Encode the ∀∃ Relational Hoare Logic into Standard Hoare Logic

SHUSHU WU, Shanghai Jiao Tong University, China
XIWEI WU, Shanghai Jiao Tong University, China
QINXIANG CAO, Shanghai Jiao Tong University, China

Verifying a real-world program's functional correctness can be decomposed into (1) a refinement proof showing that the program implements a more abstract high-level program and (2) an algorithm correctness proof at the high level. Relational Hoare logic serves as a powerful tool to establish refinement but often necessitates formalization beyond standard Hoare logic. Particularly in the nondeterministic setting, the ∀∃ relational Hoare logic is required. Existing approaches encode this logic into a Hoare logic with ghost states and invariants, yet these extensions significantly increase formalization complexity and soundness proof overhead. This paper proposes a generic encoding theory that reduces the ∀∃ relational Hoare logic to standard (unary) Hoare logic. Precisely, we propose to redefine the validity of relational Hoare triples while preserving the original proof rules and then encapsulate the ∀∃ pattern within assertions. We have proved that the validity of encoded standard Hoare triples is equivalent to the validity of the desired relational Hoare triples. Moreover, the encoding theory demonstrates how common relational Hoare logic proof rules are indeed special cases of standard Hoare logic proof rules, and relational proof steps correspond to standard proof steps. Our theory enables standard Hoare logic to prove ∀∃ relational properties by defining a predicate Exec, without requiring modifications to the logic framework or re-verification of soundness.

CCS Concepts: • **Theory of computation → Logic and verification**; **Hoare logic**; *Denotational semantics*.

Additional Key Words and Phrases: encoding, relational Hoare logic, program refinement

ACM Reference Format:
Shushu Wu, Xiwei Wu, and Qinxiang Cao. 2025. Encode the ∀∃ Relational Hoare Logic into Standard Hoare Logic. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 360 (October 2025), 28 pages. https://doi.org/10.1145/3763138

1 Introduction

Hoare Logic [18, 22] is widely used to prove the functional correctness of programs formally. Nevertheless, when dealing with complex programs, it is often more convenient to decompose the proof into two parts [4, 27]: (1) a refinement proof showing that the concrete low-level program refines a more abstract high-level program and (2) an algorithm correctness proof of the high-level program. This verification mode [1] enables significant proof reuse. For instance, once the algorithm correctness proof of a high-level program is established, it can be reused across different low-level implementations; likewise, once a refinement proof is established, it can be leveraged for new usage scenarios focusing on different aspects of algorithm correctness. To support proofs that relate two

[1]This mode differs from verification tools like Why3, which rely on ghost code to help verify properties of real programs. A detailed discussion on these tools is provided in Sec. 9.2.

Authors' Contact Information: Shushu Wu, Shanghai Jiao Tong University, Shanghai, China, Ciel77@sjtu.edu.cn; Xiwei Wu, Shanghai Jiao Tong University, Shanghai, China, yashen@sjtu.edu.cn; Qinxiang Cao, Shanghai Jiao Tong University, Shanghai, China, caoqinxiang@gmail.com.

ACM 2475-1421/2025/10-ART360
https://doi.org/10.1145/3763138

programs, relational Hoare logic [8] has been introduced and is widely used to establish program refinement [19, 20, 31, 40]. In practice, one might formalize a program logic in a proof assistant like Rocq for deductive verification. However, the verification mode above requires formalizing both standard Hoare logic (for functional correctness of simple programs) and relational Hoare logic (for refinement proofs), each with its own soundness proof, which can become cumbersome. Then, to reason both single- and multi-program properties in a unified logic framework, researchers have explored how to encode relational Hoare logic into standard Hoare logic.

