, POPL, Article 46 (January 2026), 27 pages. https://doi.org/10.1145/3776688

1 Introduction

Formal methods apply mathematical reasoning to software development, aiming to guarantee correctness, reliability, and security of programs through automated analyses. Over the last decades these techniques have led to a number of high-profile successes. For example, the Astrée static analyzer uses abstract interpretation to prove the absence of run-time errors in safety-critical C code [Blanchet et al. 2003], Microsoft's SLAM project (and its Static Driver Verifier tool) applied model checking to millions of lines of Windows driver code, uncovering subtle bugs in API usage [Ball and Rajamani 2001], the CompCert project produced a formally verified C compiler: its machine-checked proof of semantic preservation guarantees that any property proved on the source code holds on the

Authors' Contact Information: Flavio Ascari, University of Konstanz, Konstanz, Germany, flavio.ascari@phd.unipi.it; Roberto Bruni, University of Pisa, Pisa, Italy, roberto.bruni@unipi.it; Roberta Gori, University of Pisa, Pisa, Italy, roberta.gori@unipi.it; Azalea Raad, Imperial College London, London, United Kingdom, azalea.raad@imperial.ac.uk.



ACM 2475-1421/2026/1-ART46
https://doi.org/10.1145/3776688

compiled executable [Leroy 2009]. Other tools have shown similar impact: the VCC verifier checks annotated concurrent C programs against strong safety and functional properties [Baudin et al. 2021], and the Frama-C platform provides a collaborative, extensible suite of static and deductive analyzers for C, supported by a large academic/industrial community [Baudin et al. 2021].

Despite these achievements, the widespread adoption of formal verification in general software development remains limited. A key obstacle is their focus on proving correctness of the program, which often leads to so-called false alarms. They are warnings produced by the analysis that do not correspond to actual bugs in the code. Such spurious errors stem from over-approximation and are particularly frustrating for experienced engineers, who tend to perceive them as distractions rather than helpful insights. This issue becomes even more pronounced in industrial settings, where code is rarely correct on the first attempt. Instead, code development typically follows an iterative process of writing, testing, and refinement. These considerations have prompted a shift in perspective, from *verifying correctness* to *detecting incorrectness*. As a result, there is a growing interest in developing formal methods that are aimed at actively uncover bugs rather than prove their absence. In industrial contexts, under-approximation techniques—such as testing and bounded model checking—are often preferred because they avoid false positives. A notable development in this direction is Incorrectness Logic (IL) [O'Hearn 2020], a program logic specifically designed for bug detection: any error state appearing in the postcondition is guaranteed to be reachable from some input state satisfying the precondition. IL has inspired the creation of practical tools, such as Pulse, which builds on Incorrectness Separation Logic [Raad et al. 2020], and Pulse-X [Le et al. 2022]. This foundational work has sparked a new line of research on principled under-approximate approaches to bug detection [Ascari et al. 2025a; Möller et al. 2021; Zilberstein et al. 2023], as well as on the development of industrial-strength tools for scalable bug finding [Distefano et al. 2019; Sadowski et al. 2018].

When dealing with real-world applications that must be both effective and scalable, some form of approximation is unavoidable. This is a direct consequence of Rice's theorem [Rice 1953], which states that any non-trivial semantic property of programs is undecidable. Therefore, approaches based on under-approximation offer a practical means of scaling analyses, as they allow discarding part of the information while preserving soundness. As an example, the inference rules of logics for under-approximation can discard disjuncts or bound the number of loops of iterative commands. Such abstractions are especially valuable in industrial settings, where incomplete information is the norm and nondeterministic behaviors may emerge simply due to the lack of source code or specifications for external library calls. In these scenarios, under-approximations enable analysis tools to efficiently produce sound results, often at the expense of completeness—that is, the tools may fail to detect all errors—by relying on heuristics that automate the analysis.

The Problem. While early approaches in the literature typically focused on proving either correctness or incorrectness, several successful proposals have since emerged that combine these complementary techniques, resulting in methods that are either more powerful (e.g., [Bradley 2011; Bruni et al. 2023]) or capable of expressing a broader range of properties (e.g., [Raad et al. 2024a; Zilberstein et al. 2023]). The work that we present here follows this line of thought, firmly grounded in the Aristotelian conviction that "the whole is greater than the sum of its parts." In particular, we aim at combining Incorrectness Logic (IL) [O'Hearn 2020] with Sufficient Incorrectness Logic (SIL) [Ascari et al. 2025a]. While the goal of IL is to discover reachable error states—such that *any error state in the postcondition can be reached from some input state satisfying the precondition*—the goal of SIL, once the postcondition characterizes potential errors, is to identify their sources. In fact, SIL guarantees that *every state in the precondition has an execution leading to an error state in the postcondition*. Both IL and SIL are based on under-approximations; however, while IL can be

expressed using a forward semantics, SIL relies on a backward semantics. IL postconditions expose only true errors, and when paired with the corresponding SIL preconditions, they can be presented to programmers as both reachable errors themselves together with the input states that lead to them, thus aiding the debugging process. This suggests their combination is more informative than what each logic can tell individually. Indeed, even if sometimes the path conditions leading to an error can be made explicit in the post of IL triples by means of logical variables—variables not appearing in the program and thus never modified during execution—that can keep track of initial values, the next example shows that IL triples may not be informative enough to provide the programmer with the input conditions from which to start debugging. This is the case, for example, when the constraints are related to the shape of the heap.

Example 1.1. Consider the following program:

$$\texttt{r} \triangleq \texttt{tmp := [x]; if (tmp == 0) \{ free(x); error() \}}$$

This code fragment reads a value from the pointer x and expects a value different from 0. However, if it loads 0, it first deallocates x to reclaim resources before throwing an error. In this case, the path condition leading to the error is that x points to 0, but this cannot be encoded in the postcondition using logical variables because the heap has changed from the pre to the post, due to the execution of `free(x)`. In principle, the Incorrectness Separation Logic (ISL) [Raad et al. 2020] triple $[x \mapsto 0]$ r $[er : \texttt{tmp} = 0 * x \not\mapsto]$ which highlights the error precondition $x \mapsto 0$ is provable. However, ISL (and IL) validity condition does not guarantee that some error state is necessarily reachable from all states satisfying the precondition. For instance, a straightforward application of ISL proof system proves the triple $[\texttt{tmp} = \texttt{tmp}']$ r $[er : \texttt{tmp} = 0 * x \not\mapsto]$, which does not contain any reference to $x \mapsto 0$ as the condition leading to the error.

The previous example suggests that the use of SIL, starting from the error postcondition identified through IL, could be useful for generating a warning that also highlights the input that led to the error state, namely the precondition $x \mapsto 0$ in the previous example. Our proposal is not to combine IL and SIL in a single proof system where both triples can be derived, but rather to propose a novel proof system that taken a derivation in one logic uses such proof tree to automatically instruct the inference of a triple in the other logic. Next example shows that this can be particularly useful.

Example 1.2. With SIL backward analysis, it is not always possible to determine which execution paths will produce the error. Although this limitation also exists in IL, the highly nondeterministic nature of SIL's backward semantics makes the problem particularly severe. Consider the following program, which inevitably exhibits an erroneous behavior:

$$\texttt{r} \triangleq \texttt{x := 10; while (x > 0) \{ x-- \}; error()}$$

Unfortunately, for detecting that true is a valid SIL pre to an error postcondition, an analyzer has to guess that the loop must be executed 10 times. For instance, if the analyzer decides to unroll the loop (backward) only twice, it will derive the triple $\langle x = 2 \rangle$ `while (x > 0) { x-- }` $\langle \text{true} \rangle$, which would propagate as $\langle 10 = 2 \rangle$ `x := 10` $\langle x = 2 \rangle$ using Hoare's axiom for assignment, which does not expose any cause for the error—in fact, false is always a valid under-approximation.

To ensure it tracks back to some source of errors, SIL analysis should take into account all possible nondeterministic backward-oriented executions, which is infeasible. Therefore, there is the need for good heuristics to prune the search. Another example of the problems raised by high degree of nondeterminism is related to pointer aliasing. While aliasing created during a function execution is easy to detect and track in a forward analysis, it is much harder to infer in a backward analysis; therefore, to find non-trivial preconditions, all possible aliasing must be considered until

$\langle \text{true} * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto) \rangle$
`  y := [v];`
$\langle \text{true} * v \mapsto - * y \mapsto - * (x = y \vee x \not\mapsto) \rangle$
`  free(y);`
$\langle x \not\mapsto * v \mapsto - * \mathbf{emp} * \text{true} \rangle$
`  y := alloc();`
$\langle x \not\mapsto * v \mapsto - * \text{true} \rangle$
`  [v] := y;`
$\langle x \not\mapsto * \text{true} \rangle$

Fig. 1. SIL derivation for the reallocation case of push_back [Ascari et al. 2025a, Fig. 6]

reaching a program point where we can prove they are not admissible. Again, the same issue can happen in a forward analysis, but in practice functions seldom receive two aliased pointers as parameters. On the contrary, it is very common to create some temporary aliases of a pointer for local manipulation, that are discarded before the function returns.

Example 1.3. Consider this code fragment, which models the reallocation case of C++ push_back function (see Example 4.4):

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

and the Separation Logic precondition ($v \mapsto x * x \mapsto -$). Executing the assignment `y := [v]` aliases x and y, so that the `free(y)` deallocates the pointer x. It is easy to find this information in the preceding line of the forward analysis, where y gets assigned the value pointed by v that is exactly x, and find that at the end x is deallocated. By contrast, if we start from the error postcondition ($x \not\mapsto * \text{true}$) with a backward analysis, this information is not known until we reach the caller of this code fragment, and therefore we have to consider both possibilities in the pre. This can be seen in the SIL derivation in Fig. 1 (first presented in Ascari et al. [2025a]), that must account for both cases, whether they are aliased ($x = z$) or they are not ($z \not\mapsto * x \not\mapsto$).

The idea of combining IL and SIL analyses has already appeared in the literature. Notably, in Raad et al. [2024a], the authors point out the importance of both forward and backward under-approximation information, and exploit it to reason about termination, introducing the new UNTer proof system. UNTer logics effectively prove triples that are valid for both IL and SIL. When turning to the implementation, they realize that Pulse (an industrial-strength automated tool in use at Meta) *already* implemented an analysis that computed triples valid both in IL and SIL (albeit without realizing it explicitly), demonstrating the strength and impact of this combined approach. However, integrating forward and backward reasoning into a single proof system presents certain challenges.

Designing a proof system that supports both directions crucially relies on formulating appropriate axioms for atomic commands. For instance, while in SIL both classical axioms for assignment used in Hoare logic—Hoare's backward substitution rule [Hoare 1969] and Floyd's forward transformer [Floyd 1967]—remain valid, ensuring similar validity and completeness in a unified system is non-trivial. The axioms are:

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

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]. Of course, one natural solution is to consider Floyd's forward axiom, which is valid for both IL and SIL triples.

However, this raises an important question: is this the most general axiom we can design for assignments? Could it be that the most general axiom is neither of the previously proposed ones, which were tailored specifically for forward or backward reasoning?

Another key question is: which direction should be prioritized? In other words, should we begin with the precondition and attempt to infer the appropriate postcondition in a forward style (as done in IL and UNTer), or should we start from the postcondition and infer the corresponding precondition in a backward style (as done in SIL)? In the case of assignment, Floyd's axiom would be the natural candidate for the first approach and Hoare's axiom for the second. In a combined proof system such as UNTer's, proving a triple usually requires the user to make an informed guess for the appropriate pre- and postconditions to use. In fact, it is not the case that for any precondition p (respectively, any postcondition q) we can find a corresponding triple that is valid in both IL and SIL. This aspect can be particularly challenging, especially when reasoning about unknown or partially known code, as the user must provide valid triples even in the absence of full information.

Contribution. In response to the above questions, we make two key contributions.

