On Extending Incorrectness Logic with Backwards Reasoning

FREEK VERBEEK, Open Universiteit of the Netherlands, Netherlands and Virginia Tech, USA
MD SYADUS SEFAT, Virginia Tech, USA
ZHOULAI FU, State University of New York, Republic of Korea, Stony Brook University, USA, and Virginia Tech, USA
BINOY RAVINDRAN, Virginia Tech, USA

This paper studies an extension of O'Hearn's incorrectness logic (IL) that allows backwards reasoning. IL in its current form does not generically permit backwards reasoning. We show that this can be mitigated by extending IL with underspecification. The resulting logic combines underspecification (the result, or postcondition, only needs to formulate constraints over relevant variables) with underapproximation (it allows to focus on fewer than all the paths). We prove soundness of the proof system, as well as completeness for a defined subset of presumptions. We discuss proof strategies that allow one to derive a presumption from a given result. Notably, we show that the existing concept of loop summaries – closed-form symbolic representations that summarize the effects of executing an entire loop at once – is highly useful. The logic, the proof system and all theorems have been formalized in the Isabelle/HOL theorem prover.

CCS Concepts: • **Theory of computation → Logic and verification**; *Hoare logic*; Assertions; **Program verification**; *Pre- and post-conditions.*

Additional Key Words and Phrases: incorrectness logic, backwards reasoning, program logic

ACM Reference Format:
Freek Verbeek, Md Syadus Sefat, Zhoulai Fu, and Binoy Ravindran. 2025. On Extending Incorrectness Logic with Backwards Reasoning. *Proc. ACM Program. Lang.* 9, POPL, Article 14 (January 2025), 25 pages. https://doi.org/10.1145/3704850

1 Introduction

O'Hearn's Incorrectness Logic (IL) is a logic intended for reasoning over the presence of bugs in programs [27]. Bug-finding tools often are lacking the formal fundament that many verification tools do tend to have. If a bug-finding tool is based on an *underapproximative* program logic such as IL, that brings the advantages of having no false positives: reported bugs are actually reachable.

IL concerns triples of the form $\langle P\rangle\ C\ \langle Q\rangle$ where P models a *presumption* and Q models a *result* (akin to respectively Hoare logic's pre- and postconditions). It asserts that *all result-states are reachable from some presumption-state.* This is formally defined as: $Q \subseteq \text{post}[C](P)$. In words, all Q-states are in the post-set of the presumption. It is therefore dual to standard Hoare Logic, which formulates triples as $\text{post}[C](P) \subseteq Q$.

The key difference between IL and Hoare Logic is that in IL, the result Q underapproximates the reachable states, whereas in Hoare Logic, the postcondition overapproximates them. Consider, e.g., the program $y \leftarrow x + 1$. The triple $[\![x = 9]\!]\ y \leftarrow x + 1\ [\![y = 10]\!]$ is a valid triple when

Authors' Contact Information: Freek Verbeek, Open Universiteit of the Netherlands, Heerlen, Netherlands and Virginia Tech, Blacksburg, USA, freek@vt.edu; Md Syadus Sefat, Virginia Tech, Blacksburg, USA, sefat@vt.edu; Zhoulai Fu, State University of New York, Incheon, Republic of Korea and Stony Brook University, New York, USA and Virginia Tech, Blacksburg, USA, zhoulaifu@sunykorea.kr; Binoy Ravindran, Virginia Tech, Blacksburg, USA, binoy@vt.edu.

ACM 2475-1421/2025/1-ART14
https://doi.org/10.1145/3704850

interpreted as a Hoare triple. It is, however, not a valid IL triple. The reason is that the result should be an underapproximation of the set of states reachable from the presumption. The result contains, e.g, the unreachable state where $y = 10 \wedge x = 0$, and thus does not underapproximate. By using underapproximative results, false positives are prevented [27]. In the above example, the overapproximative result $y = 10$, if used in subsequent analysis, may lead us to falsely conclude that an error is reachable from a state where $x = 0$.

De Vries and Koutavas show that IL (in their paper called *Reverse Hoare Logic*) allows natural specification of non-deterministic programs when needing to universally quantify over results [34]. Consider the program $y \leftarrow \mathsf{rand}()$, and the specification that all results where $y < 10$ should be reachable. The triple $[\![\mathsf{True}]\!]\ y \leftarrow \mathsf{rand}()\ [\![y < 10]\!]$ does not hold when interpreted as a Hoare triple. It is an IL triple: all result states are reachable from some state satisfying the presumption. Note that if a result state is reachable from some initial state s, this does not imply that all states reachable from s satisfy the result. Underapproximative results thus allow elegant formulation of specifications of non-deterministic programs.

The name Incorrectness Logic, thus, does not cover all its uses. When IL is used for reasoning over incorrectness, the result models a bug and IL aims at proving the presence of bugs without false positives. However, phrased more generically IL is useful for *proving that a program can at least produce all desired results*. This can be seen as incorrectness reasoning (the program can at least produce the bug), but also as correctness or reachability.

With these use cases in mind, it makes sense to combine IL with backwards reasoning, i.e., automatically generating a presumption from a desired result. This would enable one to formulate desired results (e.g., assertions or the negations thereof), and *prove* that these results are triggerable from some initial state satisfying the generated presumption. However, IL in its current form cannot generically be used for backwards reasoning. The difficulty of combining IL with backwards reasoning can be observed by looking at the assignment rule. The standard rule for HL (doing substitution) is unsound for IL. For IL, the assignment rule is as follows:

$$\frac{}{\langle P \rangle\ y \leftarrow e\ \langle \exists y' \cdot P[y'/y] \wedge y = e[y'/y] \rangle}\ \texttt{Assign}$$

This rule allows one to derive a result from a presumption (i.e., forwards reasoning), but not the other way around.

The underlying issue is that IL does not allow *underspecification* of the result. Consider again the first example above: the program $y \leftarrow x + 1$ with a result $y = 10$. This result has no presumption, since not *all* result-states are reachable: a state with $y = 10$ and $x = 0$ satisfies the result but is unreachable. In the context of backwards reasoning, one should have formulated a result $y = 10 \wedge x = 9$, to obtain a presumption $x = 9$. However, in a context where the desired result is defined by a user, requiring full specification is burdensome and infeasible.

The contribution of this paper is an extension of IL with underspecification, called Underspecified Incorrectness Logic (UIL). UIL explicitly enables underspecification by allowing one to treat variables as *irrelevant* variables. In the example above, one could formulate that one is interested in a result $y = 10$. That will generate a presumption where $x = 9$ with variable x underspecified ("irrelevant") in the result, and variable y underspecified in the presumption. In words, all states where $y = 10$ where we do not care about x are reachable from some state where $x = 9$ where we do not care about y.

We provide a proof system for backwards reasoning over UIL that is sound, and complete for a defined subset of presumptions. Practically, a challenge is dealing with loops. We show that UIL enables the use of existing techniques for *loop summarization* [15, 18, 30, 32, 35, 36]: aiming to overapproximate the effects of a loop with a single set of symbolic assignments. Given a loop summary, loops can be taken care of programmatically, i.e., no manual instantiation of loop invariants is required. The more variables can be summarized, the more precise the generated presumption becomes.

Developing trustworthy, scalable and principled bug-finding and reachability tools is a major challenge in both academia and industry. Providing developers of such a tool with a larger toolbox (e.g., backwards reasoning) will aid in their development. One can imagine, e.g., achieving more scalability by combining forward and backwards symbolic execution [3]. This paper focuses not on the development of such tools, but on extending IL with backward reasoning, aiming to add a new component to the toolbox of principled reachability analysis.

In the landscape of program logics, UIL distinguishes itself by being the only logic that allows both underspecification and underapproximation of the result. We show that this facilitates backwards reasoning. Section 2.1 concretizes these claims and their arguments. Even though various program logics related to incorrectness have been studied recently [25, 37], the variation presented in this paper is – to the best of our knowledge – novel. In Section 6 we discuss the relation to the state-of-the-art in more detail.

2 Underspecified Incorrectness Logic

Underspecified Incorrectness Logic (UIL) revolves around triples of the form:

$$\langle P \perp\!\!\!\perp V \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle$$

Here, C denotes a program, P and Q denote state predicates, and V and V' denotes sets of variables. The intuition is that a Q-state is reachable by execution of C from some P-state, *as long as variables from V (V') can be ignored in the presumption (result).* We call them *irrelevant* variables. This is very close to the intuition behind an incorrectness triple, which states that any Q-state is reachable from some P-state.

We use μ to denote a mapping from variables to values. Notation $s[V := \mu]$ is used to modify state s for all variables in V with the value from mapping μ. A transition from state s to state s' after execution of program C is denoted with $s \xrightarrow{C} s'$. Finally, we use $\overline{P}$ to denote the set of states satisfying predicate P. We redefine the post-set of a set of states to incorporate underspecification:

$$\mathrm{post}[C](P, V, V') \equiv \{s' \mid \exists s \in \overline{P} \cdot \forall \mu \cdot \exists \mu' \cdot s[V := \mu] \xrightarrow{C} s'[V' := \mu']\}$$

In words, when underspecifying variables V (V') in the presumption (result), the set of post-states reachable from a P-state is defined by considering transitions from *any* P-state where variables in V can be *any* value. If values for variables in V' can be selected so that state s' is produced, it is considered a post-state. Note that if both sets of variables V and V' are empty, the definition reverts back to the traditional definition of a post-set.

Example 2.1. Consider the program C as $y \leftarrow x + 4 \,\%\, 7$ (here we use $\leftarrow$ for assignment). Let $V = \varnothing$ and $V' = \{x\}$, that is, we treat variable x as irrelevant in the result. The set of post-states of presumption $x = 0$ is the set $\{s \mid s.y = 4\}$ (here $s.y$ denotes the value of variable y in state s). In contrast, if we take $V' = \varnothing$, the normal post-set of the presumption is the set $\{s \mid s.y = 4 \land s.x = 0\}$.

Definition 2.2. A *UIL*-triple is defined as:

$$\langle P \perp\!\!\!\perp V \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle \equiv \overline{Q} \subseteq \mathrm{post}[C](P, V, V')$$

The triples are under-approximative in the same fashion as incorrectness triples are [27]: the set of Q-states is a *subset* of the set of states reached from a P-state. The definition of a UIL triple formulates that for any Q-state s', there must be some P-state s that leads to s'. That transition is independent of the values of variables in V in state s, and may produce different values for variables in V' in state s'. In other words, V-variables in the presumption and V'-variables in the result are ignored. If both V and V' are empty, the definition reverts back to regular incorrectness logic.

2.1 Usefulness of UIL

To discuss usefulness of UIL, we contrast it with IL, Hoare Logic (HL) and Sufficient Incorrectness Logic (SIL, [2]). In this context, one would consider Hoare Triples of the form $\{\neg P\}\ C\ \{\neg Q\}$, which we will call Negated Hoare Logic (NHL). SIL uses triples defined as $P \subseteq \mathsf{pre}[C](Q)$, i.e., all P-states lead to some Q-state. For deterministic and terminating programs, this is equivalent to standard Hoare Logic.

We will make the following claims over UIL:

(1) In contrast to IL, UIL enables *backwards reasoning*. It thus enables a use-case where a desired result is provided by a user, and from that result a presumption can be generated.
(2) In contrast to IL, UIL allows *underspecification of a result*. This is important in the context of backwards reasoning over reachability of a result, since the result will typically not specify constraints over the entire state.
(3) In contrast to both NHL and SIL, but in accordance to IL, UIL allows *underapproximation of results*. In the context of reasoning over reachability, underapproximation is desired as the purpose only is finding *some* path that triggers the result.
(4) In contrast to SIL, but in accordance to IL, UIL aims at *proving all results are reachable*.