A notable technique is self-composition [7, 39], which reduces the ∀∀ relational Hoare logic, also known as *2-safety*, to standard Hoare logic. The ∀∀ logic requires that for all pairs of program executions, if their initial states satisfy the precondition, then any terminal states must satisfy the postcondition. For deterministic programs, this logic is sufficient to capture refinement. Nevertheless, when dealing with nondeterministic programs, the ∀∀ logic cannot express program refinement. Instead, program refinement is captured through the ∀∃ relational Hoare logic. This logic requires that for any execution of the low-level program, there exists a corresponding execution of the high-level program such that the relational postcondition holds. The ∀∃ relational Hoare logic can be encoded into a Hoare logic augmented with ghost states and invariants [20, 30, 38, 42]. In this logic, the judgment is defined in a $(\forall\exists)^\omega$ pattern [2]: for any update of the physical state, there exists a way to update the ghost state such that the invariant holds, and the two states can continue to be updated as described before or terminate at states satisfying the postcondition. While powerful, this Hoare logic with ghost states and invariants is significantly more complex to formalize and prove sound compared to standard Hoare logic. To our knowledge, only a few frameworks provide machine-checked frameworks for Hoare logic with ghost states and invariants, most notably Iris [24] and the Verified Software Toolchain (VST) [3]. Even in the Iris documentation, the presentation begins with a weakest precondition definition [3] without ghost states and adds ghost states with view shifts only later. Similarly, VST initially lacks ghost reasoning and only supports it after extensive extensions to its memory model and a subsequent re-verification of soundness.

Before this paper, it remains unknown how the ∀∃ relational Hoare logic can be reduced to standard Hoare logic. Although this question may appear purely theoretical, such a reduction could enable standard Hoare logic to reason ∀∃ properties and thus have significant practical implications for simplifying verification frameworks and proofs. To answer this question, this paper explores the theoretical connections between the ∀∃ relational Hoare logic and standard Hoare logic and presents a generic encoding theory. This encoding theory shows that a lightweight extension to standard Hoare logic—by defining a $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$ predicate and its related rules—suffices to achieve relational reasoning without modifying the framework or redoing soundness proof.

1.1 Known Theoretical Connection: Similarities in Proof Rules

Various forms of judgments have been proposed in relational Hoare logic. We develop our encoding theory based on relational Hoare triples [4], which use *program-as-resource* assertions [30, 41, 42]. This concept originates from concurrent program verification and can be applied to verify sequential programs. We choose relational triples to explore encoding because they share a similar set of proof rules with standard Hoare logic. The validity of relational Hoare triples is defined as follows:

Definition 1 (Relational Hoare Triples). *As shown in Fig. 1, relational Hoare triple* $\langle \mathcal{P}\rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q}\rangle$ *is valid if given any initial states* $(\sigma_1^{\mathrm{L}},\sigma_1^{\mathrm{H}})$ *and a high-level statement* c_1^{H} *such that* $(\sigma_1^{\mathrm{L}},\sigma_1^{\mathrm{H}},c_1^{\mathrm{H}}) \models \mathcal{P}$,

[2] Detailed explanation for this $(\forall\exists)^\omega$ pattern can be found in Sec. 9.

[3] Iris uses weakest preconditions as the underlying logical primitive; Hoare triples can be defined using weakest preconditions.

[4] Some frameworks choose to employ different judgments like Hoare quadruples. However, a Hoare quadruple $\{\mathbb{P}\}\ c^{\mathrm{L}} \preceq c^{\mathrm{H}}\ \{\mathbb{Q}\}$ can be transformed into a equivalent relational Hoare triple $\langle \mathbb{P} \wedge [c^{\mathrm{H}}]\rangle\ c^{\mathrm{L}}\ \langle \mathbb{Q} \wedge [\mathsf{skip}]\rangle$.

for any final state σ_2^{L} after execution of the low-level statement c^{L}, there exists a multi-step transition from $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}})$ to $(\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ such that $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models Q$.

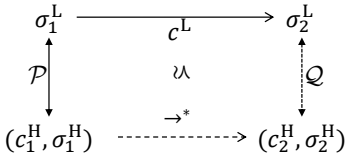

Fig. 1. Relational Hoare Triples

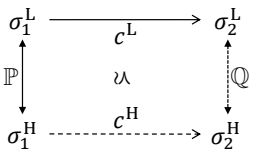

Fig. 2. Program Refinement

To express the refinement between a low-level command c^{L} and a high-level command c^{H}, as illustrated in Fig. 2, we lift the binary assertions $\mathbb{P}$ and $\mathbb{Q}$ [5], as well as the high-level program c^{H}, into program-as-resource assertions over triples $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}})$.