First, we address the problem of formulating axioms for atomic commands that ensure derivation of all and only triples that are valid in both IL and SIL. To this aim, we propose an *axiom schema* for atomic commands that is sound and complete. From this schema, we derive axioms for atomic commands that are correct by construction, and we demonstrate how such axioms can be instantiated within the UNTer proof system. Moreover, our contribution goes further: by introducing this general schema, we establish a methodology that can be systematically applied to any atomic command. As a concrete example, we consider non-deterministic assignment—a command not previously supported in UNTer—and immediately derive a new sound and complete axiom for it.

Our second contribution addresses the direction of the analysis. Instead of proposing a single, combined proof system that naturally favors one or the other direction, we suggest a "smart sequential composition" of one analysis followed by the other one. In particular, in this paper, we instantiate this idea by focusing on the strategy that applies IL followed by SIL—that is, where the results of IL analysis serve as the starting point for SIL. However, the opposite strategy is also possible and we briefly outline it in the paper. As mentioned earlier, IL helps identifying reachable error states, and SIL complements this information by producing inputs and warnings that aid the programmer in debugging their code. We introduce U-Turn proof system, which allows to follow any IL derivation with a backward SIL analysis. This combination is not only more informative—since the result satisfies the properties guaranteed by each individual method—but, to the best of our knowledge, it is also the first case where the heuristic exploited by one method is used to guide the application of the other.

Example 1.4. We briefly revisit the previous examples to show how U-Turn solves their issues.

In Example 1.1, the issue is solved by doing a SIL backward step starting from the error postcondition found by IL. Moreover, this backward step is guided by the forward analysis, that considered the then-branch of the conditional statement, therefore SIL will only analyze it and skip the analysis of the else-branch, finding the desired precondition $\langle x \mapsto 0 \rangle$.

In Example 1.2, a first forward step with IL needs to unroll the loop until it exits (i.e., 10 times) to find the postcondition $[er : x = 0]$. Taking this information into account, the backward SIL step can unroll the loop the same number of times to derive $\langle x = 10 \rangle$ `while (x > 0) { x-- }` $\langle \text{true} \rangle$, which will then propagate as $\langle 10 = 10 \rangle$ `x := 10` $\langle x = 10 \rangle$ recovering the error precondition $\langle true \rangle$.

In Example 1.3, SIL was already able to infer the error precondition by itself (see [Ascari et al. 2025a, Ex. 5.2]), but it had to consider both the aliasing and not aliasing of y and x. However, the forward IL analysis already has the information that x and y will be aliased. U-Turn is able to

transfer this information, forcing SIL to only consider this latter case and drop the other possibility. We show the details of this interaction in Example 4.4.

Structure of the Paper. The paper is structured as follows. In Section 2 we set the notation and introduce relevant concepts from the literature. In Section 3 we present our first contribution, the forward-backward axiom schema for atomic commands. In Section 4 we detail our second contribution, the U-Turn proof system. In Section 5 we outline possible directions for future works. Proofs and additional technical material can be found in the extended version [Ascari et al. 2025b].

2 Background

2.1 Regular Commands

Following the trend of many other incorrectness logics [Ascari et al. 2025a; O'Hearn 2020; Raad et al. 2020, 2024a] we consider a language of regular commands. We use standard definitions for arithmetic expressions $\mathsf{a} \in \mathsf{AExp}$ and Boolean expressions $\mathsf{b} \in \mathsf{BExp}$:

$$\mathsf{AExp} \ni \mathsf{a} ::= n \mid x \mid \mathsf{a} + \mathsf{a} \mid \mathsf{a} - \mathsf{a} \mid \mathsf{a} \cdot \mathsf{a} \mid \dots \qquad \mathsf{BExp} \ni \mathsf{b} ::= \mathsf{false} \mid \neg \mathsf{b} \mid \mathsf{b} \wedge \mathsf{b} \mid \mathsf{a} \asymp \mathsf{a}$$

where $\asymp \in \{=, \neq, \leq, <, \dots\}$ accounts for all standard comparison operators.

The syntax of regular commands $\mathsf{r} \in \mathsf{RCmd}$ is:

$$\mathsf{ACmd} \ni \mathsf{c} ::= \texttt{skip} \mid \texttt{x := a} \mid \texttt{b?} \mid \texttt{x := nondet()} \qquad \mathsf{RCmd} \ni \mathsf{r} ::= \mathsf{c} \mid \mathsf{r}; \mathsf{r} \mid \mathsf{r} \boxplus \mathsf{r} \mid \mathsf{r}^* \tag{1}$$

Note that we include an explicit nondeterministic assignment `x := nondet()` as one of the atomic commands in the language, as well as nondeterministic choice $\boxplus$ and iteration $(\cdot)^*$.

This formulation accommodates a standard imperative while-language [Winskel 1993] with the encoding below:

$$\texttt{if (b) \{r}_1\texttt{\} else \{r}_2\texttt{\}} \triangleq (\texttt{b?; r}_1) \boxplus ((\neg \texttt{b})\texttt{?; r}_2)$$
$$\texttt{while (b) \{r\}} \triangleq (\texttt{b?; r})^*; (\neg \texttt{b})\texttt{?}$$

To give a semantics to regular commands, we consider a finite set of variables Var. Let stores $\sigma \in \Sigma \triangleq (\mathsf{Var} \to \mathbb{Z})$ be (total) functions from variables to values. As usual, store update is denoted by $\sigma[x \mapsto v]$. Evaluation of arithmetic and boolean expressions in a store σ, denoted by $(\!|\cdot|\!)\sigma$, is standard. We consider a collecting denotational semantics for regular commands. We define it as a function $[\![\cdot]\!] : \mathsf{RCmd} \to \Sigma \to \wp(\Sigma)$, which is then lifted to $[\![\cdot]\!] : \mathsf{RCmd} \to \wp(\Sigma) \to \wp(\Sigma)$ by union. The semantics of atomic commands $\mathsf{c} \in \mathsf{ACmd}$ and $S \in \wp(\Sigma)$ is defined as follows:

$$[\![\texttt{skip}]\!]\sigma \triangleq \{\sigma\} \qquad [\![\texttt{x := a}]\!]\sigma \triangleq \{\sigma[x \mapsto (\!|\mathsf{a}|\!)\sigma]\}$$
$$[\![\texttt{b?}]\!]\sigma \triangleq \{\sigma \mid (\!|\mathsf{b}|\!)\sigma = \mathbf{tt}\} \qquad [\![\texttt{x := nondet()}]\!]\sigma \triangleq \{\sigma[x \mapsto v] \mid v \in \mathbb{Z}\}$$

We then define the semantics of composite regular commands by induction as follows:

$$[\![\mathsf{r}_1; \mathsf{r}_2]\!]\sigma \triangleq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]\sigma) \qquad [\![\mathsf{r}_1 \boxplus \mathsf{r}_2]\!]\sigma \triangleq [\![\mathsf{r}_1]\!]\sigma \cup [\![\mathsf{r}_2]\!]\sigma \qquad [\![\mathsf{r}^*]\!]\sigma \triangleq \bigcup_{n \geq 0} [\![\mathsf{r}]\!]^n \sigma \tag{2}$$

Roughly speaking, given a set of stores $S \subseteq \Sigma$, the collecting forward semantics $[\![\mathsf{r}]\!]S$ is the set of output states reachable from input states in S by executing r.

The forward semantics can also be viewed as a binary relation over Σ, relating a pair of states (σ, σ') if and only if $\sigma' \in [\![\mathsf{r}]\!]\sigma$. Following the presentation of SIL [Ascari et al. 2025a, §3.1], we define the backward semantics $[\![\overleftarrow{\mathsf{r}}]\!]$ as the function inducing the opposite relation, that is

$$\sigma \in [\![\overleftarrow{\mathsf{r}}]\!]\sigma' \iff \sigma' \in [\![\mathsf{r}]\!]\sigma \qquad \text{or, equivalently,} \qquad [\![\overleftarrow{\mathsf{r}}]\!]\sigma' \triangleq \{\sigma \mid \sigma' \in [\![\mathsf{r}]\!]\sigma\}. \tag{3}$$

$$\frac{}{\vdash [ok : P]\ \mathsf{x}\ :=\ \mathsf{a}\ [ok : [\![\mathsf{x}\ :=\ \mathsf{a}]\!]P]}\ [\text{assign}] \qquad \frac{}{\vdash [ok : P]\ \mathsf{b?}\ [ok : P \wedge b]}\ [\text{assume}]$$

$$\frac{}{\vdash [ok : P]\ \mathsf{x}\ :=\ \mathsf{nondet()}\ [ok : \exists x.P]}\ [\text{nondet}] \qquad \frac{}{\vdash [ok : P]\ \mathsf{skip}\ [ok : P]}\ [\text{skip}]$$

$$\frac{\vdash [P_1]\ \mathsf{r}\ [Q_1] \quad \vdash [P_2]\ \mathsf{r}\ [Q_2]}{\vdash [P_1 \vee P_2]\ \mathsf{r}\ [Q_1 \vee Q_2]}\ [\text{disj}] \qquad \frac{P \impliedby P' \quad \vdash [P']\ \mathsf{r}\ [Q'] \quad Q' \impliedby Q}{\vdash [P]\ \mathsf{r}\ [Q]}\ [\text{cons}]$$

$$\frac{\vdash [P]\ \mathsf{r}_1\ [R] \quad \vdash [R]\ \mathsf{r}_2\ [Q]}{\vdash [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}\ [\text{seq}] \qquad \frac{}{\vdash [er : P]\ \mathsf{r}\ [er : P]}\ [\text{er-id}]$$

$$\frac{\vdash [P]\ \mathsf{r}_1\ [Q]}{\vdash [P]\ \mathsf{r}_1 \boxplus \mathsf{r}_2\ [Q]}\ [\text{choiceL}] \qquad \frac{\vdash [P]\ \mathsf{r}_2\ [Q]}{\vdash [P]\ \mathsf{r}_1 \boxplus \mathsf{r}_2\ [Q]}\ [\text{choiceR}]$$

$$\frac{}{\vdash [P]\ \mathsf{r}^*\ [P]}\ [\text{iter0}] \qquad \frac{\vdash [P]\ \mathsf{r}^*;\mathsf{r}\ [Q]}{\vdash [P]\ \mathsf{r}^*\ [Q]}\ [\text{unroll}]$$

Fig. 2. Incorrectness Logic rules for regular commands [O'Hearn 2020]

As before, we additively lift the definition of backward semantics to set of states by union. Roughly speaking, $[\![\overleftarrow{\mathsf{r}}]\!]S$ is the set of input states that can reach some output state in S.[1]

2.2 Assertion Language

In the paper, we interpret assertions as sets of states. They are described by the following grammar:

$$\text{Asl} \ni P, Q ::= P \implies Q \mid \exists x.P \mid \mathsf{b} \mid [\![\mathsf{r}]\!]P \mid [\![\overleftarrow{\mathsf{r}}]\!]P$$

Encoding of other logical connectives is standard (eg. $\neg P \triangleq P \implies \text{false}$, note that false is part of the syntax of b). We include in our assertion language constructors for the collecting semantics. While this is theoretically sound, an implementation requires an equivalent closed formula for the semantics, which may or may not be available depending on the command r. For instance, there are such closed formulae for both the forward and backward semantics of all atomic commands in our language:

$$[\![\mathsf{x}\ :=\ \mathsf{a}]\!]P \equiv \exists v.P[v/x] \wedge x = a[v/x] \qquad [\![\overleftarrow{\mathsf{x}\ :=\ \mathsf{a}}]\!]Q \equiv Q[a/x]$$

$$[\![\mathsf{b?}]\!]P \equiv P \wedge b \qquad [\![\overleftarrow{\mathsf{b?}}]\!]Q \equiv Q \wedge b$$

$$[\![\mathsf{skip}]\!]P \equiv P \qquad [\![\overleftarrow{\mathsf{skip}}]\!]Q \equiv Q$$

$$[\![\mathsf{x}\ :=\ \mathsf{nondet()}]\!]P \equiv \exists x.P \qquad [\![\overleftarrow{\mathsf{x}\ :=\ \mathsf{nondet()}}]\!]Q \equiv \exists x.Q$$

where v is a fresh variable (i.e., v does not appear in P, x or a). Note that the formulae for assignment are precisely the forward transformer of Floyd's axiom for the forward semantics and Hoare's backward substitution for the backward semantics. In the rest of the paper we will often use $[\![\mathsf{x}\ :=\ \mathsf{a}]\!]P$ instead of the more verbose $\exists v.P[v/x] \wedge x = a[v/x]$.