Claim 1. The first claim is argued by the fact that for IL, backwards reasoning is impossible, whereas for UIL we will provide a proof system for backwards reasoning in Section 2.2. For IL, there is no backwards rule for dealing with assignment. The standard rule for HL is unsound [27]. O'Hearn states that (Fact 9 in [27]) for IL: "valid presumptions need not exist: given a relation C and result Q, there need not exist any P such that $\langle P\rangle\ C\ \langle Q\rangle$". UIL aims to address that issue.

Table 1. Examples of program triples. We consider a state with two variables x and y storing natural numbers.

			UIL	IL	NHL	SIL
$[\![(x \% 7) = 0]\!]$	$y \leftarrow x + 4 \% 7$	$[\![y = 4]\!]$	✓		✓	✓
$[\![y = 0]\!]$	$y \leftarrow y + 4 \% 7$	$[\![y = 4]\!]$	✓	✓		✓
$[\![x = 0]\!]$	$y \leftarrow x + 4 \% 7$	$[\![y = 4]\!]$	✓			✓
$[\![x = 0]\!]$	$y \leftarrow x + 4 \% 7$	$[\![y \geq 4]\!]$				✓
$[\![(x \% 7) = 0]\!]$	$y \leftarrow x + 4 \% 7$	$[\![y = 4 \wedge (x \% 7) = 0]\!]$	✓	✓	✓	✓

Claim 2. Consider the first example in Table 1. In the example, we are interested in a result where $y = 4$, and x and y are natural numbers. In the context of IL, there is no presumption P, in other words, there is no IL triple for this result. Reason is that not *all* states where $y = 4$ are reachable. For IL, the result may not be underspecified. In this example, the result must also specify $(x \% 7) = 0$ for an IL triple to hold. UIL allows to establish this triple, by treating x as an irrelevant variable in the result.

Claim 3. Consider the second example of Table 1. IL differs with NHL whenever there are multiple ways to achieve the same result. For IL and UIL, it suffices that the presumption models *some* way to achieve the result ($y = 0$ in the example). NHL requires the presumption to model *all* ways to achieve the result. This is a fundamental characteristic of IL that UIL inherits. It can be seen from the inference rules `Pre-Weaken` and `Post-Strengthen` in Figure 1. What UIL inherits from IL is *postcondition strengthening*. This, in words of O'Hearn, "allows to focus on fewer than all the paths, a feature which is a hallmark of under-approximation" and "can be used in order to help a reasoning tool scale" [27]. In other words, IL (and thus UIL) allows postcondition strengthening, whereas NHL and SIL do not.

The third example combines the above claims into an example where UIL enables underspecification (variable x is not specified in the result) and underapproximation (there are more reachable states where $y = 4$ than provided by presumption $x = 0$). IL is *stronger* than UIL without irrelevant variables in the presumption, i.e.:

$$\langle P \rangle\ C\ \langle Q \rangle \implies \langle P \perp\!\!\!\perp \varnothing \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle$$

Claim 4. Finally, the purpose of IL (and thus UIL) is to prove all results are reachable. Consider the fourth row in Table 1. Not *all* result states are reachable from the presumption, which is why both IL and UIL disallow it. SIL shows that *some* result state (namely, where $y = 4$) is *always* reached from the initial state where $x = 0$.

Loop Unrolling. A hallmark of IL is *loop unrolling* [27]. This allows proving triples over a loop, without needing invariants. UIL inherits loop unrolling from IL, i.e., the exact same form of reasoning is possible for UIL. The loop unrolling rule for UIL is as follows:

$$\frac{\forall i \leq n \cdot \langle P \perp\!\!\!\perp V \rangle\ C^i\ \langle Q_i \perp\!\!\!\perp V' \rangle}{\langle P \perp\!\!\!\perp V \rangle\ \texttt{While True Do}\ C\ \langle \bigvee_{i \leq n} Q_i \perp\!\!\!\perp V' \rangle}\ \texttt{LoopUnrolling}$$

Here C^i denotes i repetitions of program C. The rule generalizes to loops with loop conditions other than True.

Loop unrolling states that one can prove a triple over a loop by proving the triple over a bounded number of iterations. This is sound, since we only need to prove the result is reachable through some path, i.e., there is no need to reason over all possible behaviors of the loop. Thus, there is no need for invariants. Consider program C as $\texttt{While}\ x < 10\ \texttt{Do}\ x \longleftarrow x + 1$ and a result Q as $x = 10$. We can derive a triple for this result simply by unrolling the loop a certain number of iterations. As such, we can derive $\langle x = 5 \rangle\ C\ \langle Q \rangle$ by unrolling five iterations, but also $\langle Q \rangle\ C\ \langle Q \rangle$ by simply unrolling zero iterations.

Combining UIL with test-case generation. A UIL triple $\langle P \perp\!\!\!\perp V \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle$ shows that a bug characterized by Q is reachable. It shows that in order to replay the bug, a state can be initialized where variables not in V must have some value according to presumption P. Variables from V need not be initialized with specific values. Deriving a UIL triple thus becomes valuable when presumption P is *strong* and set V is *large*. A triple with the weakest presumption and smallest set of irrelevant variables $\langle \text{True} \perp\!\!\!\perp \varnothing \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle$ shows that the result is reachable, but does not provide any information on how to obtain that result. In contrast, a triple $\langle x = 0 \perp\!\!\!\perp U - x \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle$ – where U is the universe set of all variables – provides a lot of information at it shows that all result states are reachable from a state where $x = 0$ and that no other variables influence that reachability.

Similarly, a UIL triple becomes more valuable if the result is *weak* and set V' is *small.* A weaker result means that more states have been proven reachable. In contrast, the result False vacuously holds. If set V' is the universe set U and the result is not False, then triple $\langle P \perp\!\!\!\perp V \rangle\ C\ \langle Q \perp\!\!\!\perp U \rangle$ only shows that P is not False (assuming program C terminates). That triple basically only formulates "there exists some P-state that leads to some state". In contrast, a triple $\langle P \perp\!\!\!\perp V \rangle\ C\ \langle x \geq 0 \perp\!\!\!\perp \{\} \rangle$ shows that all states where $x \geq 0$ are reachable from the presumption.

Since result Q is provided by a user, we thus argue UIL is useful in the context of test-case generation if it provides strong presumptions that underspecify many variables.

$$\frac{\mathbf{g} = \overleftarrow{f} \text{ on } \mathsf{VS}(y, Q) \qquad x \neq y \implies x \in V \qquad Q \implies y \in \text{image(f)}}{\langle Q[f(x)/y] \land x \in \text{image}(\mathbf{g}) \perp\!\!\!\perp V + y - x \rangle\ y \ \leftarrow\ f(x)\ \langle Q \perp\!\!\!\perp V \rangle}\ \texttt{Assign}$$

$$\frac{Q \implies y = f(x) \qquad x \notin V}{\langle Q[f(x)/y] \perp\!\!\!\perp V + y - x \rangle\ y \ \leftarrow\ f(x)\ \langle Q \perp\!\!\!\perp V \rangle}\ \texttt{Assign\_Relevant\_Src}$$

$$\frac{y \in V}{\langle Q \perp\!\!\!\perp V \rangle\ y \ \leftarrow\ f(x)\ \langle Q \perp\!\!\!\perp V \rangle}\ \texttt{Assign\_Irrelevant\_Dst}$$

$$\frac{}{\langle \exists v \cdot Q[v/y] \perp\!\!\!\perp V + y \rangle\ \texttt{destroy(y)}\ \langle Q \perp\!\!\!\perp V \rangle}\ \texttt{Destroy}$$

$$\frac{\langle P \perp\!\!\!\perp V \rangle\ C_0\ \langle Q \perp\!\!\!\perp V' \rangle \qquad \langle Q \perp\!\!\!\perp V' \rangle\ C_1\ \langle R \perp\!\!\!\perp V'' \rangle}{\langle P \perp\!\!\!\perp V \rangle\ C_0; C_1\ \langle R \perp\!\!\!\perp V'' \rangle}\ \texttt{Seq}$$

$$\frac{\langle P_0 \perp\!\!\!\perp V_0 \rangle\ C_0\ \langle Q \land b \perp\!\!\!\perp V' - b \rangle \qquad \langle P_1 \perp\!\!\!\perp V_1 \rangle\ C_1\ \langle Q \land \neg b \perp\!\!\!\perp V' - b \rangle}{\langle P_0 \lor P_1 \perp\!\!\!\perp V_0 \cap V_1 \rangle\ \texttt{If}\ b\ \texttt{Then}\ C_0\ \texttt{Else}\ C_1\ \langle Q \perp\!\!\!\perp V' \rangle}\ \texttt{ITE}$$

$$\frac{\langle P_0 \perp\!\!\!\perp V_0 \rangle\ C_0\ \langle Q \land b \perp\!\!\!\perp V' - b \rangle \qquad b \in V'}{\langle P_0 \perp\!\!\!\perp V_0 \rangle\ \texttt{If}\ b\ \texttt{Then}\ C_0\ \texttt{Else}\ C_1\ \langle Q \perp\!\!\!\perp V' \rangle}\ \texttt{ITE\_Irrelevant\_True}$$

$$\frac{\langle \mathbf{P}(n') \land b \perp\!\!\!\perp V \rangle\ C\ \langle \mathbf{P}(n' + 1) \land n' < \mathbf{n} \perp\!\!\!\perp V \rangle \qquad Q \implies \mathbf{P}(\mathbf{n}) \land \neg b}{\langle \mathbf{P}(0) \perp\!\!\!\perp V \rangle\ \texttt{While}\ b\ \texttt{Do}\ C\ \langle Q \perp\!\!\!\perp V \rangle}\ \texttt{While}$$

$$\frac{\langle \mathbf{P}_0 \perp\!\!\!\perp \mathbf{V}_0 \rangle\ C\ \langle \mathbf{Q}_0 \perp\!\!\!\perp \mathbf{V'}_0 \rangle \qquad \overline{\mathbf{P}_0} \subseteq \overline{P} \qquad \overline{Q} \subseteq \overline{\mathbf{Q}_0} \qquad V \subseteq \mathbf{V}_0 \qquad \mathbf{V'}_0 \subseteq V'}{\langle P \perp\!\!\!\perp V \rangle\ C\ \langle Q \perp\!\!\!\perp V' \rangle}\ \begin{array}{l}\texttt{Pre-Weaken}\\ \texttt{Post-Str.}\end{array}$$

$$\frac{\langle P_0 \perp\!\!\!\perp V_0 \rangle\ C\ \langle Q_0 \perp\!\!\!\perp V' \rangle \qquad \langle P_1 \perp\!\!\!\perp V_1 \rangle\ C\ \langle Q_1 \perp\!\!\!\perp V' \rangle}{\langle P_0 \lor P_1 \perp\!\!\!\perp V_0 \cap V_1 \rangle\ C\ \langle Q_0 \lor Q_1 \perp\!\!\!\perp V' \rangle}\ \texttt{SplitResult}$$

Fig. 1. Proof system for UIL suitable for backward reasoning. The bold parts need to be instantiated with values when applying the rules in backwards fashion. Wrt. notation: we use $V + y$ to insert element y in set V (and similar for removal with $-$.)

2.2 Backwards Reasoning over UIL

As programs, we consider a simple `While` language [19], with non-determinism. We consider if-statements with singular Boolean variables b as branching conditions, that are not modified by execution of the if-statement itself. This allows us to elegantly formulate a rule that selects a path in backwards fashion. Note that this is not a restriction: any if-statement `If` e `Then` C_0 `Else` C_1 (for some expression e) can be rewritten to $b \ \leftarrow\ e;$ `If` b `Then` C_0 `Else` C_1 for some fresh variable b. To introduce non-termination, we introduce the `perpetuate` construct. To introduce non-determinism, we add the `destroy(y)` construct, that writes some unknown non-deterministic value to variable y. Semantics are given by a standard transition relation over states $\xrightarrow{C}$. For the `destroy(y)` construct, for any value v we have $s \xrightarrow{\texttt{destroy(y)}} s[y := v]$. There is no s' such that $s \xrightarrow{\texttt{perpetuate}} s'$.