Definition 2 (Assertion Lifting). *Any High-level program c_0^{H}, binary assertion $\mathbb{P}$, unary assertions P^{L} and P^{H} (about low-level and high-level program states respectively) are lifted as follows:*

(1) $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \models [c_0^{\mathrm{H}}]$ *iff.* $c^{\mathrm{H}} = c_0^{\mathrm{H}}$
(2) $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \models \mathbb{P}$ *iff.* $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}) \models \mathbb{P}$
(3) $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \models \lfloor P^{\mathrm{L}} \rfloor$ *iff.* $\sigma^{\mathrm{L}} \models P^{\mathrm{L}}$
(4) $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \models \lceil P^{\mathrm{H}} \rceil$ *iff.* $\sigma^{\mathrm{H}} \models P^{\mathrm{H}}$

We include unary assertion liftings for completeness, though they become more relevant later when we discuss proof rules. With these lifted assertions, the refinement between c^{L} and c^{H} can be described by the triple $\langle \mathbb{P} \wedge [c^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle \mathbb{Q} \wedge [\mathtt{skip}] \rangle$ [6], where skip denotes that the high-level program has terminated. Consider the bitmask/set example in Fig. 3: a low-level bitmask program that records

bit_mask	set_union	A relational triple
$\mathrm{x}^{\mathrm{L}} := 0;$	$\mathrm{s}^{\mathrm{H}} := \{\ \};$	$\langle$[set_union]$\rangle$
$\mathrm{x}^{\mathrm{L}} := \mathrm{x}^{\mathrm{L}} \mid (1 << a_0);$	$\mathrm{s}^{\mathrm{H}} := \mathrm{s}^{\mathrm{H}} \cup \{a_0\};$	bit_mask
$\mathrm{x}^{\mathrm{L}} := \mathrm{x}^{\mathrm{L}} \mid (1 << a_1);$	$\mathrm{s}^{\mathrm{H}} := \mathrm{s}^{\mathrm{H}} \cup \{a_1\};$	$\langle \mathrm{x}^{\mathrm{L}} = \Sigma_{a \in \mathrm{s}^{\mathrm{H}}} 2^a \wedge [\mathtt{skip}] \rangle$

Fig. 3. The Bitmask/Set Example

constants a_0 and a_1, and a high-level set program with similar functionality. The refinement is then expressed by the triple $\langle$[set_union]$\rangle$ bit_mask $\langle \mathrm{x}^{\mathrm{L}} = \Sigma_{a \in \mathrm{s}^{\mathrm{H}}} 2^a \wedge [\mathtt{skip}] \rangle$, where the postcondition states that x^{L} encodes the elements in s^{H} as a sum of powers of 2, and the precondition is trivial (i.e., True) and therefore omitted. This example serves as a running illustration throughout the section.

On the other hand, standard Hoare logic commonly reasons about the safety property of one program, and the validity for standard Hoare triples is defined as follows:

Definition 3 (Standard Hoare Triples). *Standard Hoare triple $\{P\}\ c\ \{Q\}$ is ∀-valid [7] if, for any initial state $\sigma_1 \models P$ and for any final state σ_2 after executing program c, we have $\sigma_2 \models Q$.*

[5] For clarity, this paper uses different fonts to distinguish assertions as follows.

Notation	P	$\mathbb{P}$	$\mathcal{P}$	P^{L}	P^{H}
Meaning	Unary assertion	Binary assertion	Program-as-resource assertion	Low-level assertion	High-level assertion

[6] While program-as-resource assertions are common in separation logic frameworks, $[-]$ can adapt to a separating context.
[7] Standard Hoare triples are ∀-valid for nondeterministic programs and we use "∀-valid" to distinguish them from the angelic triple in rule High-Focus.

$$\textsc{Rel-Seq}\ \frac{\langle \mathcal{P}\rangle\ c_1^{\mathrm{L}}\ \langle \mathcal{R}\rangle \qquad \langle \mathcal{R}\rangle\ c_2^{\mathrm{L}}\ \langle \mathcal{Q}\rangle}{\langle \mathcal{P}\rangle\ c_1^{\mathrm{L}}; c_2^{\mathrm{L}}\ \langle \mathcal{Q}\rangle} \qquad \textsc{Seq}\ \frac{\vdash_\forall \{P\}\ c_1\ \{R\} \qquad \vdash_\forall \{R\}\ c_2\ \{Q\}}{\vdash_\forall \{P\}\ c_1; c_2\ \{Q\}}$$

$$\textsc{Rel-ExIntro}\ \frac{\forall a.\ \langle \mathcal{P}(a)\rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q}\rangle}{\langle \exists a. \mathcal{P}(a)\rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q}\rangle} \qquad \textsc{ExIntro}\ \frac{\forall a.\ \vdash_\forall \{P(a)\}\ c\ \{Q\}}{\vdash_\forall \{\exists a. P(a)\}\ c\ \{Q\}}$$