We observe the following relation between forward and backward semantics of assignments.

Lemma 2.1. *Given an assertion P, define $Q \triangleq [\![\mathsf{x}\ :=\ \mathsf{a}]\!]P$. Then $P \implies [\![\overleftarrow{\mathsf{x}\ :=\ \mathsf{a}}]\!]Q = Q[a/x]$.*

2.3 Incorrectness Logic

Incorrectness Logic (IL) was first introduced in O'Hearn [2020] as a foundation for formal methods aimed to prove program *incorrectness* rather than correctness. The idea of IL is to consider a *subset* of program behaviors rather than a superset. This way, any erroneous behavior identified by the analysis is intrinsic to the program and not a false alarm induced by the approximation. The ability to consider only subsets of the behaviors by dropping disjuncts and bounded loop unrolling enables scalability at the expense of precision, a worth trade-off in many industrial settings [Godefroid 2005].

Formally, the validity of an IL triple $[P]\ \mathsf{r}\ [Q]$ is defined by the under-approximation condition $[\![\mathsf{r}]\!]P \supseteq Q$, which is equivalent to

$$\forall \sigma' \in Q. \exists \sigma \in P. \sigma' \in [\![\mathsf{r}]\!]\sigma\ .$$

In other words, any state σ' in the postcondition Q is reachable by a real execution of the program starting from some state in P, so that all the bugs in Q are reachable.

A hallmark of proof systems based on IL is to tag post, but not pre, with a flag to distinguish between normal and erroneous termination, respectively flagged using the green marker ok and the red marker er. In this paper, we instead follow the approach of Bruni et al. [2021, § 6] (see also Ascari et al. [2025a, Remark 3.9]): instead of attaching flags solely to the postconditions of triples, we enrich the entire state space with them. This is reflected as well in the assertion language, and we assume the semantics of any command acts as the identity on er-roneous states, i.e., $[\![\mathsf{r}]\!](er : \sigma) = er : \sigma$ for any $\mathsf{r} \in \mathsf{RCmd}$ and $\sigma \in \Sigma$. The benefit of this approach is a more uniform treatment of flags, but it does not introduce any conceptual difference. This leads us to the modified proof system in Fig. 2. Untagged assertions P, Q can contain any disjunction of ok and er states, while tagged assertions $ok : P$ and $er : P$ can only contain states with the specified tag. Axioms for atomic commands are obtained from IL by forcing the pre to only contain ok states. The only new rule is [er-id], that reflects the identity semantics of any command on er states.

As usual, we write $\vDash [P]\ \mathsf{r}\ [Q]$ for a valid triple and $\vdash [P]\ \mathsf{r}\ [Q]$ for a provable one.

Theorem 2.2 (IL soundness [O'Hearn 2020]). *Any provable IL triple is valid:*

$$\vdash [P]\ \mathsf{r}\ [Q] \implies \vDash [P]\ \mathsf{r}\ [Q]\ .$$

2.4 Sufficient Incorrectness Logic

While IL only finds true bugs in the post, it does not guarantee anything about the states in the pre. Particularly, thanks to rule [cons], it is always possible to weaken the pre to include states unrelated to the bugs found in the post. This limitation was acknowledged (sometimes less explicitly), e.g., in Ascari et al. [2025a]; Le et al. [2022]; Zilberstein et al. [2023]. A proposed solution is a logic that constrains the pre instead of the post, with the meaning that every state in the pre can reach at least one state in the post. Such a logic had multiple names in the literature, e.g., Lisbon logic [O'Hearn 2020; Zilberstein et al. 2023], backward under-approximate triples [Le et al. 2022; Möller et al. 2021; Raad et al. 2024a], or Sufficient Incorrectness Logic [Ascari et al. 2025a].

In this paper, we are interested in combining forward and backward under-approximation, therefore we take inspiration from the presentation in Ascari et al. [2025a], but we enrich their rules with error flags to better match the IL rules (these changes were already sketched in Ascari et al. [2025a, § 5.6]). The resulting proof system is in Fig. 3. Note that the IL and SIL proof systems have remarkably similar structural rules: the only differences are the rules of consequence and the infinitary rule for iteration ([BackwardsVariant] in O'Hearn [2020] for IL, $\langle \mathsf{iter} \rangle$ in Ascari et al.

[1]This was first presented by Hoare [1978, §5.3] as the weakest possible precondition calculus. Note that this definition is different from Dijkstra's weakest (liberal) precondition [Dijkstra 1975]

$$\frac{}{\vdash \langle ok : Q[a/x]\rangle\ \mathsf{x}\ :=\ \mathsf{a}\ \langle ok : Q\rangle}\ \langle \text{assign}\rangle \qquad \frac{}{\vdash \langle ok : Q \wedge b\rangle\ \mathsf{b?}\ \langle ok : Q\rangle}\ \langle \text{assume}\rangle$$

$$\frac{}{\vdash \langle ok : \exists x.Q\rangle\ \mathsf{x}\ :=\ \mathtt{nondet()}\ \langle ok : Q\rangle}\ \langle \text{nondet}\rangle \qquad \frac{}{\vdash \langle ok : Q\rangle\ \mathtt{skip}\ \langle ok : Q\rangle}\ \langle \text{skip}\rangle$$

$$\frac{\vdash \langle P_1\rangle\ \mathsf{r}\ \langle Q_1\rangle \quad \vdash \langle P_2\rangle\ \mathsf{r}\ \langle Q_2\rangle}{\vdash \langle P_1 \vee P_2\rangle\ \mathsf{r}\ \langle Q_1 \vee Q_2\rangle}\ \langle \text{disj}\rangle \qquad \frac{P \implies P' \quad \vdash \langle P'\rangle\ \mathsf{r}\ \langle Q'\rangle \quad Q' \implies Q}{\vdash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle}\ \langle \text{cons}\rangle$$

$$\frac{\vdash \langle P\rangle\ \mathsf{r}_1\ \langle R\rangle \quad \vdash \langle R\rangle\ \mathsf{r}_2\ \langle Q\rangle}{\vdash \langle P\rangle\ \mathsf{r}_1; \mathsf{r}_2\ \langle Q\rangle}\ \langle \text{seq}\rangle \qquad \frac{}{\vdash \langle er : P\rangle\ \mathsf{r}\ \langle er : P\rangle}\ \langle \text{er-id}\rangle$$

$$\frac{\vdash \langle P\rangle\ \mathsf{r}_1\ \langle Q\rangle}{\vdash \langle P\rangle\ \mathsf{r}_1 \boxplus \mathsf{r}_2\ \langle Q\rangle}\ \langle \text{choiceL}\rangle \qquad \frac{\vdash \langle P\rangle\ \mathsf{r}_2\ \langle Q\rangle}{\vdash \langle P\rangle\ \mathsf{r}_1 \boxplus \mathsf{r}_2\ \langle Q\rangle}\ \langle \text{choiceR}\rangle$$

$$\frac{}{\vdash \langle P\rangle\ \mathsf{r}^*\ \langle P\rangle}\ \langle \text{iter0}\rangle \qquad \frac{\vdash \langle P\rangle\ \mathsf{r}^*; \mathsf{r}\ \langle Q\rangle}{\vdash \langle P\rangle\ \mathsf{r}^*\ \langle Q\rangle}\ \langle \text{unroll}\rangle$$

Fig. 3. Sufficient Incorrectness Logic rules for regular commands [Ascari et al. 2025a]

[2025a] for SIL), but the latter is omitted in this paper. This will allow us to follow derivations in one proof systems using the other one.

Validity of a SIL triple $\langle P\rangle\ \mathsf{r}\ \langle Q\rangle$ is defined by the equation $[\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P$, which is equivalent to

$$\forall \sigma \in P. \exists \sigma' \in Q. \sigma' \in [\![\mathsf{r}]\!]\sigma\ .$$

Again, we write $\vdash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle$ for a provable triple and $\vDash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle$ for a valid one.

Theorem 2.3 (SIL soundness [Ascari et al. 2025a]). *Any provable SIL triple is valid:*

$$\vdash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle \implies \vDash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle\ .$$

We conclude observing the following simple facts about valid SIL and IL triples

Lemma 2.4. *For any regular command* r *and any assertions P and Q, it holds:*

(1) *If* $\vDash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle$ *and* $Q = \varnothing$, *then* $P = \varnothing$
(2) *If* $\vDash [P]\ \mathsf{r}\ [Q]$ *and* $P = \varnothing$, *then* $Q = \varnothing$

Corollary 2.5. *For any regular command* r *and any assertions P and Q, it holds:*

(1) *If* $\vDash \langle P\rangle\ \mathsf{r}\ \langle Q\rangle$ *and* $P \neq \varnothing$, *then* $Q \neq \varnothing$
(2) *If* $\vDash [P]\ \mathsf{r}\ [Q]$ *and* $Q \neq \varnothing$, *then* $P \neq \varnothing$

2.5 Separation Logic

In this section we give a brief primer on Separation Logic (see, eg., O'Hearn [2019] for an overview and O'Hearn et al. [2001] for a more technical explanation).

First, we augment the program syntax with primitives to operate on the heap. We consider a different set of heap atomic commands HACmd, and use them to obtain the full language of heap regular commands HRCmd:

$$\begin{aligned}\mathsf{HACmd} \ni \mathsf{c} ::= {} & \mathtt{skip} \mid \mathsf{x}\ :=\ \mathsf{a} \mid \mathsf{b?} \mid \mathsf{x}\ :=\ \mathtt{nondet()} \\ & \mid \mathsf{x}\ :=\ \mathtt{alloc()} \mid \mathtt{free(x)} \mid \mathsf{x}\ :=\ \mathtt{[y]} \mid \mathtt{[x]}\ :=\ \mathsf{y} \\ \mathsf{HRCmd} \ni \mathsf{r} ::= {} & \mathsf{c} \mid \mathsf{r}; \mathsf{r} \mid \mathsf{r} \boxplus \mathsf{r} \mid \mathsf{r}^*\end{aligned}$$

Roughly speaking, the semantics of heap regular commands is interpreted over sets of pairs Store $\times$ Heaplet. A *heaplet* is a partial function $h \in \mathsf{Heaplet} = (\mathbb{Z} \rightharpoonup \mathsf{Val} \uplus \{\delta\})$, where the input is interpreted as a memory address. Intuitively, a heaplet h only describes a *portion* of the global heap:

$$\frac{}{\vdash [ok : x = x' * y \mapsto e]\ \mathtt{x\ :=\ [y]}\ [ok : x = e[x'/x] * y \mapsto e[x'/x]]}\ [\mathsf{Load(ISL)}]$$

$$\frac{x \notin \mathrm{fv}(a)}{\vdash \langle ok : y \mapsto a * q[a/x]\rangle\ \mathtt{x\ :=\ [y]}\ \langle ok : y \mapsto a * q\rangle}\ \langle \mathsf{Load(SepSIL)}\rangle$$

$$\frac{}{\vdash [ok : x \not\mapsto]\ \mathtt{[x]\ :=\ y}\ [er : x \not\mapsto]}\ [\mathsf{StoreEr(ISL)}]$$

$$\frac{}{\vdash \langle ok : x \not\mapsto \rangle\ \mathtt{[x]\ :=\ y}\ \langle er : x \not\mapsto \rangle}\ \langle \mathsf{StoreEr(SepSIL)}\rangle$$

Fig. 4. Relevant excerpt of ISL [Raad et al. 2020] and Separation SIL [Ascari et al. 2025a] rules.

any location not in the domain of h is unknown (it may be not allocated or belong to a different heaplet); the special value δ denotes a known-to-be-deallocated location. We use notation $h[l \mapsto v]$ for function update (possibly adding l to the domain of h), $[]$ for the empty heaplet (ie. the heaplet with an empty domain) and a list notation $[l \mapsto v]$ as a shorthand for $[][l \mapsto v]$ (ie. the heaplet mapping l to v and undefined anywhere else).