As predicates, we consider predicates that may include a $\top$ symbol (an unknown value). A predicate is interpreted as Kleene's strong three-valued logic [21]. Notably, both $\top \implies \top$ and

$\top = \top$ evaluate to $\top$. We consider a state to be satisfying predicate P if and only if predicate P *possibly* holds, i.e.:

$$s \in \overline{P} \iff P \text{ can evaluate to true in state } s$$

For example, $\overline{\top}$ is the universe set of all states.

Figure 1 provides a proof system for UIL for backwards reasoning. We explain the rules below. Note there is no rule for the `perpetuate` construct, as any result after that is unreachable.

Assignment. We present an assignment rule for assigning $f(x)$ to variable y. Note that x can be a tuple, in case of functions with multiple arguments (examples follow in Section 2.3). Function f can be the identity function in case of simple data movement.

The rule states that a presumption $Q[f(x)/y]$ can be derived from a result Q, which is akin to the standard assignment rule for Hoare logic and for backwards underapproximative reasoning as defined in [25]. However, it requires the assumption that for all Q-states variable y is within the image of function f. Consider again the first example of Table 1. If result Q would permit states where $y \geq 7$, i.e., outside of the image of f, then it is not reachable. When x is used as input to the assignment, it must be an irrelevant variable in the result.

Moreover, function f must have a *right inverse*. Function $\mathbf{g}$ of type $Y \mapsto X$ is a right inverse of function f of type $X \mapsto Y$ on set Y', notation $\mathbf{g} = \overleftarrow{f}$ on Y', if and only if, $f(\mathbf{g}(y)) = y$ for all $y \in Y'$. In words, function f can "undo" function $\mathbf{g}$ for all y in Y'. Assuming the axiom of choice, every function f has a right inverse on the image of f.

We loosen the requirement on $\mathbf{g}$ by requiring it to be a right inverse only for relevant values. Let the *value set* of variable y in predicate Q, notation $\mathsf{VS}(y, Q)$ be the set of all values that variable y can have in Q:

$$\mathsf{VS}(y, Q) \equiv \{v \mid \exists s \in \overline{Q} \cdot s.y = v\}$$

We require $\mathbf{g}$ to be right inverse only on the value set of variable y in predicate Q.

Function $\mathbf{g}$ can be used to "zoom in" on specific values when instantiated appropriately. The generated presumption allows us to restrict the values of input x to the image of g. This is essential for dealing with assignments *underapproximatively*. Section 2.3 provides further examples on how to instantiate $\mathbf{g}$.

Example 2.3. Let $f(x) = x + 4 \,\%\, 7$ and $\mathbf{g}(y) = y - 4 \,\%\, 7$. Since $f(\mathbf{g}(y)) = ((y - 4 \,\%\, 7) + 4 \,\%\, 7) = y$ for all $y < 7$ (i.e., for all y in the image of f), $\mathbf{g}$ is the right inverse of f. The image of $\mathbf{g}$ amounts to $x < 7$. Thus we can derive:

$$\frac{}{\langle (x + 4 \,\%\, 7) = 4 \land x < 7 \perp\!\!\!\perp \{y\}\rangle\ y \ \leftarrow\ x + 4 \,\%\, 7\ \langle y = 4 \perp\!\!\!\perp \{x\}\rangle}\ \texttt{Assign}$$

The presumption simplifies to $x = 0$ (the third example of Table 1). The result underspecifies variable x. The presumption states that the value of y is irrelevant for reachability of the result. Note that we could have also instantiated $\mathbf{g}$ with $\mathbf{g}(y) = y - 4$. However, that would not have allowed us to "zoom in" on value $x = 0$ as the image of that function is the universe set.

In cases where input x is considered relevant, proving that the result is necessarily reachable can only be done for a specific set of results. Rule `Assign_Relevant_Src` shows that backwards traversal is then possible, if result Q is strong enough to imply reachability. Essentially, in this case one does not have the benefit of underspecification, and therefore Q must be sufficiently specified. Rule `Assign_Irrelevant_Dst` shows that as long as variable y is irrelevant in the result, we can simply ignore it.

Sequence. The sequence rule is straightforward. It shows that intermediate irrelevant variables are gathered.

Example 2.4. We can derive the following:

$$\dfrac{\langle x > 3 \perp\!\!\!\perp V+z+y-x\rangle\ y \leftarrow x+3\ \langle y > 6 \perp\!\!\!\perp V+z-y\rangle \qquad \langle y > 6 \perp\!\!\!\perp V+z-y\rangle\ z \leftarrow y+4\ \langle z > 10 \perp\!\!\!\perp V\rangle}{\langle x > 3 \perp\!\!\!\perp V+z+y-x\rangle\ y \leftarrow x+3;\ z \leftarrow y+4\ \langle z > 10 \perp\!\!\!\perp V\rangle}\ \texttt{Seq}$$

In words, all states where $z > 10$ are reachable from a state where $x > 3$. Variables z and y are irrelevant variables in the presumption, whereas x is relevant.

If-then-else. The if-then-else rule shows that we can split a result into two paths. Only variables that are irrelevant for both paths can be considered to be irrelevant for the if-then-else statement.

Note that if the result specifies the path of interest (by implying a value for branching condition b), then one does not need to explore the other path. Since for the other path the result will be logically equivalent to False, one can derive as presumption for that path True with as irrelevant set V the universe set.

Example 2.5. We can derive the following for any V not containing b:

$$\dfrac{\langle b \perp\!\!\!\perp V+y\rangle\ y \leftarrow 0\ \langle y=0 \wedge b \perp\!\!\!\perp V\rangle}{\langle b \perp\!\!\!\perp V+y\rangle\ \texttt{If}\ b\ \texttt{Then}\ y \leftarrow 0\ \texttt{Else}\ \ldots\ \langle y=0 \wedge b \perp\!\!\!\perp V\rangle}\ \texttt{ITE}$$

In words, all states where $y = 0$ are reachable from a state where branching condition b holds. Variable y is an irrelevant variable in the presumption, whereas b is relevant. Note that this triple can be established without any knowledge over the other branch (the dots).

A similar reasoning is applied through rule `SplitResult`, but more generically. If result Q can be split into two disjuncts, then one can perform backwards traversal on both disjuncts. The presumptions can be combined through disjunction, and the sets of irrelevant variables are intersected. This rule is necessary for a proof of completeness (see Theorem 2.12).

However, it may also be the case that the result underspecifies the path of interest by having $b \in V'$. In that case one can choose during backwards exploration which path to take. Rule `ITE_Irrelevant_True` shows that one can explore one path (the rule for the `False` case is omitted but is similar).

While. Dealing with loops is the most involved. It requires instantiation of two constituents: a *parameterized* state predicate $\mathbf{P}$ that is backwards inductive, i.e., a state satisfying $\mathbf{P}(n'+1)$ should be reachable from a state satisfying $\mathbf{P}(n')$. (the term "backwards inductive" will be defined formally in Section 3.2). Moreover, it requires finding an *iteration count* $\mathbf{n}$ that models the number of iterations the loop is executed. The result Q should imply $\mathbf{P}(\mathbf{n})$, i.e., it should imply $\mathbf{n}$ loop iterations, and it should imply the negation of the branching condition. The latter is necessary: a while-loop with branching condition b can never result in a state where b holds. The rule states that if there *exists* an instantiation for $\mathbf{n}$ such that *for all* $n' < \mathbf{n}$ the assumption holds, the conclusion can be derived.

Consider the following program W: $\texttt{While}\ x < 16\ \texttt{Do}\ x \leftarrow x+2$. Consider any Q that implies the negation of the branching condition, e.g, $x = 100$. One can always instantiate $\mathbf{P}(n') = Q$ for all n' and instantiate $\mathbf{n} = 0$. That models that the loop is not executed at all. The assumption of the while-rule becomes vacuously true (note that the result of the triple in the assumption becomes false). This way, it is trivial to derive:

$$\dfrac{}{\langle x = 100 \perp\!\!\!\perp V\rangle\ W\ \langle x = 100 \perp\!\!\!\perp V\rangle}\ \texttt{While}$$

However, if one wants to show reachability of the result after a certain number of iterations, $\mathbf{P}$ must be instantiated more cleverly.

Example 2.6. Consider program W above. We consider a result Q where $x = 16$. Let $\mathbf{P}(n')$ return true iff $x = 4 + 2n'$ and let $\mathbf{n} = 6$. We have $Q \implies x = 4 + 2 \cdot 6 \wedge x \geq 16$. Therefore, we can derive for all $n' < 6$:

$$\frac{\langle x = 4 + 2n' \wedge x < 16 \perp\!\!\!\perp V\rangle\ x \leftarrow x+2\ \langle x = 4 + 2(n'+1) \wedge n' < 6 \perp\!\!\!\perp V\rangle}{\langle x = 4 \perp\!\!\!\perp V\rangle\ W\ \langle x = 16 \perp\!\!\!\perp V\rangle}\ \texttt{While}$$

In words, all states where $x = 16$ are reachable from a state where $x = 4$ after 6 loop iterations.

This rule again exposes the ability to focus on fewer than all paths. This time not by selecting a path through a branching condition, but by instantiating an iteration count. One can always choose 0 iterations. In the example above, we could also have instantiated $\mathbf{P}(n')$ with $x = 2n'$ and $\mathbf{n}$ with 8. That would lead to presumption $x = 0$. In Section 3.2, we will show how parameterized state predicates and iteration counts for loops can be found programmatically.

2.3 Examples of Applying Rule Assign

We first show an example that illustrates how the image of function $\mathbf{g}$ can be used to "zoom in" on specific parts of the domain, similar to Example 2.3.

Example 2.7. Consider a program $y \leftarrow \mathsf{sin}(x)$ with x and y real numbers. For any n, function $\mathbf{g}(y) = sin^{-1}(y) + n\pi$ is a right inversion on the image of sin. That function has as image $-\frac{1}{2}\pi + n\pi \leq y \leq \frac{1}{2}\pi + n\pi$. Consider as result $0 \leq y \leq 0.5$. We can derive, among others, the following presumptions:

$$\begin{array}{lll}
0 \leq x \leq \frac{1}{6}\pi & & \text{by taking } \mathbf{g}(y) = sin^{-1}(y) \\
\frac{5}{6}\pi \leq x \leq \pi & & \text{by taking } \mathbf{g}(y) = sin^{-1}(y) + \pi \\
\ldots & & \\
\quad \mathbf{m}\pi \leq x \leq (\mathbf{m} + \frac{1}{6})\pi & \text{and } \mathbf{m} \text{ is even} & \multirow{2}{*}{} \text{by taking } \mathbf{g}(y) = sin^{-1}(y) + \mathbf{m}\pi \\
\vee\ (\mathbf{m} - \frac{1}{6})\pi \leq x \leq \mathbf{m}\pi & \text{and } \mathbf{m} \text{ is odd} &
\end{array}$$

Each of these cases can be considered a path, i.e., one can choose one of these intermediate presumptions and continue from there to try and reach the entry point. Note that the final example delays picking a path by introducing a fresh variable $\mathbf{m}$ and by *not* instantiating it with a value. We expand on these points in Section 3.