$$\textsc{High-Focus}\ \frac{\vdash_\exists \{P^{\mathrm{H}}\}\ c_1^{\mathrm{H}}\ \{R^{\mathrm{H}}\} \qquad \langle \lfloor F^{\mathrm{L}}\rfloor \wedge \lceil R^{\mathrm{H}}\rceil \wedge [c_2^{\mathrm{H}}]\rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q}\rangle}{\langle \lfloor F^{\mathrm{L}}\rfloor \wedge \lceil P^{\mathrm{H}}\rceil \wedge [c_1^{\mathrm{H}}; c_2^{\mathrm{H}}]\rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q}\rangle} \qquad \textsc{Conseq-Pre}\ \frac{P \Rightarrow R \qquad \vdash_\forall \{R\}\ c\ \{Q\}}{\vdash_\forall \{P\}\ c\ \{Q\}}$$

Fig. 4. Proof Rules in Relational and Standard Hoare Logic

Therefore in comparison a standard Hoare judgment presents a ∀-structure, while to illustrate program refinement a relational judgment presents a ∀∃ pattern as shown in Def. 1.

Despite this semantic difference, it is known that ∀∃ relational Hoare logic has similar proof rules to standard Hoare logic's rules, as depicted in Fig. 4. In Transfinite Iris [37] the rule Rel-Seq [8] can be derived to evaluate the currently focused statement in a manner similar to the reasoning employed by the rule Seq in standard Hoare logic. Besides, relational Hoare logic frameworks support the typical rule Rel-ExIntro to introduce the existential variables in preconditions, which corresponds to the rule ExIntro. In real program verification, relational proofs often proceed by focusing on and evaluating one side—either the low-level or high-level program—at a time [20]. To support such reasoning, many frameworks represent assertions in a *decomposed form* that separates the components related to the low-level state, high-level state, and high-level program [19, 43]:

$$\exists\, \vec{a}.\, B(\vec{a}) \wedge \lfloor P^{\mathrm{L}}(\vec{a})\rfloor \wedge \lceil P^{\mathrm{H}}(\vec{a})\rceil \wedge [c^{\mathrm{H}}]$$

Here, $\vec{a}$ represents a list of existential logical variables, and B is a pure logical assertion used to constrain $\vec{a}$. The unary low-level assertion P^{L} and high-level assertion P^{H} are both lifted to program-as-resource assertions. For bitmask/set example, the relational triple is then rewritten into

$$\langle [\texttt{set\_union}]\rangle\ \texttt{bit\_mask}\ \langle \exists\, l.\, \lfloor \mathrm{x}^{\mathrm{L}} = \sum\nolimits_{a\in l} 2^a\rfloor \wedge \lceil \mathrm{s}^{\mathrm{H}} = l\rceil \wedge [\mathrm{skip}]\rangle. \tag{1}$$

Decomposed assertions enable focusing rules, such as High-Focus to angelically evaluate the currently focused high-level statement. Here the triple $\vdash_\exists \{P^{\mathrm{H}}\}\ c_1^{\mathrm{H}}\ \{R^{\mathrm{H}}\}$ represents that if an initial state $\sigma_1^{\mathrm{H}} \models P^{\mathrm{H}}$, then there exists a final state σ_2^{H} after executing program c such that $\sigma_2^{\mathrm{H}} \models R^{\mathrm{H}}$. As shown in Fig. 5a, the high-level assignment $\mathrm{s}^{\mathrm{H}} := \{\}$ can be independently evaluated, reducing the precondition to $\lceil \mathrm{s}^{\mathrm{H}} = \emptyset\rceil \wedge [\mathrm{s}^{\mathrm{H}} := \mathrm{s}^{\mathrm{H}} \cup \{a_0\}; \cdots]$. In view of the low-level program, the rule High-Focus weakens the precondition, similar to the rule Conseq-pre in standard Hoare logic.

1.2 Preview of Our Encoding Theory: From Relational to Standard Reasoning

This paper presents an encoding theory that defines assertion encoding $(\!|-|\!)_X$ to transform program-as-resource assertions into unary assertions. This encoding guarantees that the validity of encoded standard Hoare triples is equivalent to the validity of corresponding relational Hoare triples.