The assertion language for Separation Logic is an instance of the logic of bunched implications [Pym et al. 2004]. We use the following grammar:

$$\mathrm{Asl} \ni P, Q ::= P \implies Q \mid \exists x.P \mid \mathsf{b} \mid [\![\mathsf{r}]\!]P \mid [\![\overleftarrow{\mathsf{r}}]\!]P \mid \mathbf{emp} \mid x \mapsto \mathsf{a} \mid x \not\mapsto \mid P * Q$$

The interpretation of spatial constructs ($\mathbf{emp}$, $\mapsto$, $\not\mapsto$ and $*$) is as follows. $\mathbf{emp}$ is valid on any state $(s, [])$, independently of the store s. $x \mapsto v$ is valid on any state $(s, [s(x) \mapsto v])$, where the heaplet contains only location $s(x)$: this means the memory address stored in variable x points to the value v. Similarly, $x \not\mapsto$ holds on $(s, [s(x) \mapsto \delta])$. Finally, the separation conjunction $P * Q$ holds on any state where the heaplet can be split in two sub-heaplets with *disjoint* domains, one satisfying P and the other satisfying Q. The disjointness condition ensures that only one of the two subformulae can take *ownership* of each location, and it is the key ingredient to enable distinguishing features of separation logics (eg. the frame rule).

Example 2.6. Consider the assertion $(\mathrm{true} * v \mapsto - * y \mapsto - * (x = y \vee x \not\mapsto))$ from Example 1.3. Using distribution laws of $*$ and $\vee$ we can rewrite it as

$$(\mathrm{true} * v \mapsto - * y \mapsto - * x = y) \vee (\mathrm{true} * v \mapsto - * y \mapsto - * x \not\mapsto)$$

On the one hand, the first disjunct explicitly says that x and y are aliased. On the other hand, the other disjunct implicitly says that they are *not* aliased: the separate conjunction $(y \mapsto - * x \not\mapsto)$ ensures that the addresses stored in x and y are different. In fact, if they were the same location l, it would be impossible to split the heaplet in such a way that l is in the domains of both the (disjoint) sub-heaplets satisfying $y \mapsto -$ and $x \not\mapsto$, respectively.

Both IL and SIL have been extended to a separation counterpart, Incorrectness Separation Logic and Separation SIL, respectively. They validate the same axioms as their non-separation counterparts, together with rules for the new atomic commands (an excerpt is in Fig. 4) and the frame rule:

$$\frac{\vdash [p]\ \mathsf{c}\ [q] \quad \mathrm{comp}(\mathsf{c}, f)}{\vdash [p * f]\ \mathsf{c}\ [q * f]}\ [\mathsf{Frame(ISL)}] \qquad \frac{\vdash \langle p\rangle\ \mathsf{c}\ \langle q\rangle \quad \mathrm{comp}(\mathsf{c}, f)}{\vdash \langle p * f\rangle\ \mathsf{c}\ \langle q * f\rangle}\ \langle \mathsf{Frame(SepSIL)}\rangle$$

where $\mathrm{comp}(\mathsf{c}, f)$ means that command c does not modify any of the free variables of assertion f.

2.6 UNTer

UNTer [Raad et al. 2024a] is a proof system inspired by IL aimed at proving the presence of (non)termination bug. To do so, alongside forward under-approximation (IL) triples, it introduces backward under-approximation (SIL) triples. One of the key observations is that forward and backward under-approximation share most of the structural rules, with the most notable exception being the rule of consequence. This leads to a simple automation of backward under-approximation, since it can just reuse most of the reasoning engine already implemented for forward under-approximation (via the use of indexed disjunctions and matched dropping, as described in Raad et al. [2024a, § 2, *Forward versus Backward Under-Approximate Triples*]). In particular, by presenting a *kernel* set of IL-inspired rules [Raad et al. 2024a, Fig. 1, $\vdash_\dagger$ proof system] that does not include the rule of consequence, UNTer details a proof system that can prove triples valid for both IL and SIL at the same time. Such a kernel set contains the same rules as the IL proof system in Fig. 2 with the following differences: the rule [cons] is replaced with dropping of indexed disjuncts and the rule [assume] is replaced by the rule

$$\frac{}{\vdash_\dagger [P \wedge b]\ \mathsf{b?}\ [P \wedge B]}\ \text{assume}$$

Similarly, the separation logic instance of UNTer uses the same axioms as ISL, except for the rule of consequence and [assume], changed as above.

UNTer kernel proof system is proved sound with respect to both validity as IL and SIL triples (only one at a time) [Raad et al. 2024a, Th. 7]. Moreover, when the consequence rule of IL (resp. SIL) is added to the kernel set of rules, the resulting proof systems becomes also complete for IL (resp. SIL) [Raad et al. 2024a, Th. 8]. To our knowledge, there is no completeness result concerning only the kernel set of rules with respect to triples that are valid both for IL and SIL at the same time.

3 Forward/Backward Axioms for Atomic Commands

In UNTer [Raad et al. 2024a, Fig. 7] the authors proves that IL (and ISL) axioms for atomic commands are also valid as SIL triples. However, these axioms are based on Incorrectness Separation Logic and hand-crafted. Therefore, it is natural to ask the following two questions.

(1) Are these axiom as general as possible?
(2) Is there a general procedure to derive axioms for new atomic commands, not relying on pre-existing ISL axioms?

The first question is partially answered in the positive by UNTer completeness result [Raad et al. 2024a, § 6]: since the resulting proof system is complete, each axiom together with the rule of consequence of IL (resp. SIL) is able to prove every valid IL (resp. SIL) triple for that particular atomic command. However, nothing is said about completeness with respect to triples that are both IL and SIL at the same time: is it possible to prove any such triple without resorting to the consequence rule of either logic (which can make the triple unsound for the other logic)?

We tackle this problem by addressing the second question. In more detail, we propose an axiom *schema* for atomic commands that is sound and complete for triples that are both IL and SIL. From this, we derive axioms for atomic commands that are sound and complete by construction, and we then show that we can derive such axioms in UNTer. However, our contribution goes beyond this: by providing this general schema, we give a methodology that can be applied to any atomic command. As an example, we will consider non-deterministic assignment, a command missing in UNTer, and derive a new axiom for it.

Proposition 3.1. *For every command* c *and assertions* P, Q, *the pre* $(P \wedge [\![\overleftarrow{\mathsf{c}}]\!]Q)$ *and the post* $(Q \wedge [\![\mathsf{c}]\!]P)$ *makes both a valid IL and SIL triple:*

$$\vDash [P \wedge [\![\overleftarrow{\mathsf{c}}]\!]Q]\ \mathsf{c}\ [Q \wedge [\![\mathsf{c}]\!]P] \qquad \wedge \qquad \vDash \langle P \wedge [\![\overleftarrow{\mathsf{c}}]\!]Q\rangle\ \mathsf{c}\ \langle Q \wedge [\![\mathsf{c}]\!]P\rangle$$

We show below some examples of applications of this schema. Note that we will often replace the (forward or backward) semantics with the equivalent formula from Section 2.2.

Example 3.2. For assignments, the above schema yields the axiom:

$$[P \wedge Q[a/x]]\ \mathtt{x\ :=\ a}\ [Q \wedge [\![\mathtt{x\ :=\ a}]\!]P]\ .$$

This is equivalent to the UNTer/IL axiom, that is precisely Floyd's forward axiom. Setting $Q = \text{true}$ in our axiom yields precisely the UNTer axiom. Conversely, substituting P with $(P \wedge Q[a/x])$ in the UNTer axiom yields ours, after some equivalence-preserving transformation of the formula in the post.

For Boolean guards, the above schema yields the axiom:

$$[P \wedge Q \wedge b]\ \mathtt{b?}\ [P \wedge Q \wedge b]\ .$$

Again, to derive the UNTer axiom $\vdash_\dagger [P \wedge b]\ \mathtt{b?}\ [P \wedge b]$ it suffices to take $Q = \text{true}$ in our axiom. Conversely, substituting P with $(P \wedge Q)$ in the UNTer axiom yields ours.

To show how our schema can be used to handle new constructs, we consider nondeterministic assignment, which was not explicitly discussed in UNTer.

Example 3.3. Recalling that for nondeterministic assignment

$$[\![\mathtt{x\ :=\ nondet()}]\!]P = \exists x.P \qquad\qquad [\![\overleftarrow{\mathtt{x\ :=\ nondet()}}]\!]Q = \exists x.Q$$

we obtain the axiom

$$[P \wedge \exists x.Q]\ \mathtt{x\ :=\ nondet()}\ [Q \wedge \exists x.P]$$

that is new and stronger than other proposals. In UNTer, nondeterministic assignments are not present in the programming language. IL and SIL use, respectively, the axioms

$$\vdash [P]\ \mathtt{x\ :=\ nondet()}\ [\exists x.P] \qquad\qquad \vdash \langle \exists x.Q\rangle\ \mathtt{x\ :=\ nondet()}\ \langle Q\rangle$$

which can be combined in

$$[\exists x.P]\ \mathtt{x\ :=\ nondet()}\ [\exists x.P]$$

However, this latter axiom is weaker than the proposal obtained with our methodology. For instance, it cannot be exploited to prove the triple $[\text{true}]\ \mathtt{x\ :=\ nondet()}\ [x > 0]$.

Interestingly, ISL [Raad et al. 2020] uses the axiom [havoc]

$$\vdash [x = n]\ \mathtt{x\ :=\ nondet()}\ [x = m]$$

that is equivalent to ours. In fact, assuming n and m are free names in P, Q, we can prove our axiom from [havoc] (together with [frame] and [exist]) with the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[x = n]\ \mathtt{x\ :=\ nondet()}\ [x = m]}\,[\mathsf{havoc}]}{[x = n \wedge P[n/x]]\ \mathtt{x\ :=\ nondet()}\ [x = m \wedge P[n/x]]}\,[\mathsf{frame}]}{[\exists n.(x = n \wedge P[n/x])]\ \mathtt{x\ :=\ nondet()}\ [\exists n.(x = m \wedge P[n/x])]}\,[\mathsf{exist}]}{[P \wedge Q[m/x]]\ \mathtt{x\ :=\ nondet()}\ [x = m \wedge \exists n.P[n/x] \wedge Q[m/x]]}\,[\mathsf{frame}]}{[P \wedge \exists m.Q[m/x]]\ \mathtt{x\ :=\ nondet()}\ [Q \wedge \exists n.P[n/x]]}\,[\mathsf{exist}]$$

Conversely, we can derive [havoc] from our axiom just by taking $P = (x = n)$ and $Q = (x = m)$. More abstractly, we know the valid triple $[x = n]\ \mathtt{x\ :=\ nondet()}\ [x = m]$ (as well as any other valid triple) is derivable by completeness of our axiom schema (Proposition 3.5 below).