Next, we show how the fact that $\mathbf{g}$ only needs to be a right inverse on the value set of Q can be leveraged. For function sin, we can prove:

$$\langle x = 0 \perp\!\!\!\perp \{y\}\rangle\ y \leftarrow \mathsf{sin(x)}\ \langle y = 0 \perp\!\!\!\perp \{x\}\rangle$$

simply by instantiating $\mathbf{g}$ such that $\mathbf{g}(y) = 0$, even though that function is not a right inverse of sin on its entire image. The value set of y in Q is $\{0\}$, which makes it easy to prove a presumption such as $x = 0$ or $x = \pi$ for result $y = 0$.

One might have the intuition that if function f destroys information (i.e., performs abstraction), the function cannot be reversed. However, the underapproximative nature of UIL only requires to find *some* values that lead to the result.

Example 2.8. Let x (y) be a real (natural) number, and consider the floor function that abstracts real numbers to naturals. It has no inverse, but we can prove:

$$\langle x \leq 10 \perp\!\!\!\perp \{y\}\rangle\ y \leftarrow \lfloor x \rfloor\ \langle y \leq 10 \perp\!\!\!\perp \{x\}\rangle$$

by using $\mathbf{g}(y) = \mathsf{min}(y, 10)$, which is a right inverse on the value set of y in the result. Note that the presumption is a simplification of $\lfloor x \rfloor \leq 10 \wedge x \in \mathsf{image}(\mathbf{g})$, and only the right hand side of this conjunction actually provides the underapproximative bound 10 of the presumption.

Finally, we show an example for an assignment with multiple inputs. Input x is considered a tuple, and thus function $\mathbf{g}$ must produce a tuple from a value y.

Example 2.9. Consider a situation where for the xor function we want to prove:

$$\langle x = w \perp\!\!\!\perp \{y\}\rangle\ y \ \leftarrow\ x \oplus w\ \langle y = 0 \perp\!\!\!\perp \{x, w\}\rangle$$

We instantiate $\mathbf{g}$ such that $\mathbf{g}(y) = (y, 1)$ and establish that $f(\mathbf{g}(y)) = y \oplus 1 = y$. This derives presumption $x \oplus w = 0 \land w = 1$, and with precondition weakening we can prove the example.

2.4 Soundness and Completeness

Theorem 2.10 (Soundness). *The proof system in Figure 1 is sound, i.e., any triple derived from these rules holds.*

$$\vdash \langle P \perp\!\!\!\perp V\rangle\ C\ \langle Q \perp\!\!\!\perp V'\rangle \implies \models \langle P \perp\!\!\!\perp V\rangle\ C\ \langle Q \perp\!\!\!\perp V'\rangle$$

Proof. The theorem has been formalized and proven in the Isabelle theorem prover in Higher-Order-Logic (HOL). We here discuss the proof for rule `Assign` as example.

We need to prove that for any state $s' \in \overline{Q}$, there exists some state $s \in \overline{P}$ such that for all μ there exists a μ' such that $s[V - y + x := \mu] \xrightarrow{y \leftarrow f(x)} s'[V := \mu']$. We take as value for x the value $\mathbf{g}(s'.y)$. We construct state s as follows:

$$s \overset{\text{def}}{=} s'[x := \mathbf{g}(s'.y)]$$

We prove that $s \in \overline{Q}[f(x)/y]$ (this requires the assumptions that $\mathbf{g}$ is the right inverse of f and that $s'.y$ is in the image of f). Moreover, we have that $s.x \in \text{image}(\mathbf{g})$. Thus we have $s \in \overline{P}$.

Let μ be any mapping. We construct μ' as follows:

$$\mu' \overset{\text{def}}{=} \mu \left[\begin{array}{l} x := \begin{cases} s'.y & \text{if } x = y \\ \mathbf{g}(s'.y) & \text{if otherwise} \end{cases} \\ y := s'.y \end{array} \right]$$

We prove that $s[V - y + x := \mu] \xrightarrow{y \leftarrow f(x)} s'[V := \mu']$. From this, the conclusion follows. □

Completeness concerns both the question 1.) whether for any result every valid presumption is derivable, and 2.) whether for any set of irrelevant result-variables the largest valid set of irrelevant presumption-variables can be generated.

We first consider 2.). The proof system is not complete in this regard. Consider the following triple:

$$\langle \text{True} \perp\!\!\!\perp \{t, w\}\rangle\ t \ \leftarrow\ w\,;w \ \leftarrow\ 0\,;y \ \leftarrow\ w\,;\ w \ \leftarrow\ t\ \langle \text{True} \perp\!\!\!\perp \{t, w\}\rangle$$

This triple holds, all result states are reachable from some presumption-state, and in the presumption variables t and w do not matter. However, the rule `Assign` from Figure 1 removes variable `w` from the set of irrelevant variables. Thus we can only derive:

$$\langle \text{True} \perp\!\!\!\perp \{t\}\rangle\ t \ \leftarrow\ w\,;w \ \leftarrow\ 0\,;y \ \leftarrow\ w\,;\ w \ \leftarrow\ t\ \langle \text{True} \perp\!\!\!\perp \{t, w\}\rangle$$

In other words, the proof system does not enable us to prove that variable w is irrelevant.

For this reason, completeness can only be proven relative to the set of irrelevant variables generated while deriving a presumption. Given the initial set V' of irrelevant variables in the result, that set can be computed by traversing program C backwards. For each assignment the written (read) variable is added (subtracted) to the set. For each if-then-else statement, the intersection is taken after traversing both branches. The set of irrelevant presumption-variables generated in this fashion – given program C and set of irrelevant result-variables V' – is denoted with $\text{irrelevant}(C, V')$.

With respect to 1.), rule `Assign` generates a presumption with a constraint over variable x only, and no constraints that combine the inputs to the assignment with other variables. Consider, e.g., the following valid triple:

$$\langle w + x = 10 \perp\!\!\!\perp \{y\}\rangle\ y \ \leftarrow\ x\ \langle y \leq 10 \perp\!\!\!\perp \{w, x\}\rangle$$

This triple is not derivable. The proof system only allows to derive a presumption $x \leq 10$.

Thus we must formulate that a presumption treats a (set of) variables as *detached* from other variables:

Definition 2.11. Variable x is *detached* in predicate P, notation $\mathsf{detached}(x, P)$, if and only if:

$$\forall s \in \overline{P}, v \in \mathsf{VS}(x, P) \cdot s[x := v] \in \overline{P}$$

In words, for any P-state s, the current value of x in s can be replaced with another value for x that satisfies P, and the result still satisfies P. For example, in the property $x < 10 \wedge y < z$, variable x is detached. In the above example, variable x is not detached in the presumption as its value influences the value of w. The definition can be formulated similarly when x is a tuple of multiple variables, for assignments with multiple inputs.

For presumptions where all input variables are detached, the proof system in Figure 1 is complete. The rules for dealing with loops, if-statements and sequence are complete. We have formalized this, by stating that *if* hypothetically there would be a rule `Assign` for assignments that introduces undetached presumptions, then the proof system is complete.

Theorem 2.12 (Completeness). *Assume that there exists some rule* `Assign_Undetached` *such that for any presumption* P_U *and result* Q*:*

$$\frac{\neg\mathsf{detached}(x, P_U) \qquad y \notin V' \qquad \models \langle P_U \perp\!\!\!\perp V\rangle\ y \ \leftarrow\ f(x)\ \langle Q \perp\!\!\!\perp V'\rangle}{\vdash \langle P_U \perp\!\!\!\perp V\rangle\ y \ \leftarrow\ f(x)\ \langle Q \perp\!\!\!\perp V'\rangle}\ \texttt{Assign\_Undetached}$$

then the proof system in Figure 1 together with rule `Assign_Undetached` *is complete for any presumption* P *and set of irrelevant variables* V *such that* $V \subseteq \mathsf{irrelevant}(C, V')$*:*

$$\models \langle P \perp\!\!\!\perp V\rangle\ C\ \langle Q \perp\!\!\!\perp V'\rangle \implies \vdash \langle P \perp\!\!\!\perp V\rangle\ C\ \langle Q \perp\!\!\!\perp V'\rangle$$

Proof. The theorem has been formalized and proven in the Isabelle/HOL theorem prover [5, 11, 26]. We here discuss the proof for rule `Assign` as example.

We must prove that assuming ($\mathcal{A}$) a triple with presumption P and result Q holds, then it can be derived. There are various cases possible. Consider the case where variable y is relevant in the result ($y \notin V$), variable x is irrelevant ($x \in V$), and variable x is detached in P. We first prove that assumption ($\mathcal{A}$) implies that for all values in the value set of y in Q, there exists some value in the value set of x in P that can be used to compute the y-value:

$$\forall v_y \in \mathsf{VS}(y, Q) \cdot \exists v_x \in \mathsf{VS}(x, P) \cdot f(v_x) = v_y$$

From this, it is possible to construct the right inversion $\mathbf{g}$ (this uses the axiom of choice). We define P_{strong} as follows:

$$P_{\mathsf{strong}} \overset{\text{def}}{=} Q[f(x)/y] \wedge x \in \mathsf{image}(\mathbf{g})$$

We then prove that $P_{\mathsf{strong}} \implies P$. What we actually prove is that each state in $\overline{P_{\mathsf{strong}}}$ implies the existence of a state s_0 in $\overline{P}$, such that states s and s_0 agree on all values but the value for x. Since that variable is detached in P, that implies that state s is in $\overline{P}$ as well.

Finally, we prove that Q necessarily implies that y is in the image of f, since otherwise result Q would model unreachable states which violates assumption ($\mathcal{A}$).

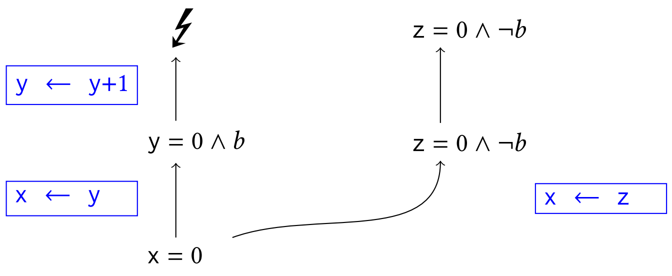

Fig. 2. Backwards exploration up to the entry point.

We can now show that the triple is derivable, by deriving it as follows:

$$\dfrac{\dfrac{\cancel{\mathbf{g} = \overleftarrow{f}\ \text{on}\ \mathsf{VS}(y,Q)} \qquad \cancel{x \neq y \implies x \in V} \qquad \cancel{Q \implies y \in \mathsf{image}(\mathsf{f})}}{\langle P_{\text{strong}} \perp\!\!\!\perp V + y - x\rangle\ y \leftarrow f(x)\ \langle Q \perp\!\!\!\perp V\rangle} \qquad \cancel{P_{\text{strong}} \implies P}}{\langle P \perp\!\!\!\perp V + y - x\rangle\ y \leftarrow f(x)\ \langle Q \perp\!\!\!\perp V\rangle}$$

This finishes the proof for the case where $y \notin V$, $x \in V$ and x is detached in P. The other cases, e.g., the case where x and y both are relevant, are proven using different derivations. □

3 Proof Strategies

Underapproximation allows to "pick a path", and that comes with the task to find the right path. It may be the case that one traverses a program backwards, but cannot complete the path all the way up to the entry point. In such cases, one must backtrack and pick a different path. Underapproximation, thus, causes a backwards traversable *search space*, where a path towards the entry point needs to be found. As soon as the entry point is hit, backwards exploration can stop and unexplored paths need not be visited. Note that generating a false presumption is success: that proves unsatisfiability of the result. However, it may be the case that no rules can be applied to some intermediate presumption.

Consider the program $y \leftarrow y+1$; `If` b `Then` $x \leftarrow y$ `Else` $x \leftarrow z$ and result $\mathsf{x} = 0$ with b an irrelevant variable (x, y and z are natural numbers). Both rules `ITE_Irrelevant_*` can be applied, but only one will lead to the entry point. See Figure 2: the intermediate presumption $\mathsf{y} = 0 \land b$ is unreachable, even though the original result was reachable.