[8] In their paper, the programming language is a higher-order functional language. We depict these rules for an imperative language to explicitly compare with standard Hoare logic.

```
// ⟨[s^H := { }; s^H := s^H ∪ {a_0}; ; ...]⟩
// high-level step
// ⟨⌈s^H = ∅⌉ ∧ [s^H := s^H ∪ {a_0}; ...]⟩
x^L := 0;
// low-level step
// ⟨⌊x^L = 0⌋ ∧ ⌈s^H = ∅⌉ ∧ [s^H := s^H ∪ {a_0}; ...]⟩
...
```

(a) Part of Proof Based on Relational Hoare Triples

```
// {Exec_X(True, s^H := { }; s^H := s^H ∪ {a_0}; ...)}
// consequence rule to update the predicate
// {Exec_X(s^H = ∅, s^H := s^H ∪ {a_0}; ...)}
x^L := 0;
// sequencing rule to evaluate low-level assignment
// {Exec_X(s^H = ∅, s^H := s^H ∪ {a_0}; ...) ∧ x^L = 0}
...
```

(b) Part of Proof Based on Standard Hoare Triples

Fig. 5. Relational and Standard Proofs for the Bitmask/Set Example

Theorem 4 (Encoding Relational Triples). *For any low-level statement c^{L} and program-as-resource assertions $\mathcal{P}$ and $\mathcal{Q}$, the relational Hoare triple $\langle \mathcal{P} \rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q} \rangle$ is valid if and only if, given any assertion X on high-level states, the standard Hoare triple $\{(\!|\mathcal{P}|\!)_X\}\ c^{\mathrm{L}}\ \{(\!|\mathcal{Q}|\!)_X\}$ is $\forall$-valid.*

Notably, the variable X serves only as a necessary placeholder. In subsequent discussions, we will see that it does not play any actual role in the encoded relational rules and proofs.

To apply this encoding in practice, our encoding theory also introduces a predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$ to transform decomposed program-as-resource assertions into decomposed unary assertions:

$$(\!|\exists \vec{a}.\, B(\vec{a}) \wedge \lfloor P^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}]|\!)_X \iff \exists \vec{a}.\, B(\vec{a}) \wedge P^{\mathrm{L}}(\vec{a}) \wedge \mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$$

This equivalence shows that the execution constraints of the high-level program can be encoded into a pure logical predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$, while preserving the low-level assertion unchanged. As a result, with predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$, standard Hoare logic can not only express program refinement but also support refinement proofs using standard reasoning rules. For the bitmask/set example, Our encoding transforms relational triple (1) into a standard triple:

$$\{\mathsf{Exec}_X(\mathsf{True}, \texttt{set\_union})\}\ \texttt{bit\_mask}\ \{\exists\, l.\, \mathsf{Exec}_X(\mathrm{s}^{\mathrm{H}} = l, \mathsf{skip}) \wedge \mathrm{x}^{\mathrm{L}} = \textstyle\sum_{a \in l} 2^a\}. \tag{2}$$

Fig. 5b shows part of the proof using standard Hoare logic, where the proof proceeds using standard consequence and sequencing rules. The first step of the proof weakens the precondition by performing an update of the predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$:

$$\frac{\vdash_{\exists}\ \{{P_1}^{\mathrm{H}}\}\ c_1^{\mathrm{H}}\ \{{P_2}^{\mathrm{H}}\}}{\mathsf{Exec}_X({P_1}^{\mathrm{H}}, c_1^{\mathrm{H}}; c_2^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X({P_2}^{\mathrm{H}}, c_2^{\mathrm{H}})}$$

This update reflects a single step of high-level evaluation. In this case, c_1^{H} is the assignment $\mathrm{s}^{\mathrm{H}} := \{\}$. The second step applies the standard rule Seq, and evaluates the low-level assignment $\mathrm{x}^{\mathrm{L}} := 0$. At this point, the precondition includes the updated predicate $\mathsf{Exec}_X(\mathrm{s}^{\mathrm{H}} = \emptyset, \mathrm{s}^{\mathrm{H}} := \mathrm{s}^{\mathrm{H}} \cup \{a_0\}; \ldots)$, describing the remaining behavior of the high-level program. Since this predicate does not mention or depend on the low-level state, it can be freely carried across the evaluation of the assignment. The remaining proof proceeds by alternating updates to the predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$ and standard reasoning steps on the low-level program, similar to the first two steps.