Example 3.4. Let us consider an artificial example to show how we can apply our schema to new expressions. Consider a new atomic command `x++?` that nondeterministically can opt to increment `x` or leave it unchanged. Semantically, it is equivalent to the code `skip ⊞ (x := x+1)`. From this, we derive its forward and backward semantics

$$[\![\texttt{x++?}]\!]P = P \vee P[x-1/x] \qquad [\![\overleftarrow{\texttt{x++?}}]\!]Q = Q \vee Q[x+1/x]$$

so that our schema readily yields the axiom

$$[P \wedge (Q \vee Q[x+1/x])]\ \texttt{x++?}\ [Q \wedge (P \vee P[x-1/x])]$$

This axiom schema is also complete. To prove this, we show that any triple valid for both IL and SIL can be rewritten as $[P \wedge [\![\overleftarrow{\texttt{c}}]\!]Q]\ \texttt{c}\ [Q \wedge [\![\texttt{c}]\!]P]$, thus being provable with our axiom schema.

Proposition 3.5. *For every command* c *and sets of states P, Q, if both the IL triple* $\vDash [P]\ \texttt{c}\ [Q]$ *and the SIL triple* $\vDash \langle P\rangle\ \texttt{c}\ \langle Q\rangle$ *are valid, then both* $P \wedge [\![\overleftarrow{\texttt{c}}]\!]Q = P$ *and* $Q \wedge [\![\texttt{c}]\!]P = Q$.

3.1 Heap-Manipulating Axioms

The result in the previous section considers a simple, imperative language. In theory, the approach can be extended directly to heap-manipulating commands by changing the semantics. However, this approach does not take into account the locality principle of separation logic, according to which one should define *small* axioms—whose pre- and postconditions deal with the minimal amount of information needed to execute the command—that can be extended by need to larger heaps thanks to a suitable *frame* rule, the hallmark of separation logics.

To recover local axioms, we can consider a local semantics $[\![\cdot]\!]_L$ instead of the global $[\![\cdot]\!]$. We define such a semantics based on the relation **foot** in Raad et al. [2020, § 4.1]. Intuitively, **foot**(c) relates a pair of states $(s,h), (s',h')$ if executing c starting from (s,h) can yield the final state (s',h') and h is a minimal heaplet allowing for such an execution. In other words, if we remove any location from h then the command c can no longer execute from the reduced state.

We can then define the local semantics $[\![\texttt{c}]\!]_L$ as the functional version of the **foot** relation: $[\![\texttt{c}]\!]_L\,(s,h) = \{(s',h') \mid ((s,h),(s',h')) \in \textbf{foot}(\texttt{c})\}$ and then extended by additivity to sets of states. Leveraging their footprint theorem [Raad et al. 2020, Th. 2], we obtain an analogous decomposition of the (global) semantics in terms of local semantics and frames:

Proposition 3.6. *For any command* c, *assertions P, R such that* $[\![\texttt{c}]\!]_L\,\sigma$ *is defined for every* $\sigma \in P$

$$[\![\texttt{c}]\!](P * R) = ([\![\texttt{c}]\!]_L\,P) * R$$

From this, we obtain a "local axiom schema" for heap manipulating commands:

Proposition 3.7. *For every command* c *and assertions P, Q, the pre* $(P \wedge [\![\overleftarrow{\texttt{c}}]\!]_L\,Q)$ *and the post* $(Q \wedge [\![\texttt{c}]\!]_L\,P)$ *makes both a valid ISL and Separation SIL triple:*

$$\vDash [P \wedge [\![\overleftarrow{\texttt{c}}]\!]_L\,Q]\ \texttt{c}\ [Q \wedge [\![\texttt{c}]\!]_L\,P] \qquad \wedge \qquad \vDash \langle P \wedge [\![\overleftarrow{\texttt{c}}]\!]_L\,Q\rangle\ \texttt{c}\ \langle Q \wedge [\![\texttt{c}]\!]_L\,P\rangle$$

The proof is identical to that of Proposition 3.1 by recalling that $[\![\overleftarrow{\texttt{c}}]\!]_L$ is a subset of $[\![\overleftarrow{\texttt{c}}]\!]$. Moreover, the locality of $[\![\overleftarrow{\cdot}]\!]_L$ forces locality in the axiom thanks to the conjunction $\wedge$: even if, for instance, P talks about locations outside the footprint of c, these are filtered out by the $[\![\overleftarrow{\texttt{c}}]\!]_L\,Q$ conjunct in the precondition, forcing locality.

As an example, we apply our schema to derive the axiom for a load command.

Example 3.8. Consider a load command `x := [y]`. Its local forward and backward semantics are:

$$[\![\texttt{x := [y]}]\!]_L\,(y \mapsto v \wedge P) = y \mapsto v \wedge \exists z.(P[z/x] \wedge x = v)$$
$$[\![\overleftarrow{\texttt{x := [y]}}]\!]_L\,(y \mapsto v \wedge Q) = y \mapsto v \wedge Q[v/x]$$

Note the additional conjunct $y \mapsto v$ in the input states: this ensures that the heap(let) is only defined on the location pointed by y, that is exactly the footprint of the load statement.

Our schema applied to these semantics yields the triple

$$[P \wedge (y \mapsto v \wedge Q[v/x])]\ \texttt{x := [y]}\ [Q \wedge (y \mapsto v \wedge \exists z.(P[z/x] \wedge x = v))]$$

which can be simplified to

$$[y \mapsto v \wedge P \wedge Q[v/x]]\ \texttt{x := [y]}\ [y \mapsto v \wedge x = v \wedge Q \wedge \exists z.P[z/x]]$$

From this, we can recover the load axiom in ISL/UNTer [Raad et al. 2024b, Fig. 10] (both use the same axiom). To do so, we first instance our axiom by taking $Q \triangleq (\text{true})$ and $P \triangleq (x = x' \wedge v = e)$, and then use rule [exists] to hide v. The precondition simplifies as

$$\begin{aligned} & \exists v.(y \mapsto v \wedge (x = x' \wedge v = e) \wedge (\text{true})[v/x]) \\ \equiv\ & \exists v.(y \mapsto e \wedge x = x' \wedge v = e) \\ \equiv\ & y \mapsto e \wedge x = x' \end{aligned}$$

and the postcondition simplifies as

$$\begin{aligned} & \exists v.(y \mapsto v \wedge x = v \wedge (\text{true}) \wedge \exists z.(x = x' \wedge v = e)[z/x]) \\ \equiv\ & \exists v.(y \mapsto v \wedge x = v \wedge \exists z.(z = x' \wedge v = e[z/x])) \\ \equiv\ & \exists v.(y \mapsto v \wedge x = v \wedge v = e[x'/x]) \\ \equiv\ & y \mapsto e[x'/x] \wedge x = e[x'/x] \end{aligned}$$

thus recovering exactly the UNTer axiom.

4 U-Turn: Following IL Derivations with SIL

The approach in the previous section provides a technique to derive axioms that are valid in both IL and SIL. Paired with UNTer, it enables a single analysis resulting in a triple with a double guarantee: all errors in the post are reachable from states in the pre and all states in the pre can lead to some error in the post. However, our axiom schema requires previous knowledge of *both* P and Q, that is both the pre and the post of the expected triple or at least some over-approximation of them. Since the analysis typically follows the control flow either in the forward or backward direction, it is often the case that only one of the two is available (the pre in a forward analysis and the post in a backward one). A possible solution would be to use a default value (such as true) for the unknown pre- or postcondition.

In this section, we tackle the problem from a completely different angle. Instead of doing a single analysis, whose result is valid both for IL and SIL but that is tied to either the forward or backward flow in its computation, we perform two consecutive analyses. We start with a forward, IL-based analysis, and then we trace it backward using SIL principles. In doing so, we take advantage of the information discovered during the forward analysis. We call **U-Turn** the resulting proof system.

Intuitively, each IL derivation outlines those *code paths* that have been explored to find the result. For instance, if the proof uses [choiceL] to analyze an `if` statement, it means that we are only considering the then-branch path in the code, dropping the analysis of the else-branch. Similarly, usage of rules [iter0] and [unroll] details how many loop unrolling have been performed. This information is incredibly valuable for a backward step with SIL, because it guides the proof search down paths that are *guaranteed to succeed*. While this strategy does not ensure completeness by itself (i.e., we may still miss some sources of the errors in the post) it is often useful to report *some* sufficient preconditions for the errors rather than aiming to collect all of them.

To formally develop this idea, we consider U-Turn judgments of the form

$$\frac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \ \vdash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$$

where d is a *proof tree*, built from the rules in Fig. 2, for the provable IL triple $\vdash [P]\ \mathsf{r}\ [Q]$. With a slight abuse of notation, when d is a proof tree whose conclusion is $\vdash [P]\ \mathsf{r}\ [Q]$, we often highlight such conclusion explicitly for clarity in U-Turn judgments. As discussed above, the derivation d contains useful information on the code paths that is not summarized in the final IL triple. We will discuss how such reasoning is implemented by some U-turn rules.

Definition 4.1 (Judgment validity). Given a proof tree d for the provable IL triple $\vdash [P]\ \mathsf{r}\ [Q]$ (using the IL proof system in Fig. 2), we say that the U-Turn judgment $\frac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \ \vdash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$ is *valid*, written $\frac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \ \vDash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$, if

(1) $\vDash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$,
(2) $P' \subseteq P$,
(3) $Q' \subseteq Q$,
(4) either $P' = Q' = \emptyset$, or both $P', Q' \neq \emptyset$.

A valid U-Turn judgment entails the validity of the corresponding SIL triple $\langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$ (condition (1)) and that both P' and Q' are subsets of the corresponding IL pre/posts (conditions (2) and (3)). In the post, this inclusion means we are allowed to only focus on a subset Q' of the states found in the post Q. This freedom is primarily a technical requirement used within derivations, rather than a mechanism for discarding errors found by the IL analysis, for instance to drop some non-interesting ok states. This requirement is immediately evident when considering, e.g., the analysis of two consecutive code fragments: given the IL derivation:

$$\frac{\dfrac{d_1}{\vdash [P]\ \mathsf{r}_1\ [R]} \quad \dfrac{d_2}{\vdash [R]\ \mathsf{r}_2\ [Q]}}{\vdash [P]\ \mathsf{r}_1; \mathsf{r}_2\ [Q]}$$

we use SIL to trace back the sources of errors in Q that reside in R w.r.t. executing r_2, which may lead to a proper subset $R' \subset R$ of IL postcondition for r_1 in

$$\frac{d_1}{\vdash [P]\ \mathsf{r}_1\ [R]}\ .$$

Hence the need to be able, inductively, to start the inference process in SIL along r_1 starting from any subset R' of R rather than from R itself. Condition (4) is a bit more involved. If Q' is not empty we care about reachability of some final state in Q, and forcing P' to be non-empty means we find (some) states in P that surely lead to errors in Q. If instead Q' is empty it means we are not considering any of the states found by the IL analysis in the post Q, so by taking $P' = \emptyset$ we ignore completely the program path that ends at Q'. This gives us the freedom to drop some of the code paths explored in the IL triple if we deem them not interesting. Formally, condition (4) is justified by Lemma 2.4.1: if Q' is empty, P' must be empty as well since we require $\vDash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$.

As a main contribution, we define the U-Turn proof system for such judgments. A relevant excerpt on which we focus is given in Fig. 5. Most U-Turn rules can only be applied when the IL derivation d ends with the application of a specific IL rule, and in that case they share its name. This constraint means that the SIL derivation in the U-Turn proof system will mimic the IL one.