Similarly, the underapproximative rule `Assign` can lead to a path that cannot reach the entry point. Consider the program $x \leftarrow x + 10;\ y \leftarrow x + 4 \,\%\, 7$ with result $y = 4$. For the second part of the program, we have derived presumption $x = 0$ in Example 2.3. However, if x is a natural number, such states are unreachable after execution of $x \leftarrow x + 10$, since 0 is not in the image of +10. We should have proven a presumption where x is at least 10. In context of the larger program, we have instantiated function $\mathbf{g}$ wrongly.

Finally, rule `While` also is underapproximative, as it requires instantiation of an iteration count $\mathbf{n}$. If choosen wrongly, an intermediate presumption may become unreachable. For example, one can always choose a path where $\mathbf{n} = 0$, but the result may only be reachable after the loop body has been executed at least once.

In this section, we discuss approaches to reduce this search space.

3.1 Delaying Underapproximation

Consider again the program $x \leftarrow x + 10;\ y \leftarrow x + 4 \,\%\, 7$ with result $y = 4$. Instead of instantiating $\mathbf{g}$ with a specific image (that may be the wrong choice), we can instantiate $\mathbf{g}$ such that $\mathbf{g}(y) = (y - 4 \,\%\, 7) + 7\mathbf{x}_0$, leaving meta-variable $\mathbf{x}_0$ uninstantiated. We have $x \in \mathsf{image}(\mathbf{g}) \longleftrightarrow 7\mathbf{x}_0 \le x < 8\mathbf{x}_0$. We first derive:

$$\frac{y = 4 \implies y \le 7}{\langle x \,\%7 = 0 \wedge 7\mathbf{x}_0 \le x < 8\mathbf{x}_0 \perp\!\!\!\perp \{y\}\rangle\ y \leftarrow x + 4 \,\%\, 7\ \langle y = 4 \perp\!\!\!\perp \{x\}\rangle}\ \texttt{Assign}$$

Then, the assignment rule is applied again:

$$\frac{7\mathbf{x}_0 \le x < 8\mathbf{x}_0 \implies x \ge 10}{\langle x \,\%7 = 4 \wedge 7\mathbf{x}_0 \le x + 10 < 8\mathbf{x}_0 \perp\!\!\!\perp \{y\}\rangle\ x \leftarrow x + 10\ \langle x \,\%7 = 0 \wedge 7\mathbf{x}_0 \le x < 8\mathbf{x}_0 \perp\!\!\!\perp \{y\}\rangle}\ \texttt{Assign}$$

Combined, this produces:

$$\frac{7\mathbf{x}_0 \le x < 8\mathbf{x}_0 \implies x \ge 10 \qquad y = 4 \implies y \le 7}{\langle x \,\%7 = 4 \wedge 7\mathbf{x}_0 \le x + 10 < 8\mathbf{x}_0 \perp\!\!\!\perp \{y\}\rangle\ x \leftarrow x + 10;\ y \leftarrow x + 4 \,\%\, 7\ \langle y = 4 \perp\!\!\!\perp \{x\}\rangle}\ \texttt{Seq}$$

At this point, we have gone backwards and reached the entry point of the program. We can now instantiate a value for $\mathbf{x}_0$ by asking an SMT solver [12] to find a model where all assumptions hold:

$$\exists \mathbf{x}_0 \cdot \begin{cases} \forall x \cdot 7\mathbf{x}_0 \le x < 8\mathbf{x}_0 \implies x \ge 10 \\ \forall y \cdot y = 4 \implies y \le 7 \end{cases}$$

For example, this may produce $\mathbf{x}_0 = 3$. After instantiation, the derived triple simplifies to:

$$\langle x = 11 \perp\!\!\!\perp \{y\}\rangle\ x \leftarrow x + 10;\ y \leftarrow x + 4 \,\%\, 7\ \langle y = 4 \perp\!\!\!\perp \{x\}\rangle$$

The point here is that by leaving variables uninstantiated, one can traverse the program backwards towards the entry point, and then use an SMT solver to discharge the generated assumptions.

We provide another example of this, showing how an SMT solver can be used to find specific iteration counts. Consider the program $x \leftarrow x + 4;\ W$ with W from Example 2.6 and a result where $x = 16$. We instantiate $\mathbf{P}$ such that $\mathbf{P}(n')$ returns true iff $x = \mathbf{x}_0 + 2n'$. We start from the result going backwards:

$$\frac{\dfrac{}{\langle x = \mathbf{x}_0 + 2n' \perp\!\!\!\perp V\rangle\ x \leftarrow x + 2\ \langle x = \mathbf{x}_0 + 2(n' + 1) \wedge n' < \mathbf{n} \perp\!\!\!\perp V\rangle} \qquad x = 16 \implies x = \mathbf{x}_0 + 2\mathbf{n}}{\langle x = \mathbf{x}_0 \perp\!\!\!\perp V\rangle\ W\ \langle x = 16 \perp\!\!\!\perp V\rangle}$$

Note that meta-variables $\mathbf{x}_0$ and $\mathbf{n}$ must be instantiated with values. However, we can delay this instantiation, since the triple in the assumption can be derived for all $\mathbf{x}_0$ and $\mathbf{n}$. Thus, we first derive:

$$\frac{\mathbf{x}_0 \in \mathsf{image}(+4)}{\langle x + 4 = \mathbf{x}_0 \perp\!\!\!\perp V - x\rangle\ x \leftarrow x + 4\ \langle x = \mathbf{x}_0 \perp\!\!\!\perp V\rangle}$$

The sequence rule can now be applied, producing:

$$\frac{\mathbf{x}_0 \in \mathsf{image}(+4) \qquad x = 16 \implies x = \mathbf{x}_0 + 2\mathbf{n}}{\langle x + 4 = \mathbf{x}_0 \perp\!\!\!\perp V - x\rangle\ x \leftarrow x + 4;\ W\ \langle x = 16 \perp\!\!\!\perp V\rangle}$$

For natural numbers, the image of +4 simply amounts to values greater-equal to 4. This shows that if we instantiate $\mathbf{x}_0$ and $\mathbf{n}$ appropriately, states where $x = 16$ are reachable from states where $x + 4 = \mathbf{x}_0$. We thus ask an SMT solver to find values for $\mathbf{x}_0$ and $\mathbf{n}$ such that the assumptions for both the `While` and `Assign` rule hold:

$$\exists \mathbf{x}_0, \mathbf{n} \cdot \begin{cases} \forall x \cdot x = 16 \implies x = \mathbf{x}_0 + 2\mathbf{n} \\ \mathbf{x}_0 \in \mathsf{image}(+4) \end{cases}$$

The assumptions can be send to an SMT solver, producing a model where $\mathbf{x}_0 = 6$ and $\mathbf{n} = 5$. This shows that states where $x = 16$ are reachable from initial states where $x + 4 = 6$, after 5 iterations of the loop.

3.2 Using Loop Summaries to Derive UIL Triples

Example 2.6 shows that in order to derive a UIL triple for loops, one needs to find a parameterized predicate P that is “backwards inductive”. This neatly can be solved using *loop summaries* [15, 18, 30, 32, 35, 36]. Loop summaries are typically used in symbolic execution to overapproximate the behavior of snippets of code. They can be an effective technique in battling path explosion.

A summary σ_ι is an assignment of symbolic expressions to variables. Each symbolic expression models the value of a variable after ι iterations of a loop. Figure 3 provides an example. Note that not all variables need to be summarized: if it is not possible to find a closed-form expression, the summary may omit the variable. We use V_σ to denote the set of variables summarized by loop summary σ_ι, and $V_{\setminus\sigma}$ to denote the set of variables not summarized.

```
for (; i<z; i++) {
  w += 10;
  x += w + i;
  y = y³;
}
```

$$\begin{aligned}
\sigma_\iota(\mathtt{w}) &:= \mathtt{w} + 10\iota \\
\sigma_\iota(\mathtt{x}) &:= \mathtt{x} + \iota \cdot \mathtt{w} + \tfrac{\iota(\iota+1)}{2} \cdot 10 + \iota \cdot \mathtt{i} + \tfrac{\iota(\iota-1)}{2} \\
\sigma_\iota(\mathtt{i}) &:= \mathtt{i} + \iota \\
\sigma_\iota(\mathtt{z}) &:= \mathtt{z}
\end{aligned}$$

Fig. 3. Example of loop summary.

Instantiation of parameterized predicate $\mathbf{P}$ can occur by substituting in Q variables with their summaries. We use notation $Q[\![\sigma_\iota]\!]$ to denote this, i.e., each variable in V_σ is replaced with its summary, and all other variables becomes $\top$ (unknown). In Q, zero further iterations need to happen. In the generated presumption, $\mathbf{n}$ iterations need to happen. Thus we generate the predicate:

$$Q[\![\sigma_{\mathbf{n}-n'}]\!]$$

The intuition is that after n' iterations from the presumption, $\mathbf{n} - n'$ more iterations need to happen to reach a Q-state. Note that for $n' = \mathbf{n}$, i.e., when 0 more iterations need to happen, we have $Q[\![\sigma_{\mathbf{n}-\mathbf{n}}]\!] = Q$, as long as σ_0 is the identity function. The presumption of the while loop sets n' to 0.

In the same fashion, we can construct $b[\![\sigma_\iota]\!]$ by doing substitutions in branching condition b. This allows to express that each further iteration of the loop body should start in a state where branching condition b holds: $\forall n'' < \mathbf{n} - n' \cdot b[\![\sigma_{n''}]\!]$. Combined, these two expressions provide a presumption that models a state where the loop has been executed n' times and requires $\mathbf{n} - n'$ further iterations to complete. The final iteration is the *first* iteration in which the branching condition does not hold, in all other iterations it does hold.

Given result Q and branching condition b, we construct the instantiation of parameterized predicate $\mathbf{P}$ as follows:

$$\mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(n') \equiv \begin{cases} & Q[\![\sigma_{\mathbf{n}-n'}]\!] \\ \wedge & \forall n'' < \mathbf{n} - n' \cdot b[\![\sigma_{n''}]\!] \end{cases}$$

Definition 3.1. A summary σ is *backwards inductive* for result Q, program C, and branching condition b, notation bw_inductive(σ, Q, C, b), if and only if, σ_0 is the identity function and for all $\mathbf{n}$ and n':

$$\models \langle \mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(n') \perp\!\!\!\perp V_{\setminus\sigma}\rangle\ C\ \langle \mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(n'+1) \wedge n' < \mathbf{n} \perp\!\!\!\perp V_{\setminus\sigma}\rangle$$

Example 3.2. The summary in Figure 3 is backwards inductive. Consider variable i, and a result $i = 100$. A presumption where n' iterations have already occurred, contains $i + \mathbf{n} - n' = 100$. The presumption for iteration $n' + 1$ contains $i + \mathbf{n} - n' - 1 = 100$. Iteration $n' + 1$ is reachable from iteration n'.