This example shows how standard proof rules can prove program refinement. Importantly, our encoding equips standard Hoare logic with comparable proof power to relational Hoare logic. The theoretical justification lies in the fact that relational proof rules can be encoded into standard Hoare rules. For instance, independently evaluating the high-level program via rules like High-Focus

corresponds to a consequence step in standard logic, via updating the predicate $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$:

$$\frac{\mathsf{Exec}_X(P^{\mathrm{H}},c_1^{\mathrm{H}}) \Rrightarrow \mathsf{Exec}_X(R^{\mathrm{H}},c_2^{\mathrm{H}}) \qquad \vdash_\forall \{\mathsf{Exec}_X(R^{\mathrm{H}},c_2^{\mathrm{H}}) \wedge F^{\mathrm{L}}\}\ c^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}{\vdash_\forall \{\mathsf{Exec}_X(P^{\mathrm{H}},c_1^{\mathrm{H}}) \wedge F^{\mathrm{L}}\}\ c^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}$$

Then applying the rule High-Focus in relational Hoare logic corresponds to applying the consequence rule in standard Hoare logic. Sec. 6 summarizes how core relational rules correspond to derivable rules in standard Hoare logic. These correspondences ensure that with our encoding, standard Hoare logic is expressive enough to prove relational triples over decomposed assertions.

Moreover, relational Hoare logic frameworks support two types of vertical composition rules: composing a refinement proof with the functional correctness proof of the high-level program, and composing two refinement proofs.

VC-FC

$$\frac{\langle \mathbb{P} \wedge [c^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle \mathbb{Q} \wedge [\mathsf{skip}] \rangle \qquad \vdash_\forall \{P^{\mathrm{H}}\}\ c^{\mathrm{H}}\ \{Q^{\mathrm{H}}\}}{\vdash_\forall \{\mathbb{P} \odot P^{\mathrm{H}}\}\ c^{\mathrm{L}}\ \{\mathbb{Q} \odot Q^{\mathrm{H}}\}}$$

VC-Refine

$$\frac{\langle \mathbb{P}_1 \wedge [c_2] \rangle\ c_1\ \langle \mathbb{Q}_1 \wedge [\mathsf{skip}] \rangle \qquad \langle \mathbb{P}_2 \wedge [c_3] \rangle\ c_2\ \langle \mathbb{Q}_2 \wedge [\mathsf{skip}] \rangle}{\langle \mathbb{P}_1 \circ \mathbb{P}_2 \wedge [c_3] \rangle\ c_1\ \langle \mathbb{Q}_1 \circ \mathbb{Q}_2 \wedge [\mathsf{skip}] \rangle}$$

Here, the operator $\odot$ denotes linking a binary assertion with a unary assertion, and the operator $\circ$ denotes linking two binary assertions. They are formally defined in Sec. 6. Under our encoding, both rules can be derived using the consequence rule and rule ExIntro in standard Hoare logic.

The central idea of our encoding theory is simple yet powerful: leveraging logical variables to embed $\forall\exists$ patterns in preconditions and postconditions. To clarify, our goal is not to develop a new program logic. Rather, by introducing a predicate $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$, we enable standard Hoare logic to provide comparable reasoning power for $\forall\exists$ relational proofs. We believe that our encoding theory will allow provers to employ the extensive body of work and tools developed for standard Hoare logic to verify relational properties.

Our main contributions are as follows:

- We introduce the first encoding theory (Theo. 4) to reduce the $\forall\exists$ relational Hoare logic to standard Hoare logic. Our encoding theory relies on a relaxed validity of relational Hoare triples based on configuration refinement (Sec. 3). Importantly, we only encode the $\forall\exists$ pattern in assertions (Sec. 4). Since a relational Hoare triple is encoded into a standard Hoare triple of the exact low-level program involved, this triple can be used in compositional reasoning with other standard Hoare logic judgments.
- We present a syntactic encoding of decomposed program-as-resource assertions into unary low-level assertions (Sec. 5). This encoding introduces the execution predicate $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$ to capture high-level behavior as a purely logical condition. The result enables direct application of our encoding for relational reasoning in standard Hoare logic.
- Our encoding fully preserves the original reasoning capabilities of relational Hoare logic. We illustrate how core inference rules in relational Hoare logic can be encoded into standard Hoare rules and present proof rules for the execution predicate $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$ (Sec. 6).
- We demonstrate the expressiveness and practicality of our approach through case studies, including merge sort, binary search trees, depth-first search, and the Knuth–Morris–Pratt (KMP) algorithm. In particular, Sec. 7 presents a detailed proof of the merging algorithm.
- Our results are formalized and machine-checked in Rocq, including all meta theorems and case studies. Our encoding theory can be extended to support function calls, undefined behavior, and separation logic. These extensions are available in the extended version [45].