Rule [assign⟩ can only be applied when IL uses its own axiom [assign]. It allows one to take any subset of the strongest post and go backward from there, then conjoin it with P to ensure the

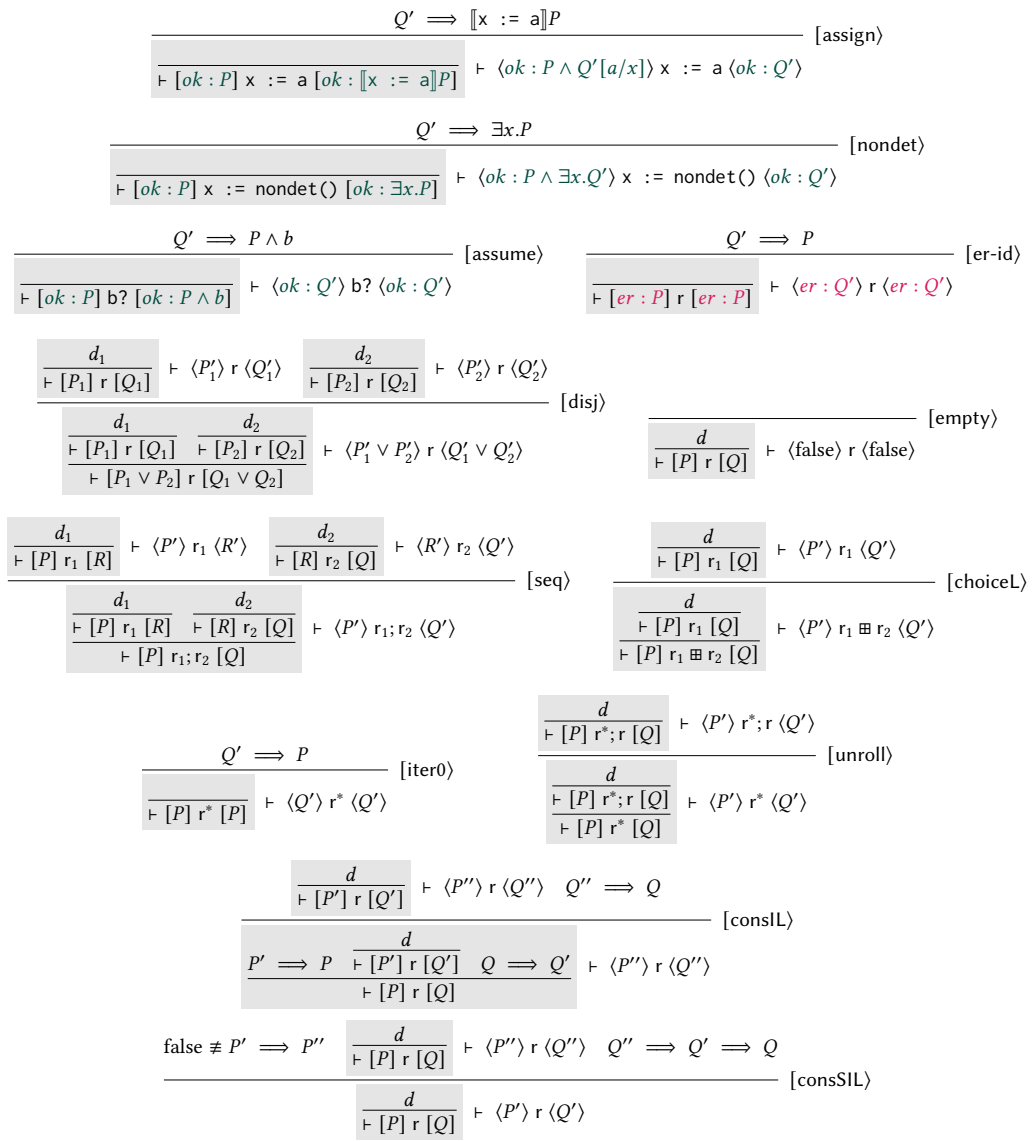


Fig. 5. The U-Turn proof system (excerpt). The full proof system can be found in Ascari et al. [2025b].

$P' \subseteq P$ validity condition of the triple. Rules [nondet⟩ and [assume⟩ work similarly, and analogous rules are available for other atoms.

If the IL derivation exploited [disj] to split the precondition in two disjuncts and analyze them separately, U-Turn requires the SIL step to do the same: it analyzes the two resulting post Q_1 and Q_2 separately and then joins the results. Similarly, whenever the IL derivation composes sequentially two sub-proofs using [seq], U-Turn forces SIL to do the same with the rule [seq⟩. Moreover, if IL found that the error originated before r and just propagated it through the command with [er-id], the corresponding U-Turn rule [er-id⟩ propagates backward the error as-is.

```
// program rlen
s := init(l);
l := 100;
i := len(s)
```

```
init(l) {
  s := alloc(l + 1);
  // initialize s[0..l-1]
  if (l != 3) {
    s[l] := "\0";
  }
  return s
}
```

```
len(s) {
  i := 0;
  while (s[i] != "\0") {
    i := i + 1;
  }
  return i
}
```

Fig. 6. The program rlen discussed in Example 4.3.

Rule [empty⟩ is peculiar in that it can be applied regardless of the IL derivation. However, it can only be applied when the post is false, and it derives the (only) valid pre false. Intuitively, this corresponds to not analyzing r when we do not care about any state it can reach.

Rules [choiceL⟩, [iter0⟩ and [unroll⟩ show how U-Turn forces SIL to follow the same code paths analyzed by IL. In all three rules, if IL decides to follow a specific code path (the left branch in an if, skip a loop or unroll it once) then SIL is forced to follow the exact same path.

If the IL derivation contains any application of its rule of consequence [cons], U-Turn skips it with [consIL⟩: when using the proof system for a backward analysis, the constraint $Q'' \implies Q$ will always be satisfied because Q'' will be the pre in some subsequent code fragment whose pre in the IL triple is Q. Intuitively, since IL rule of consequence does not change the explored code paths, it is not relevant for U-Turn.

Lastly, at any point in the derivation, U-Turn can use SIL rule of consequence via [consSIL⟩, provided it does not weaken the triple so much that it breaks one of the validity conditions of U-Turn. As we will discuss later, if rule [consSIL⟩ is never used, then we will have stronger guarantees about the result of the analysis (see Theorem 4.7).

The U-Turn proof system is sound. The proof is a standard induction on the derivation tree.

Theorem 4.2 (Soundness). *Any provable U-Turn judgment is valid.*

We present now two examples of how U-Turn guides and helps the SIL proof search.

Example 4.3. For this example we consider the separation logic instance of both IL and SIL (ISL and Separation SIL respectively). Since structural rules are the same as IL and SIL, we only have to adapt atoms, which is straightforward using the corresponding atoms from the two logics and add the frame rule. We also assume both logics include arrays, the extension being straightforward (see, e.g., the treatment in Reynolds [2002]).

Consider the faulty program rlen in Fig. 6. First, it initializes a string s with length l. However, when $l = 3$, it misses the null terminator. Then, a client tries to compute the length of the string, iterating over it and looking for the null terminator. This makes the bug emerge whenever the initial value of l is 3, but this information is obfuscated after the assignment `l := 100`.

For presentation purposes, we write $\vec{a}$ for $a0, a1, a2, a3$ and $pv(n)$ for $(a0 \neq \text{"}\backslash 0\text{"} \wedge \cdots \wedge an \neq \text{"}\backslash 0\text{"})$. Since our syntax does not allow variable dereferencing in boolean expressions, we desugar the guard of the while-loop (`s[i] != "\0"`) using a temporary variable `si` that is assigned to `si := s[0]` before the loop and to `si := s[i]` inside it. Using ISL, we can prove the following triple as shown in Fig. 7.

$$[ok : \text{true}]\ \text{rlen}\ [er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge i = 4 \wedge pv(3)]$$

The ISL proof system finds the error: it considers the "else" branch of the if statement in `init` to find that the code path where $l = 3$ yields a string without the null terminator "\0", that later leads

$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 0 \wedge si = a0]$
`(si != "\0")?;`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 0 \wedge si = a0]$
`i := i + 1;`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 1 \wedge si = a0]$
`si := s[i]`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 1 \wedge si = a1]$

(a) Linearized ISL proof for the first iteration of the body of the while loop in `len`.

$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 3 \wedge si = a3]$
`(si != "\0")?;`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 3 \wedge si = a3]$
`i := i + 1;`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3]$
`si := s[i]`
$[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3]$

(b) Linearized ISL proof for the last iteration of the body of the while loop in `len`.

$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2)]$
`i := 0;`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 0]$
`si := s[i];`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 0 \wedge si = a0]$
r_b;
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 1 \wedge si = a1]$
r_b;
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 2 \wedge si = a2]$
r_b;
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 3 \wedge si = a3]$
r_b;
$[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3]$
`(si == "\0")?`
$[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3]$

(c) Sketch of the ISL proof for `len`. We call r_b the body of the while loop. Since by using [unroll] and [iter0] the proof unrolls the loop 4 times, we do the same here: hence, the four repetitions of r_b instead of r_b^*.

$[ok : \text{true}]$
`s := init(l);`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 3 \wedge pv(2)]$
`l := 100;`
$[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2)]$
`i := len(s);`
$[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge i = 4 \wedge pv(3)]$

(d) Sketch of the ISL proof for rlen. We hide the local variable *si* of `len` using ISL rule [local].

Fig. 7. Sketch of the ISL derivation for [true] rlen $[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge i = 4 \wedge pv(3)]$.

to the error in `len` by unrolling the while loop 3 times. However, the ISL triple does not highlight the cause of error, that is the condition $l = 3$ at the beginning of the program.

In theory, Separation SIL can find the source of this error. However, doing so requires guessing the correct amount of unrolling for the while loop in `len`, since there is no indication in the post that 4 is the number of iterations: this information comes from an earlier program point, that Separation SIL has not explored yet.

$$
\begin{array}{lll}
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 3 \wedge si = a3] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 3 \wedge si = a3 \wedge si \neq \text{"\0"} \rangle \\
\downarrow \quad \texttt{(si != "\0")?;} & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 3 \wedge si = a3] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 3 \wedge si = a3 \rangle \\
\downarrow \quad \texttt{i := i + 1;} & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3 \rangle \\
\downarrow \quad \texttt{si := s[i]} & & \uparrow \\
[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3] & \rightarrow & \langle er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3 \rangle
\end{array}
$$

(a) Linearized U-Turn proof for the last iteration of the body of the while loop in `len`. Note that $si \neq$ "\0" in the first line is redundant since $si = a2$ and $pv(3)$ contains $a2 \neq$ "\0".

$$
\begin{array}{lll}
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2)] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge 0 = 0 \rangle \\
\downarrow \quad \texttt{i := 0;} & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 0] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 0 \wedge a0 = a0 \rangle \\
\downarrow \quad \texttt{si := s[i];} & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 0 \wedge si = a0] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 0 \wedge si = a0 \rangle \\
\downarrow \quad \mathsf{r}_b; & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 1 \wedge si = a1] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 1 \wedge si = a1 \rangle \\
\downarrow \quad \mathsf{r}_b; & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 2 \wedge si = a2] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 2 \wedge si = a2 \rangle \\
\downarrow \quad \mathsf{r}_b; & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2) \wedge i = 3 \wedge si = a3] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 3 \wedge si = a3 \rangle \\
\downarrow \quad \mathsf{r}_b; & & \uparrow \\
[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3] & & \langle er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3 \rangle \\
\downarrow \quad \texttt{(si == "\0")?} & & \uparrow \\
[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3] & \rightarrow & \langle er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge i = 4 \wedge pv(3) \rangle
\end{array}
$$

(b) Sketch of the U-Turn derivation for `len`. Following Fig. 7, we call r_b the body of the while loop and unroll it four times. This is enforced by the proof system since the IL derivation did the same.

$$
\begin{array}{lll}
[ok : \text{true}] & & \langle ok : l = 3 \rangle \\
\downarrow \quad \texttt{s := init(l);} & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 3 \wedge pv(2)] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge 100 = 100 \wedge pv(3) \wedge l = 3 \rangle \\
\downarrow \quad \texttt{l := 100;} & & \uparrow \\
[ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(2)] & & \langle ok : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge \text{len(s)} = 4 \wedge pv(3) \rangle \\
\downarrow \quad \texttt{i := len(s);} & & \uparrow \\
[er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge pv(3) \wedge i = 4 \wedge si = a3] & \rightarrow & \langle er : \exists \vec{a}.s \mapsto \vec{a} \wedge l = 100 \wedge i = 4 \wedge pv(3) \rangle
\end{array}
$$

(c) Sketch of the U-Turn proof for rlen.