Theorem 3.3. *The following derivation is sound:*

$$\frac{\text{bw\_inductive}(\sigma, Q, C, b) \qquad Q \implies \neg b}{\langle \mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(0) \perp\!\!\!\perp V_{\setminus\sigma} \rangle \ \texttt{While}\ b\ \texttt{Do}\ C\ \langle Q \perp\!\!\!\perp V_{\setminus\sigma} \rangle}$$

Proof. The proof follows directly from rule `While` in Figure 1, where we instantiate $\mathbf{P}$ with function $\mathbf{P}^{Q,b}_{\mathbf{n},\sigma}$. The first assumption of that rule follows from soundness of summary σ. We have to prove that $\mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(n') \implies b$ when $n' < \mathbf{n}$. This follows from the fact that the term will include $b[\![\sigma_0]\!]$ which is equivalent to b. The second assumption follows from the fact that: $Q \implies \mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(\mathbf{n})$ which follows directly from the fact that $Q[\![\sigma_0]\!] = Q$. □

Example 3.4. Consider the code and summary in Figure 3. Let Q return true iff $\mathtt{w} > 50 \land \mathtt{i} \geq \mathtt{z}$. We have:

$$\begin{array}{lcl} Q[\![\sigma_{\mathbf{n}-n'}]\!] & = & w + 10(\mathbf{n} - n') > 50 \land i + \mathbf{n} - n' \geq z \\ b[\![\sigma_{n''}]\!] & = & i - n'' < z \\ \mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(n') & = & w + 10(\mathbf{n} - n') > 50 \land \begin{cases} i + \mathbf{n} - n' = z & \text{if} \quad n' < \mathbf{n} \\ i \geq z & \text{if} \quad \text{otherwise} \end{cases} \end{array}$$

We can establish that the summary is backwards inductive by deriving the triple from Definition 3.1 with $V_{\setminus\sigma} = \{\mathtt{y}\}$. Moreover, Q contains $\neg b$. We can thus derive the following, by setting n' to 0 and considering any instantiation of $\mathbf{n}$ such that $\mathbf{n} > 0$:

$$\frac{}{\langle w + 10\mathbf{n} > 50 \land i + \mathbf{n} = \mathtt{z} \land \mathbf{n} > 0 \perp\!\!\!\perp \{\mathtt{y}\} \rangle \ \texttt{While}\ b\ \texttt{Do}\ C\ \langle \mathtt{w} > 50 \land \mathtt{i} \geq \mathtt{z} \perp\!\!\!\perp \{\mathtt{y}\} \rangle}$$

We ask an SMT solver to find values for $\mathbf{n}$, `w`, `i` and `z`, producing 3, 30, 0, and 3. This shows that states where $\mathtt{w} > 50$ and $\mathtt{i} \geq \mathtt{z}$ are reachable from states where $\mathtt{w} > 30$ after 3 iterations, with `i` initially 0.

The above proof strategy may lead to the following questions:

- The definition of "backwards inductive" is dependent on both the result Q and branching condition b. Does that mean that a new summary must be found whenever either of these changes?
- Existing approaches to loop summarization has been done in the context of Hoare Logic and aims to provide regular invariants, instead of "backwards" invariants. Can existing approaches to loop summarization be applied here?

Typically, a loop summary is *reusable* and does not depend on either result Q and branching condition b. Moreover, the techniques to loop summarization as found in existing literature do apply here. To provide arguments for these two answers, we must first formalize the intuition of loop summarization in their traditional context of Hoare Logic.

That intuition is that a summary for a program C shows that this program can be regarded as an unordered series of single static assignments for all summarized variables. The values of all other variables are unknown. To formalize this, we use notation $\overline{e}(s)$ to denote the set of values that symbolic expression e can have in state s. Summary σ *summarizes* program C if and only if:

$$s \xrightarrow{C} s' \implies \forall v \in V_\sigma \cdot s'.v \in \overline{\sigma_1(v)}(s)$$

Revisiting the example in Figure 3, this simply formalizes that the loop body can be formulated as a concurrent (i.e., non-blocking) series of assignments of the form $v \longleftarrow \sigma_1(v)$. For example, Line 2 of the loop body can be rewritten to $\mathtt{x} \longleftarrow \mathtt{x} + \mathtt{w} + 10 + \mathtt{i}$.

Moreover, the summary must be *inductive* for all variables $v \in V_\sigma$, which is defined as:

$$\sigma_{\iota+1}(v) = \sigma_\iota(v)[\![\sigma_1]\!]$$

This property can easily be checked, given a summary. For the summary in Figure 3, we have, e.g.,: $\sigma_{\iota+1}(\mathtt{w}) = \mathtt{w} + 10(\iota + 1) = \sigma_\iota(\mathtt{w})[\![\sigma_1]\!]$. A consequence of an inductive summary is that for any Q we have $Q[\![\sigma_{\iota+1}]\!] = Q[\![\sigma_\iota]\!][\![\sigma_1]\!]$.

Note that the above properties are desirable properties of summaries, regardless of whether they are used in the context of IL or HL. Existing approaches to summarization aim to provide summaries that satisfy these properties, even if they are not explicitly formalized as such.

Finally, we formalize that the summary must at least summarize all variables in branching condition b, and that the branching condition after one iteration implies that it held in the previous one:

$$b[\![\sigma_1]\!] \implies b$$

For the running example, we have $\mathtt{i} + 1 < \mathtt{z} \implies \mathtt{i} < \mathtt{z}$.

With these ingredients, we can formulate the following theorem:

Theorem 3.5. *Let σ be an inductive summary that summarizes program C. Assume $b[\![\sigma_1]\!] \implies b$. Then σ is backwards inductive for any result Q.*

Proof. Let s' be a state that satisfies $\mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(n' + 1)$. We have s' satisfies $Q[\![\sigma_{\mathbf{n}-n'-1}]\!]$. Since program C is a series of assignments, we can apply rule `Assign` per assignment. This provides a state s such that $Q[\![\sigma_{\mathbf{n}-n'-1}]\!][\![\sigma_1]\!]$, where each variable $v \in V_\sigma$ is replaced with $\sigma_1(v)$. Since σ is inductive, this is equivalent to $Q[\![\sigma_{\mathbf{n}-n'}]\!]$.

Moreover, we have s' satisfying $\forall n'' < \mathbf{n} - n' - 1 \cdot b[\![\sigma_{n''}]\!]$. With the same reasoning as above, this implies that state s satisfies $\forall n'' < \mathbf{n} - n' - 1 \cdot b[\![\sigma_{n''+1}]\!]$. This implies $\forall n'' < \mathbf{n} - n' \cdot b[\![\sigma_{n''}]\!]$ using the assumption $b[\![\sigma_1]\!] \implies b$. □

What Theorem 3.5 shows, is that a summary can be derived from loop body C using existing methods from literature: summaries must summarize program C and must be inductive. Such summaries can be reused for any result, and for any branching condition such that $b[\![\sigma_1]\!] \implies b$. The latter assumption is easily checked.

Corollary 3.6. *Let σ be an inductive summary that summarizes program C. The following derivation is sound:*

$$\frac{b[\![\sigma_1]\!] \implies b \qquad Q \implies \neg b}{\langle \mathbf{P}^{Q,b}_{\mathbf{n},\sigma}(0) \perp\!\!\!\perp V_{\setminus\sigma} \rangle \ \mathtt{While}\ b\ \mathtt{Do}\ C\ \langle Q \perp\!\!\!\perp V_{\setminus\sigma} \rangle}$$

4 Underapproximation, Underspecification and Pointers

A typical challenge of backwards symbolic execution is dealing with pointers. The problem is that it may be the case that no information is available on the mutual relation of pointers, which causes path/state space explosion. Consider the code in Figure 4. Clearly, there is a case where pointers p_0 and p_1 are separate and the result can be reached. However, it may also be the case that they alias, and that parameters x and y were equal. Without any further knowledge, there are more cases possible due to partially overlapping pointers: it may be the case that $p_1 = p_0 - 2$, and the last two bytes of x are the reverse of the first two bytes of y (e.g.: `0xAABBCCDD` and `0xCCDDEEFF`, for a little-endian architecture with 4-byte integers). Enumerating all cases when traversing a program backwards is infeasible.

Underapproximation, however, only requires finding some path from which the result is reachable. Underspecification enables to express that we do not care about the actual values of pointers (nor

```
void foo(int* p0, int* p1, int x, int y) {
  *p0 := x;
  *p1 := y;
  // RESULT: *p0 = x
}
```

Fig. 4. Example with pointers.

$$\frac{p_0 \in V' \vee p_1 \in V'}{\langle Q[x/{*p_1}][x/{*p_0}] \wedge p_0 = p_1 \perp\!\!\!\perp V' - \{p_0, p_1\}\rangle \ {*p_0} \leftarrow x \ \langle Q \perp\!\!\!\perp V'\rangle} \ \texttt{Store\_Irrelevant\_Alias}$$

$$\frac{p_0 \in V' \vee p_1 \in V'}{\langle Q[x/{*p_0}] \wedge p_0 \bowtie p_1 \perp\!\!\!\perp V' - \{p_0, p_1\}\rangle \ {*p_0} \leftarrow x \ \langle Q \perp\!\!\!\perp V'\rangle} \ \texttt{Store\_Irrelevant\_Separate}$$

$$\frac{\{p_0, p_1\} \cap V' = \emptyset \qquad Q \implies p_0 = p_1}{\langle Q[x/{*p_1}][x/{*p_0}] \perp\!\!\!\perp V'\rangle \ {*p_0} \leftarrow x \ \langle Q \perp\!\!\!\perp V'\rangle} \ \texttt{Store\_Alias}$$

$$\frac{\{p_0, p_1\} \cap V' = \emptyset \qquad Q \implies p_0 \bowtie p_1}{\langle Q[x/{*p_0}] \perp\!\!\!\perp V'\rangle \ {*p_0} \leftarrow x \ \langle Q \perp\!\!\!\perp V'\rangle} \ \texttt{Store\_Separate}$$

Fig. 5. Proof Rules for Dealing with Store

their mutual relation) as long as the result is triggered. Thus, the following triple holds:

$$\langle p_0 \bowtie p_1 \perp\!\!\!\perp \emptyset\rangle \ \texttt{foo} \ \langle {*p_0} = x \perp\!\!\!\perp \{p_0, p_1\}\rangle$$

All result-states are reachable from a state where pointers p_0 and p_1 were separate (notation: $\bowtie$). The following triple holds as well:

$$\langle x = y \wedge p_0 = p_1 \perp\!\!\!\perp \emptyset\rangle \ \texttt{foo} \ \langle {*p_0} = x \perp\!\!\!\perp \{p_0, p_1\}\rangle$$

All result-states are reachable from a state where pointers p_0 and p_1 were aliassing and $x = y$.

We can extend the proof system by adding the rules for a Store statement (see Figure 5). Rules for Load are similar. If the pointers are underspecified, rules Store_Irrelevant_Alias and Store_Irrelevant_Separate can be applied. They will introduce pointer constraints into the presumption. If the current result Q already contains an aliassing or separation constraint, rules Store_Alias and Store_Separate can be applied. The extended proof system purposefully is incomplete: it does not provide any rules for introducing partially overlapping pointers.

Remark. The rules in Figure 5 are defined for a result Q that only contains dereferences $*p_0$ and $*p_1$. Intuitively, the rules easily extend to results with multiple dereferences, where for each reference a decision on aliassing or separation must be made. That formalization requires much more notational clutter, which is why the rules have been presented like this.

One can traverse a program backward, and choose which rule to apply for each dereference in the result. This is similar to applying rules ITE_Irrelevant_* when branching condition b is underspecified, or instantiating an iteration count $\mathbf{n}$ when dealing with a loop: in each of these cases, underapproximation and underspecification allows one to "pick a path" (see Section 3). Instead of requiring to formulate predicates that model all paths, including paths concerning partially overlapping pointers, one can pick a path and see if the program can be traversed up to the entry point.

$$\begin{tikzcd}
 & \lightning & \mathsf{x} = \mathsf{y} \\
\boxed{\mathsf{p}_0 \leftarrow \mathsf{p}_1} & *\mathsf{p}_1 = \mathsf{y} \land \mathsf{p}_0 \bowtie \mathsf{p}_1 \arrow[u] & \mathsf{x} = \mathsf{y} \land \mathsf{p}_0 = \mathsf{p}_1 \arrow[u] \\
\boxed{*\mathsf{p}_0 \leftarrow \mathsf{x}} & *\mathsf{p}_1 = \mathsf{y} \arrow[u] \arrow[ur, bend right] &
\end{tikzcd}$$

Fig. 6. Backwards exploration up to the entry point.