Fig. 8. Sketch of the U-Turn derivation for `len`. We write it linearized, annotating program points with both the IL and the SIL assertion. The former are the same as Fig. 7. The latter are better read bottom-up and form the SIL triple obtained following the IL derivation that lead to the corresponding IL assertions.

In this example, the synergy between IL and SIL is essential. In fact, the ISL derivation unrolled the while loop exactly 4 times, because it knew the right number from the condition $l = 3$ in `init`. Separation SIL can thus exploit this information: by unrolling the loop 4 times, it finds exactly the error source, that is $l = 3$. This information sharing is formally captured by our combined proof system, whose derivation is shown in Fig. 8. To help readability, the arrows indicate the order of

```
// program r
x := [v];
push_back(v);
[x] := 42;
```

```
push_back(v) {
  ( y := [v];
    free(y);
    y := alloc();
    [v] := y; )
  ⊞ ( skip; )
}
```

Fig. 9. The program push_back discussed in Example 4.4.

deductions: first we perform a forward analysis of the code using ISL proof system (the flow of deduction is shown on the left hand side of the figures), which produced the assertions within square brackets, then, once the error is found, we use the proof system of U-Turn to derive SIL triples, i.e., the assertions within angle brackets, by backward analysis (accordingly, the flow of deduction is moved to the right hand side of the figures).

Note the introduction of some constraints in some SIL assertions to ensure that they are subsets of the corresponding ISL assertions. For instance, SIL would not require that $si = a3$ before the assignment `si := s[i]` in Fig. 8a, or that $l = 3$ before the assignment `l := 100`, but since the IL assertions prescribe these additional constraints, they appear in the SIL assertions too. This witnesses another way IL can transfer information to SIL, that we expand in the next example.

Example 4.4. Consider the push_back example in Fig. 9, already examined in both ISL [Raad et al. 2020] and Separation SIL [Ascari et al. 2025a] papers.

Roughly speaking, a ISL analysis can find an error in the assignment `[x] := 42` if it picks the left branch in push_back, where v gets reallocated. For Separation SIL to find such an error, it has not only to explore the same branch in push_back (the same code path), but also to guess that y is *aliased* to x. This (possible) aliasing can be detected automatically [Ascari et al. 2025a, §5.5], but the SIL backward analysis has no way to know which one is the right choice until earlier in the program (so later in the analysis). Therefore, it must consider both the cases where y and x are aliased and when they are not aliased. This is embodied by the disjunction $(x = z \vee x \not\mapsto)$ found in the precondition of push_back [Ascari et al. 2025a, Fig. 6]. Namely, the computed precondition is made of three disjuncts, corresponding to as many disjunct situations:

$$
\begin{array}{ll}
(\text{true} * v \mapsto z * z \mapsto - * x = z) \vee & \text{reallocation in push\_back, y and x aliased} \\
(\text{true} * v \mapsto z * z \mapsto - * x \not\mapsto) \vee & \text{reallocation in push\_back, y and x distinct} \\
(\text{true} * x \not\mapsto) & \text{no reallocation in push\_back}
\end{array}
$$

Note that only the last line correspond directly to a different program path than the others. However, the U-Turn proof system is able to share enough information between the two analyses to prune also the second disjunct. Intuitively, the ISL analysis contains the information that y and x are aliased in the assertions computed during the analysis (in a forward analysis it is easy to know that $v \mapsto x$ already when `y := [v]` is executed). The requirement that SIL assertions imply the IL assertions at the same program point forces this information transfer.

We focus on the left branch of push_back only, that we name r_b. We take the ISL derivation from Raad et al. [2020, Fig. 3] and apply U-Turn to it in Fig. 10. This proves the Separation SIL triple

$$\langle v \mapsto x * x \mapsto - \rangle \; \mathsf{r}_b \; \langle v \mapsto y * x \not\mapsto \; * y \mapsto - \rangle$$

as opposed to the triple

$$\langle \text{true} * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto) \rangle \; \mathsf{r}_b \; \langle x \not\mapsto \; * \text{true} \rangle$$

$$
\begin{array}{ll}
[v \mapsto x * x \mapsto -] & \langle v \mapsto x * x \mapsto - \rangle \\
\downarrow \ \texttt{y := [v];} & \uparrow \\
[v \mapsto x * x \mapsto - * y = x] & \langle v \mapsto x * x \mapsto - * y = x \rangle \\
\downarrow \ \texttt{free(y);} & \uparrow \\
[v \mapsto x * x \not\mapsto * y = x] & \langle v \mapsto x * x \not\mapsto * y = x \text{(instead of } y = x \vee y \not\mapsto \text{)} \rangle \\
\downarrow \ \texttt{y := alloc();} & \uparrow \\
[v \mapsto x * x \not\mapsto * y \mapsto -] & \langle v \mapsto x \text{(instead of } v \mapsto -\text{)} * x \not\mapsto * y \mapsto - \rangle \\
\downarrow \ \texttt{[v] := y;} & \uparrow \\
[v \mapsto y * x \not\mapsto * y \mapsto -] \quad \rightarrow & \langle v \mapsto y * x \not\mapsto * y \mapsto - \rangle
\end{array}
$$

Fig. 10. Sketch of the U-Turn derivation for push_back, linearized. We annotate program points with both the IL and the SIL assertion. The former should be read top down, the latter bottom-up. In the SIL assertions, we write in gray what we would obtain by plain application of the Separation SIL axioms, without the additional constraint to be a stronger assertion than the corresponding IL one.

Listing 1. Pseudocode of the UTurn algorithm.

```
UTurn :: ILProofTree -> Assertion -> Assertion
-- As stated in the theorem we assume that (false != Q') and (Q' -> Q)

-- Atomic commands
UTurn (ILAssign [P] x:=a [Q]) Q' = (P /\ Q'[a/x])
UTurn (ILAssume [P] b? [Q])   Q' = Q'
UTurn (ILNondet [P] x:=* [Q]) Q' = (P /\ ∃x. Q')
-- Structural rules
UTurn (ILSeq d1 d2 [P] r [Q]) Q' =
    let R' = UTurn d2 Q' in
    let P' = UTurn d1 R' in P'
UTurn (ILErId [P] r [Q]) Q' = Q'
UTurn (ILChoiceL d [P] r1 ⊞ r2 [Q]) Q' = UTurn d Q'
UTurn (ILIter0 [P] r* [Q]) Q' = Q'
UTurn (ILUnroll d [P] r* [Q]) Q' = UTurn d Q'
-- Other rules
UTurn (ILCons d' [P'] r [Q']) Q'' = let P'' = UTurn d' Q'' in P''
UTurn (ILDisj d [P] r [Q]) Q''    =
    let (d1 [P1] r [Q1]), (d2 [P2] r [Q2]) = d in
    let Q1' = Q' /\ Q1, Q2' = Q' /\ Q2 in
    let P1' = if Q1' == False then False else UTurn d1 Q1' in
    let P2' = if Q2' == False then False else UTurn d2 Q2' in P1' \/ P2'
```

from Ascari et al. [2025a, Fig. 6]. Particularly, in the triple returned by U-Turn there is only the disjunct where $x = z$, while Separation SIL alone must consider both, cluttering the analysis with useless disjuncts. This is possible due to the information in the IL assertion implicitly flowing into the SIL derivation via the $P' \implies P$ constraint in the soundness of the U-Turn judgment.

4.1 Progress and Automation

Since U-Turn must follow the IL derivation closely but imposes additional constraints, it is a non-trivial and practically relevant question whether or not it is always possible to complete a U-Turn proof given any IL derivation d. The next theorem not only answers in the affirmative, but

```
foo(b) {
  x := nondet();
  if (b ∧ x ≠ 0) { p := alloc() }
  else { p := null };
  [p] := x;
  return p
}
```

Fig. 11. The program foo discussed in Example 4.6.

also provides a high-level algorithm to do so. We call this algorithm UTurn, and we present it in Listing 1 (we omit cases for rules not in Fig. 5).

Theorem 4.5. *Given a derivation d for the IL triple $\vdash [P]\ \mathsf{r}\ [Q]$ and a Q' such that $\mathit{false} \not\equiv Q' \implies Q$, let $P' =$ UTurn d Q'. Then, $P' \not\equiv \mathit{false}$ and it holds:*

$$\frac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \ \vdash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$$

Theorem 4.5 guarantees that given any proof tree d for the IL triple $\vdash [P]\ \mathsf{r}\ [Q]$ and any (non-empty) subset of the errors $Q' \implies Q$, the UTurn algorithm always yields a $P' \not\equiv$ false such that the judgment $\dfrac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \ \vdash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$ is provable in U-Turn.

Roughly speaking, the UTurn algorithm inspects the last rule applied by the IL triple and applies the homonymous U-Turn rule, always processing the right subtree of [seq⟩ first. This produces a backward-fashioned derivation, where the post of the current rule is always provided by the previous step, and the pre is computed as prescribed by the applied rule. This inductive procedure solves the implicit nondeterminism related to which strengthened postcondition Q' to use in many U-Turn rules.

As a special case, it is worth mentioning that, given a single error state $\sigma \in Q$, it is always possible to find a non-empty P', i.e., some causes for reaching σ, by considering $Q' = \{\sigma\}$.

Note that in Examples 4.3 and 4.4 we basically applied the algorithm UTurn to perform the U-Turn derivations: this is because the algorithm follows naturally from the U-Turn rules, and therefore it gives the most natural (albeit not the only) way to use the proof system.

4.2 Following SIL Derivations with IL

In the previous sections, we presented U-Turn for following IL derivations backward with SIL. As anticipated, the reverse combination is also possible, namely to follow a SIL derivation forward with IL, obtaining a proof system that we informally call Turn-U. For brevity, we neither spell out the rules of Turn-U nor list the corresponding pseudo-code, because they are entirely dual to the ones in Fig. 5 and Listing 1. We show below how Turn-U can be useful with an example.

Example 4.6. Consider the procedure foo(b) in Fig. 11, where we use nondet() to model some opaque library call for which no summary is available. We want to produce a summary for foo telling us when it can cause errors, so we start with a SIL analysis from the postcondition $\langle er : \mathrm{true} \rangle$, and we obtain the precondition $\langle ok : b \rangle$ (some of the intermediate assertions are detailed in the combined derivation below). Unfortunately, this precondition is not informative enough: first, it

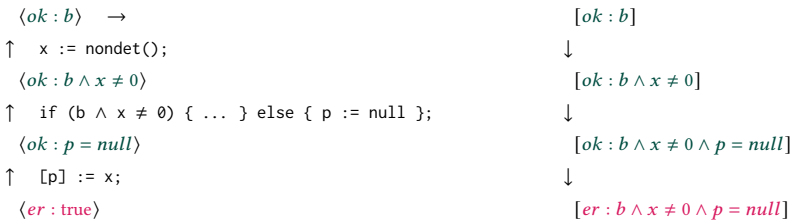


Fig. 12. Sketch of the Turn-U proof for the program foo.

does not describe precisely which errors can happen in `foo` and where; second, it does not say anything about the opaque library call. To fix these two issues, we trace SIL derivation forward using Turn-U. Note that the SIL analysis already found out that the error is caused by the else-branch of the `if`, so that IL can analyze only that branch instead of having to check both. Therefore, following the SIL analysis, we obtain the derivation in Fig. 12, where we elided the then-branch since it gets ignored:

Contrary to previous U-Turn derivations, the flow of deduction is now reversed, as illustrated by the arrows. The analysis started from the postcondition $er : \text{true}$ with the SIL derivation flowing bottom up (on the left hand side of the figure, along the assertions within angle brackets) until the precondition $ok : b$ is found and then flow is reversed by Turn-U (as shown on the right hand side of the figure, along the assertions within square brackets).