Example 4.1. Figure 6 provides an example for a program $\mathsf{p}_0 \leftarrow \mathsf{p}_1$; $*\mathsf{p}_0 \leftarrow \mathsf{x}$ and result $*\mathsf{p}_1 = \mathsf{y}$. First, Rule Store_Irrelevant_Separate is applied, picking a path where the pointers are separate. The program is traversed backwards, but we hit a point where none of the rules of the proof system can be applied. Note that Rule Assign cannot be applied, since p_0 is a relevant variable in the intermediate result. Rule Assign_Relevant_Src cannot be applied since the intermediate result is not reachable. We thus backtrack, and instead of picking a path of separation, we pick the aliassing path. This path allows backwards exploration up to the entry point, yielding presumption $\mathsf{x} = \mathsf{y}$.

Example Combining a Loop and Array. Consider the code in Figure 7 which contains a write to memory within a loop. The summary provides symbolic expressions for all variables, and also expresses that after having executed ι iterations, an equal amount of memory writes have happened.

```
for (;i<z;i+=2) {
  j += i+1;
  a[i] = j;
}
// RESULT:
// a[x] = 49 ∧ z ≥ i
```

$$\begin{array}{lcll}
\sigma_\iota(\mathsf{i}) & := & \mathsf{i} + 2\iota & \\
\sigma_\iota(\mathsf{j}) & := & \mathsf{j} + \mathsf{i}\iota + \iota^2 & \\
\sigma_\iota(\mathsf{z}) & := & \mathsf{z} & \\
\sigma_\iota(\mathsf{x}) & := & \mathsf{x} & \\
\sigma_\iota(a[\mathsf{i} + 2\kappa]) & := & \mathsf{j} + \mathsf{i}(\kappa + 1) + (\kappa + 1)^2 & (\forall 0 \le \kappa < \iota)
\end{array}$$

Fig. 7. Example of loop with array, with a summary.

First, we establish that summary σ is inductive. For example, for variable j, we have:

$$\sigma_{\iota+1}(\mathsf{j}) = \mathsf{j} + \mathsf{i}(\iota + 1) + (\iota + 1)^2 = (\mathsf{j} + \mathsf{i}\iota + \iota^2)[\![\sigma_1]\!] = \sigma_\iota(\mathsf{j})[\![\sigma_1]\!]$$

Second we establish that $b[\![\sigma_1]\!] \implies b$, i.e., $\mathsf{i} + 2 < z \implies \mathsf{i} < z$. Corollary 3.6 shows that we can now use summary σ to do substitutions in a result Q.

First, consider a path where all state parts (i.e., variables and memory regions) that the summary writes to are separate from the region $\mathsf{a}[\mathsf{x}]$ and where i and j are initialized with 0. In that case, the presumption contains (assuming array a contains 1-byte elements):

$$\mathsf{a}[\mathsf{x}] = 49 \land (\forall 0 \le \kappa < \mathbf{n} \cdot \mathsf{a} + \mathsf{x} \bowtie \mathsf{a} + 2\kappa)$$

This formulates a presumption that index x points to anywhere outside of the memory of array a accessed during $\mathbf{n}$ iterations, and that $\mathsf{a}[\mathsf{x}]$ already stores value 49.

However, it may also be the case that the summary provides some state part that aliases with region $\mathsf{a}[\mathsf{x}]$. In that case, the presumption will contain:

$$\exists 0 \le \kappa < \mathbf{n} \cdot (\kappa + 1)^2 = 49 \land \mathsf{a} + \mathsf{x} = \mathsf{a} + 2\kappa$$

The iteration count $\mathbf{n}$ should at least be 7, with $\kappa = 6$. This formulates that if there are at least 7 iterations and $\mathtt{x} = 12$, then the result can be reached.

We consider a full formalization of combining loop summaries with memory out of scope. However, the above suggests how the theory in Section 3.2 can be extended. First, a summary no longer is a simple assignment of symbolic expressions to variables, but an assignment of symbolic expressions to state parts. The summary itself should consider separate state parts only. For example, all state parts of the summary in Figure 7 are separate. Then rule `Store_Irrelevant_Alias` then can be used existentially, i.e., there must exist some summarized state part that aliases with the the state part in the result. Rule `Store_Irrelevant_Separate` can be used universally, i.e., there is a path where all summarized state parts are separate from the state part in the result.

5 Putting UIL into Practice

We implemented a predicate transformer based on the proof system in Figures 1 and 5 for the LLVM IR. The implementation, called `Broil` (Backwards Reasoner over Incorrectness Logic) starts at an assertion with some symbolic set of irrelevant variables V. As soon as a variable can either 1.) not be summarized (in case of a loop), or 2.) not be inversed (in case of an assignment), we assume that it was part of V and thus treat it as an irrelevant variable. The first typically happens in case of non-linear loop variables. The second typically happens in case of floating-point operations, since we do not support them. Considering the example in Figure 3 and a result formulating a constraint over variable y, `Broil` substitutes $\top$ for `y` and assumes variable `y` is irrelevant in the result. Thus the following triple could be generated for symbolic set V:

$$\mathtt{y} \in V \wedge \mathtt{i} \notin V \wedge \mathtt{z} \notin V \implies \langle \top = 0 \wedge \mathtt{i} \geq \mathtt{z} \perp\!\!\!\perp V \rangle\ C\ \langle \mathtt{y} = 0 \wedge \mathtt{i} \geq \mathtt{z} \perp\!\!\!\perp V \rangle$$

In other words, no presumption was generated for the result that included $y = 0$, but for the result $\top = 0 \wedge \mathtt{i} \geq \mathtt{z}$.

Once a presumption is generated, conjuncts that evaluates to $\top$ are dropped (through precondition weakening) before sending it to an SMT solver to find a model. In the above example, $\top = 0$ evaluates to $\top$ and thus we ask an SMT solver to find a model for $\mathtt{i} \geq \mathtt{z}$. The presumption has become – significantly – weaker than it would have been if variable y were properly summarized. It may even be the case that an unreachable assertion does generate a presumption. In the above example, if the result states $y = 2$ (which is not a cube), `Broil` will generate the same presumption as above: $\top \wedge \mathtt{i} \geq \mathtt{z}$ and consider the constraint over `y` irrelevant.

`Broil` treats if-statements underapproximatively, i.e., whenever possible it picks a path to see if the entry point of the program can be reached. For assignments and loops, we "zoom in" as little as possible and instead use the proof strategy of delaying underapproximation whenever possible (see Section 3). We implemented a loop summarization technique from scratch that is also capable of summarizing memory accesses to arrays in loops. Our summarization technique leverages existing work [30, 32]. We utilized a Linux machine with Intel Core i7-8086K CPU, 32GB of RAM, and Z3 for SMT solving.

We applied `Broil` to programs from the SV-COMP benchmark suite[1]. The SV-COMP benchmark is specially created for testing and verifying tools. The benchmark provides different categories of programs where each category represents a set of verification tasks. The tasks were contributed by several research and development groups. The programs are indicative of the complexity of programs that is found in real-world programs. For many of the functions in SV-COMP, the ground truth is available in the form of an annotation that tells whether the assertion is expected to be triggerable or not. For the evaluation we narrow the scope to the set of functions F for which the

[1]https://github.com/sosy-lab/sv-benchmarks/

ground truth is available. The ground truth partitions the set of all functions F into two sets F_U and F_R for functions with unreachable and reachable assertions. This set contains *all* SV-COMP programs that fall within scope, i.e., non-concurrent programs with source-level assertions that are compilable to LLVM. The benchmarks includes various types of loops, data structures and arrays. Additionally, we hand-crafted 42 micro-benchmark programs that are made to demonstrate different cases related to pointers, loops, and arrays. We named and categorized the hand-crafted micro-benchmarks according to the types of cases they handle: hand-crafted loop, hand-crafted mixed, and hand-crafted pointer. The main reason for adding these benchmarks on top of the SV-COMP benchmark suite is to expose `Broil` to programs where pointers influence the reachability of assertions.

The purpose of the prototype implementation is to demonstrate with an implementation how UIL can be put to practice, and how precise a straightforward implementation is. This heavily depends on which loop summarization technique is used: the more variables can be summarized, the fewer $\top$-substitutions will happen. It also depends on how hard it is to find a function $\mathbf{g}$ that inverses a computation. We do not aim to demonstrate scalability: the SV-COMP benchmarks generally consists of small challenges. All running times are below a hundred seconds.

For functions in F_U, we expect that when asking Z3 to find a model where all assumptions made during presumption generation hold, it will produce UNSAT. So for this case, we report the percentage of functions in F_U for which we have generated unsatisfiable assumptions. We call the percentage of functions with unreachable assertions for which `Broil` generates unsatisfiable assumptions the *UNSATSuccesRate*. A score of 100% would mean that none of the unreachable assertions lead to a presumption. If lower, then this is due to $\top$-substitution.

For functions in F_R preciseness intuitively means that the presumption contains strong constraints over variables. We therefore measure how many tries a fuzzer takes to trigger the assertion without any further guidance (T). We also measure how many of these tries satisfied the presumption (T_{SAT}). We compute the ratio $\frac{T_{SAT}}{T}$, which we call the *reduction factor*. If the reduction factor is close to 1, then the presumption contained weak constraints. This would happen if all presumptions are trivial. If the number is close to 0 then the presumption is strong. In other words, if this factor is $\frac{1}{f}$, then the presumption reduces the space of initial states f times.

We have used the LibFuzzer as fuzzer [1]. LibFuzzer, part of the LLVM project, is a coverage-guided fuzzing engine that links with libraries under test to maximize code coverage [31]. It operates by feeding fuzzed inputs via a specific entry point, tracking code areas reached, and generating mutations based on a corpus of sample inputs. We thus measure for each try starting from an entry point whether the inputs (i.e., the initial states) satisfy the presumption generated by `Broil`.

Table 2 presents results. On average, the percentage of unsatisfiability for unreachable assertions, the UNSATSuccesRate, is 64%. On average, the reduction factor for reachable assertions is 0.20. Note that the categories of SV-COMP neatly expose both strengths and weaknesses of our prototype: for examples with complex loops (such as category loop-crafted) the reduction factor is worse than for programs with simple ones (such as category array-programs). Also, there are cases where the assertions are *necessarily reachable*, i.e., *all* initial states lead to the assertion. Those cases have a reduction factor of 1.00, but that is the best result possible. This holds for six out of the total of 310 functions.

The prototype `Broil` presented in this paper is not intended as an improvement over state-of-the-art tools such as Pulse-X [22] or Infer [8]. Indeed, it reports a relatively high false positive rate, due to $\top$-substitution. Its sole purpose is to evaluate through an implementation the precision of using UIL with loop summaries. This produced insight into current limitations, and directions for

Table 2.
Benchmarks ran by `Broil`

Category	$\lvert F \rvert$	$\lvert F_U \rvert$	UNSATSuccesRate (%)	$\lvert F_R \rvert$	Reduction Factor
array-cav19	13	13	62	0	-
array-crafted	20	20	90	0	-
array-example	87	59	81	18	0.004
array-fpi	137	69	28	68	0.086
array-industry-pattern	12	6	100	3	0.007
array-lopstr16	10	10	40	0	-
array-multidimensional	20	20	75	0	-
array-patterns	30	30	100	0	-
array-programs	16	14	21	2	0.0008
array-tiling	25	23	43	2	-
bitvector	27	23	26	4	0.8
bitvector-loops	3	0	-	1	1.00
float-benchs	58	48	0	2	0.01
floats-esbmc-regression	39	35	17	0	-
forester-heap	38	21	10	12	0.027
ldv-regression	35	25	8	5	0.003
list-properties	11	5	60	5	1.00
loop-acceleration	36	20	50	6	0.077
loop-crafted	7	6	83	1	1.00
loop-floats-scientific-comp	7	4	0	3	1.00
loop-industry-pattern	2	2	0	0	-
loop-invariants	7	7	0	0	-
loop-invgen	9	8	63	1	0.33
loop-lit	11	10	30	1	1.00
loop-new	2	2	0	0	-
loop-simple	0	0	-	0	-
loop-zilu	22	22	27	0	-
loops	48	26	62	18	0.19
loops-crafted-1	48	40	33	0	-
nla-digbench	32	27	33	5	0.21
nla-digbench-scaling	436	285	46	146	0.26
uthash-2.0.2	126	126	90	0	-
hand_loop	9	0	-	4	0.02
hand_mixed	2	0	-	2	0.045
hand_pointer	2	2	100	0	-

future research. The high false positive rate is largely due to 1.) coarse loop summaries, and 2.) lack of support for floating point operations. In order to make a scalable bug-finding tool that is formally underapproximative and based on a formal logic such as UIL, we argue the next step should be to leverage function summarization on top of loop summarization such as suggested by Godefroid [16]. Moreover, the tool should be sufficiently efficient so that it can improve state-of-the-art fuzzing tools by reducing the initial state space to be explored.

6 Related Work and Conclusion

Program logics [14, 19] and predicate transformation [13] have been a base for program verification for decades. This paper is heavily inspired by incorrectness logic [27]. Table 3 presents an overview, partially inspired by the taxonomy of Ascari et al. [2], as well as the overview provided by Ball et al. [4]. Each logic has its own merits, use cases, and challenges.

Table 3. Program Logics: HL = Hoare Logic, IL = Incorrectness Logic, NC = Necessary Conditions, BR = Backwards Reasoning. pre is the weakest possible pre of a relation. Note that both definitions of *may*-transitions are logically equivalent.

Definition	**Name(s)**	**Intuition**	**BR**
$\mathrm{post}[C](P) \subseteq Q$	HL [19]	P-states fail or must lead to Q-states.	✓
$P \subseteq \mathrm{pre}[C](Q)$	Lisbon Triples Possible Correctness [20] Sufficient IL [2] Backwards Underappr. [25] *must*$^+$-transition [4]	P-states can lead to Q-states.	✓
$\mathrm{pre}[C](Q) \subseteq P$	NC [2, 10]	reachable Q-states must come from a P-state.	✓
$Q \subseteq \mathrm{post}[C](P)$	IL [27] Reverse HL [34] *must*$^-$-transition [4]	Q-states can come from a P-state.	✗
$P \cap \mathrm{pre}[C](Q) \neq \emptyset$ $Q \cap \mathrm{post}[C](P) \neq \emptyset$	*may*-transition [4]	Some P-state can lead to a Q-state.	✓

Table 3 shows that – except for IL – all logics permit backwards reasoning: generating a precondition (or presumption) from a postcondition (e.g., an assertion in the program). Backwards reasoning for HL is well-known. With respect to NC, Cousot et al. define and address the problem of generating *necessary preconditions*: infer a precondition such that, if violated, ensures the program will always be incorrect [10]. For Lisbon triples, Möller et al. suggest a Kleene algebra that can serve as a foundation for backwards reasoning [25] (even though they do not flesh this idea out fully). Backwards reasoning over *may*-transitions essentially boils down to standard backwards model checking of reachability properties [9]. For IL, however, a backwards reasoning system does not exist. This is exactly the gap that this paper aims to address.

Since O'Hearn's paper on IL, that logic has been extended and used in various ways. Raad et al. combine IL with separation logic [28]. This adds program constructs such as `alloc` and `free` to the programming language. The predicates are extended with constructs that convey that a certain value is an addressable location in memory, but also that it has been *deallocated*, as well as *ownership* of the location. Section 4 of this paper adds basic `Load` and `Store` program constructs, but does not deal with dynamic memory management. The work of Raad et al. thus strongly complements our contribution. Later, Raad et al. extended their Incorrectness Separation Logic with concurrency [29]. The resulting logic can be used for bug-catching in concurrent programs, i.e., race detection, deadlock detection, and memory safety error detection. Both logics have been proven sound, but no statements on completeness are made. Another extension of IL is Vanegue's Adversial Logic [33], that extends IL with an adversary. It can be used to analyze security vulnerabilities due to interaction between a program and its environment.

Static analysis tools and bug finders based on underapproximation, such as KLEE [7] and DART [17], do – ideally – not suffer from false positives. Le et al. provide Pulse-X, a tool for underapproximative reasoning based on ISL which rigorously focuses on not reporting false positives [22]. They show that Pulse-X scales, and has performance comparable to tools such as Coverity or Infer. Those tools generally leverage modular reasoning at the cost of false positives (and thus not being truly underapproximative).

Recently, efforts have been undertaken to unify reasoning over correctness and incorrectness within a single framework [6, 24, 25, 37]. Outcome Logic concerns triples that generalize the predicates used as pre- and postcondition to predicates over *outcome monoids* (e.g., sets of states or probability distributions) [37]. Zilberstein et al. then generalize this further to semirings, and also combine OL with separation logic [38]. Li et al. combine OL with termination reasoning [23]. Consider for sake of explanation the case where the outcome monoid is instantiated as "a set of states". Without further special logical operators, the definition of an Outcome Triple is akin to the conjunction of a Hoare and a Lisbon triple. However, the structure of Outcome Logic allows the introduction of an *outcome conjunction* operator $\oplus$ that allows to specify that two outcomes are possible. That operator also introduces underapproximation: $Q \oplus T$ formulates that outcomes according to predicate Q are possible, as well as trivial ones (T is the trivial outcome). Note, however, that the notion of underapproximation in their paper essentially is postcondition weakening: one can "drop outcomes" by considering them to be trivial. Dually, underapproximation in IL is based on postcondition strengthening: one can "drop outcomes" by considering them to be false. An interesting future endeavor would be to explore the dual of Outcome Logic: a logic whose triples relate to a conjunction over IL Triples and Necessary Conditions.

O'Hearn advocated the use of formal logic to reason over program incorrectness, giving bug-finding tools the same mathematical rigor that underlies verification tools. Incorrectness Logic is an underapproximative logic that allows to specify as a result a set of possible outcomes. It aims at proving that a program can produce at least the desired results. For regular IL, it is not necessarily possible to do backwards reasoning: transform a result into a presumption. We show that if one introduces the concept of underspecification, then there exists a sound backwards reasoning proof system for IL, complete for a subset of presumptions. This means that one can derive a presumption if and only if all result-states are reachable from some presumption-state. We think that IL, combined with separation logic, loop- and function summaries, and underspecification can be the logical foundation for scalable and applicable bug-finding tools, and more generically for tools that aim to prove that a program can at least produce the desired results.

Data-Availability Statement

All implementations, formal proofs (formalized in Isabelle/HOL) and results are publicly available at: https://doi.org/10.5281/zenodo.13970860

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which have greatly improved the paper. This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028. Zhoulai Fu's work is also supported in part by the National Research Foundation of Korea (NRF) Grant RS-2022-00165647.

References

[1] [n. d.]. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2024-01.

[2] Flavio Ascari, Roberto Bruni, Roberta Gori, and Francesco Logozzo. 2023. Sufficient Incorrectness Logic: SIL and Separation SIL. *arXiv preprint arXiv:2310.18156* (2023). https://doi.org/10.48550/arXiv.2310.18156

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39. https://doi.org/10.1145/3182657

[4] Thomas Ball. 2004. A theory of predicate-complete test coverage and generation. In *International Symposium on Formal Methods for Components and Objects*. Springer, 1–22. https://doi.org/10.1007/11561163_1

[5] Clemens Ballarin. 2003. Locales and locale expressions in Isabelle/Isar. In *International Workshop on Types for Proofs and Programs*. Springer, 34–50. https://doi.org/10.1007/978-3-540-24849-1_3

[6] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A logic for locally complete abstract interpretations. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470608

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[8] Cristiano Calcagno and Dino Distefano. 2011. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*. Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[9] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263. https://dl.acm.org/doi/10.1145/5397.5399

[10] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 128–148. https://doi.org/10.1007/978-3-642-35873-9_10

[11] Jeremy Dawson. 2009. Isabelle theories for machine words. *Electronic Notes in Theoretical Computer Science* 250, 1 (2009), 55–70. https://doi.org/10.1016/j.entcs.2009.08.005

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (aug 1975), 453–457. https://doi.org/10.1145/360933.360975

[14] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. https://doi.org/10.1007/978-94-011-1793-7_4

[15] Florian Frohn. 2020. A calculus for modular loop acceleration. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 58–76. https://doi.org/10.48550/arXiv.2001.01516

[16] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 47–54. https://doi.org/10.1145/1190215.1190226

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223. https://doi.org/10.1145/1064978.1065036

[18] Patrice Godefroid and Daniel Luchaup. 2011. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 23–33. https://doi.org/10.1145/2001420.2001424

[19] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[20] Charles Antony Richard Hoare. 1978. Some properties of predicate transformers. *Journal of the ACM (JACM)* 25, 3 (1978), 461–480. https://doi.org/10.1145/322077.322088

[21] Stephen Cole Kleene. 1952. *Introduction to Metamathematics*. P. Noordhoff N.V., Groningen.

[22] 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. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.5281/zenodo.6342311

[23] James Li, Noam Zilberstein, and Alexandra Silva. 2024. Total Outcome Logic: Proving Termination and Nontermination in Programs with Branching. *arXiv preprint arXiv:2411.00197* (2024). https://doi.org/10.48550/arXiv.2411.00197

[24] Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. 19:1–19:27. https://doi.org/10.4230/LIPIcs.ECOOP.2023.19

[25] Bernhard Möller, Peter O'Hearn, and Tony Hoare. 2021. On algebra of program correctness and incorrectness. In *Relational and Algebraic Methods in Computer Science: 19th International Conference, RAMiCS 2021, Marseille, France, November 2–5, 2021, Proceedings 19*. Springer, 325–343. https://doi.org/10.1007/978-3-030-88701-8_20

[26] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer. https://doi.org/10.1007/3-540-45949-9_5

[27] Peter W O'Hearn. 2019. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. https://doi.org/10.1145/3371078

[28] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local reasoning about the presence of bugs: Incorrectness Separation Logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*. Springer, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

[29] Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.* 6, POPL, Article 34 (jan 2022), 29 pages. https://doi.org/10.1145/3498695

[30] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 225–236. https://doi.org/10.1145/1572272.1572299

[31] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. 157–157. https://doi.org/10.1109/SecDev.2016.043

[32] Jan Strejček and Marek Trtík. 2012. Abstracting path conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 155–165. https://doi.org/10.1145/2338965.2336772

[33] Julien Vanegue. 2022. Adversarial Logic. In *Static Analysis*, Gagandeep Singh and Caterina Urban (Eds.). Springer Nature Switzerland, Cham, 422–448. https://doi.org/10.1007/978-3-031-22308-2_19

[34] Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare logic. In *International Conference on Software Engineering and Formal Methods*. Springer, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

[35] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 61–72. https://doi.org/10.1145/2950290.2950340

[36] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 188–198. https://doi.org/10.1145/2771783.2771815

[37] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 522–550. https://doi.org/10.1145/3586045

[38] Noam Zilberstein, Angelina Saliling, and Alexandra Silva. 2024. Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 276–304. https://doi.org/10.1145/3649821

Received 2024-07-10; accepted 2024-11-07