Note that the IL postcondition is very informative: not only it includes that $p = null$, the actual cause of the error (the attempt to dereference a null-pointer), but also that $x \neq 0$, therefore giving some information on the result of the (opaque) library call to reach the error. Note that SIL cannot encode it in the precondition because it refers to states *before* the library call and therefore cannot embed any information about the *output* of the call.

4.3 Relation with UNTer

We started this section by moving away from the perspective of directly proving triples that are valid for both IL and SIL. This lead us to first prove an IL triple, and then exploit its proof tree to derive a corresponding SIL triple. It turns out our approach remains close in spirit: if the U-Turn proof does not use SIL consequence rule, *the resulting triple is valid in both IL and SIL*.

Theorem 4.7. *If the judgment* $\dfrac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \vdash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$ *is provable without using rule* [consSIL⟩, *then* $\vDash [P']\ \mathsf{r}\ [Q']$.

We prove this theorem by showing that it is possible to follow again the U-Turn derivation with IL, in the spirit of what we discussed in the previous section. This suggest that it may be possible to obtain the same result even in the presence of SIL consequence rule by executing another forward step to ensure the triple obtained is valid in IL.

We can ensure this by modifying the U-Turn proof system to add a side condition on rule [consSIL⟩. The premises of [consSIL⟩ provide a SIL derivation for the triple $\langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$, on which we can apply the Turn-U proof system. This will find a triple $[P''']\ \mathsf{r}\ [Q''']$ with $P''' \implies P'$ and $Q''' \implies Q'$. By symmetry with U-Turn, if the Turn-U proof did not use the rule of consequences of IL ⟨consIL] (a dual of [consSIL⟩, where Turn-U can use the IL rule of consequence [cons] with some additional constraints) , the resulting triple is valid in SIL. We argue that, in practice, there is no need to use the rule ⟨consIL] in this latter Turn-U derivation. In fact, the main application of

the rule of consequence in under-approximation logics is to drop disjuncts whenever the formulae involved becomes unmanageably large. However, we already know that, for instance, $P''' \implies P' \implies P$, and similarly the assertions at every point of the Turn-U derivation are smaller than the corresponding assertions in the original IL derivation. Therefore, since we already completed the original IL derivation, any formula appearing in it is manageable, so the ones in this Turn-U derivation must be, too. Then, since this Turn-U proof can be carried out without rule $\langle$consIL$]$, the resulting triple is valid in both IL and SIL and can thus be used to continue the U-Turn derivation, proving a triple that is both IL and SIL in the end.

Note that the algorithm UTurn never applies $[$consSIL$\rangle$. Therefore, thanks to Theorem 4.7, it always finds triples that are valid both in SIL and IL. The same holds for the algorithm TurnU.

This result opens up a direct comparison with UNTer: since both proof systems find triples that are valid both in IL and SIL, why should we prefer one or the other? There are several points that distinguish the two. On the one hand, UNTer is able to reason about (non)termination of programs, something that U-Turn cannot do. On the other hand, we think the new way of combining triples exemplified by U-Turn is interesting in itself.

On a more technical level, most UNTer rules are inspired by IL in such a way to be applicable to a generic pre (except the rule Assume, only applicable to "specific" pre and post, cf. Section 2.6). This means that they are not immediately applicable (algorithmically, for backward analysis) to generic post-conditions, unlike U-Turn rules. As a downside, UNTer works in a single-pass algorithm, while U-Turn requires first a forward, IL-based step followed by a backward SIL-based step, leading to a two-passes algorithm.

When considering completeness of the two approaches there is no clear winner either. On the one hand, UNTer has no completeness result for triples that are valid in both IL and SIL at the same time. On the other hand, U-Turn is incomplete as well, in the traditional sense of the word (that is, every valid judgment is provable). However, nailing down the right notion of completeness for U-Turn is not a straightforward task. The above notion of completeness is very strong: namely, given any derivation d for an IL triple $\vdash [P]\ \mathsf{r}\ [Q]$ and any valid SIL triple $\vDash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$ such that $P' \implies P$ and $Q' \implies Q$, the resulting U-Turn judgment $\dfrac{d}{\vdash [P]\ \mathsf{r}\ [Q]} \vdash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$ is provable. A different but still relevant notion would be to require, for any provable IL triple $\vdash [P]\ \mathsf{r}\ [Q]$ and valid SIL triple $\vDash \langle P' \rangle\ \mathsf{r}\ \langle Q' \rangle$, the *existence* of a proof tree d for $\vdash [P]\ \mathsf{r}\ [Q]$ making the above U-Turn judgment provable. Currently, we are still unsure whether this is the case. We leave this question as future work.

5 Conclusions

Postconditions of triples in Incorrectness Logic—a forward under-approximation analysis—only expose errors that are reachable from the preconditions, but not all initial states described by the pre necessarily lead to errors. Conversely, the preconditions of triples in Sufficient Incorrectness Logic—a backward under-approximation analysis—include only initial states that can cause some of the errors in the postcondition, but not necessarily all of them. When the same triple is valid in both logics we have the best of both worlds: we are guaranteed that any initial state in the pre is the source of some error in the post and that all errors in the post are reachable from states in the pre. This form of summaries provides a highly valuable feedback for developers, because there are no false alarms and the sources of errors can facilitate testing and debugging activities.

In this paper we have explored the combination of forward and backward under-approximation approaches, to improve the precision of the analysis and to be able to match reachable errors with their sources. We are not the first ones to consider such a combined analysis: for instance, UNTer

logic already considered under-approximation triples that are valid in both directions, although, in that case, the emphasis is on exploiting backward under-approximation triples for non-termination analysis [Raad et al. 2024a]. We advance the state-of-the-art on this combination in two ways. First, the axioms for atomic commands in UNTer are handcrafted starting from the availability of a tool for forward analysis, namely PulseX. Therefore they required human ingenuity and are difficult to extend with new primitives. Moreover, they cannot be applied to either generic preconditions or postconditions. In this respect, we have provided a methodology to derive axiom schemes that are sound and complete by construction, thus solving all the above issues. Second, we propose a new way to combine IL and SIL analyses: rather than trying to derive directly a triple valid in both logics, we suggest a clever composition of the two techniques, where after deriving a valid triple in one logic we refine the proof to get a triple valid in both logics in an automatic way. Following these ideas, we defined a novel proof system, called U-Turn, whose judgments present a novel shape to compose derivations in different logics. Our main results show that U-Turn is sound, that it is able to derive triples valid in both SIL and IL under suitable assumption, and show how this inference can be automated. Interestingly, U-Turn can be used to refine a preliminary IL analysis using SIL or vice versa. Moreover, we have shown that whenever some additional form of approximation is necessary in one direction, e.g., to improve the performance of the analysis by dropping further disjuncts, the two ways of invoking U-Turn can mutually cooperate.

Future Works. There are many interesting directions we plan to explore further.

First, the shape of U-Turn judgments opens the possibility to combine different proof systems by reusing a derivation in one logic to drive the inference in the other. It would be interesting to see how far this principle can be extended to mix over- and under-approximations. To this aim, we need to further investigate which kinds of completeness are best suited to characterize the resulting proof systems, both from a purely technical perspective and with respect to applications.

Second, the use of under-approximation approaches to incorrectness reasoning has already been paired with abstract interpretation techniques for correctness analysis, and we would like to extend U-Turn in this respect. In Bruni et al. [2021] the authors introduce Local Completeness Logic to combine the derivation of IL triples $[P]\ \mathsf{r}\ [Q]$ with over-approximation in an abstract domain A in order to guarantee that the same under-approximation can be used for both correctness and incorrectness reasoning. We are confident that the local completeness technique can be smoothly extended to backward under-approximation and possibly integrated with U-Turn so that at least one source of errors will be exposed by the analysis whenever some error is possible.

Third, we plan to integrate our proof system in tools such as Pulse-X or Pulse$^\infty$. We envision two possibilities. The most natural one would be to instrument the forward analysis of incorrectness to record the minimal information from which the entire IL derivation d can be reconstructed and then execute UTurn a posteriori, starting from the whole postcondition Q proved by d. The other would be to directly interleave UTurn application during forward analysis. Whenever the IL postcondition computed so far by the analysis must be strengthened to match the precondition required by the next code fragment, the backward propagation using UTurn would guarantee that any derived IL triple will be also a SIL triple. Notably, backward propagation can also be run in parallel with the continuation of the IL analysis.

Fourth, the growing interest around hyperlogics for studying relational and hyperproperties [Benton 2004; Clarkson and Schneider 2008; Cousot and Wang 2025; Dardinier and Müller 2024; Sousa and Dillig 2016] can provide some challenging analysis scenarios, where U-Turn approach can play a fundamental role in tackling the complexity of the state-space by means of under-approximation.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback that helped us to improve the presentation and clarify many aspects of our contribution.

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 Pisa PRA_2022_99 *Formal methods for the healthcare domain based on spatial information (FM4HD)*. Azalea Raad is supported by a UKRI fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1, and by VeTSS.

References

Flavio Ascari, Roberto Bruni, Roberta Gori, and Francesco Logozzo. 2025a. Revealing Sources of (Memory) Errors via Backward Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 127 (4 2025), 28 pages. doi:10.1145/3720486

Flavio Ascari, Roberto Bruni, Roberta Gori, and Azalea Raad. 2025b. U-Turn: Enhancing Incorrectness Analysis by Reversing Direction (extended version). arXiv:2510.09292 [cs.LO] doi:10.48550/ARXIV.2510.09292

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. doi: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. doi:10.1145/3470569

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. doi:10.1145/964001.964003

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. doi:10.1145/781131.781153

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. 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, 70–87. doi:10.1007/978-3-642-18275-4_7

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. doi:10.1109/LICS52264.2021.9470608

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* 70, 2 (2023), 15:1–15:45. doi:10.1145/3582267

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 51–65. doi:10.1109/CSF.2008.7

Patrick Cousot and Jeffery Wang. 2025. Calculational Design of Hyperlogics by Abstract Interpretation. *Proc. ACM Program. Lang.* 9, POPL (2025), 446–478. doi:10.1145/3704852

Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1485–1509. doi:10.1145/3656437

Edsger W. Dijkstra. 1975. Guarded commands, non-determinacy and a calculus for the derivation of programs. In *Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975*, Martin L. Shooman and Raymond T. Yeh (Eds.). ACM, 2. doi:10.1145/800027.808417

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

Patrice Godefroid. 2005. The Soundness of Bugs is What Matters (Position Statement). https://patricegodefroid.github.io/public_psfiles/bugs2005.pdf

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

C. A. R. Hoare. 1978. Some Properties of Predicate Transformers. *J. ACM* 25, 3 (1978), 461–480. doi: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. doi:10.1145/3527325

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

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. doi:10.1007/978-3-030-88701-8_20

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. doi:10.1145/3211968

Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. doi: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. doi:10.1007/3-540-44802-0_1

David J. Pym, Peter W. O'Hearn, and Hongseok Yang. 2004. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.* 315, 1 (2004), 257–305. doi:10.1016/J.TCS.2003.11.020

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. doi:10.1007/978-3-030-53291-8_14

Azalea Raad, Julien Vanegue, and Peter W. O'Hearn. 2024a. Non-termination Proving at Scale. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 246–274. doi:10.1145/3689720

Azalea Raad, Julien Vanegue, and Peter W. O'Hearn. 2024b. Non-termination Proving at Scale - Extended Version. https://www.soundandcomplete.org/papers/OOPSLA2024/Unter/

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. doi:10.1109/LICS.2002.1029817

H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. doi:10.1090/S0002-9947-1953-0053041-6

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. doi:10.1145/3188720

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 57–69. doi:10.1145/2908080.2908092

G. Winskel. 1993. *The Formal Semantics of Programming Languages: an Introduction*. MIT press.

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. doi:10.1145/3586045

Received 2025-07-10; accepted 2025-11-06