2 Background: Relational Hoare Logic with Programs as Resources

The relational framework proposed by Turon et al. [42] and other works [19, 30, 31, 37, 38, 41] share a notable innovation: treating high-level programs as resources. In this section, we provide a brief overview of this approach and illustrate how to prove program refinement based on it. To focus on the core ideas of both relational and standard Hoare logic, program assertions are defined semantically as subsets of states before Sec. 5.2.

2.1 High-Level Programs as Resources

Treating high-level programs as resources involves extending a binary assertion $\mathbb{P} \subseteq \Sigma^{\mathrm{L}} \times \Sigma^{\mathrm{H}}$ with a high-level program $c^{\mathrm{H}} \in \mathrm{Prog}^{\mathrm{H}}$. Here, Σ^{L} and Σ^{H} represent the set of low-level program states and the set of high-level program states, respectively. Then the main judgment based on this approach is the relational Hoare triple $\langle \mathcal{P} \rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q} \rangle$, whose definition is restated as follows:

Definition 1 (Relational Hoare Triples). *The relational Hoare triple $\langle \mathcal{P} \rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q} \rangle$ is valid if for any $(\sigma_1^{\mathrm{L}}, \sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \models \mathcal{P}$ and any σ_2^{L} such that $(\sigma_1^{\mathrm{L}}, \sigma_2^{\mathrm{L}}) \in [\![c^{\mathrm{L}}]\!]_{\mathsf{nrm}}$, there exist σ_2^{H} and c_2^{H} such that $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \rightarrow^* (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ and $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models \mathcal{Q}$.*

Here $[\![-]\!]_{\mathsf{nrm}}$ represents the normal evaluation of denotational semantics. For any program statement $c \in \mathrm{Prog}^{\mathrm{L}}$, $(\sigma_1^{\mathrm{L}}, \sigma_2^{\mathrm{L}}) \in [\![c]\!]_{\mathsf{nrm}}$ if the execution of c from initial state σ_1^{L} may terminate at state σ_2^{L}. Besides, $\rightarrow^*$ represents multi-step transition based on small-step semantics and $(c_1^{\mathrm{H}}, \sigma_1^{\mathrm{H}}) \rightarrow^* (c_2^{\mathrm{H}}, \sigma_2^{\mathrm{H}})$ indicates that $(c_1^{\mathrm{H}}, \sigma_1^{\mathrm{H}})$ reduces to $(c_2^{\mathrm{H}}, \sigma_2^{\mathrm{H}})$ in zero or more steps. If the small-step semantics and denotational semantics both characterize the high-level programming language, it is expected that they would be consistent with each other. Specifically, the multi-step transitions $\rightarrow^*$ and the denotational semantics function $[\![-]\!]_{\mathsf{nrm}}$ both describe a large reduction:

Proposition 5. *$(\sigma_1^{\mathrm{H}}, c^{\mathrm{H}}) \rightarrow^* (\sigma_2^{\mathrm{H}}, \mathsf{skip})$ is equivalent to $(\sigma_1^{\mathrm{H}}, \sigma_2^{\mathrm{H}}) \in [\![c^{\mathrm{H}}]\!]_{\mathsf{nrm}}$.*

Then the relational Hoare triple $\langle \mathbb{P} \wedge [c^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle \mathbb{Q} \wedge [\mathsf{skip}] \rangle$ represents that the low-level program c^{L} refines the high-level program c^{H}, provided that the assertions $\mathbb{P}$ and $\mathbb{Q}$ meaningfully relate their respective initial and final states. If $\mathbb{P}$ and $\mathbb{Q}$ are both false, the triple is trivially valid but vacuous, and thus does not constitute a real refinement.

2.2 Proofs Based on Relational Hoare Triples

Fig. 6 shows core proof rules of relational Hoare logic for sequential imperative programs. The first two rules are focusing rules over decomposed assertions, used to evaluate either the low-level program with standard Hoare triples or the high-level program with angelic Hoare triples. The third rule is a loop rule for low-level programs. These rules support the following three key examples: the refinement proof for two nondeterministic programs, the refinement proof for two simple sequential programs in bitmask/set example, and the refinement proof for programs involving loops. To distinguish between program variables and logical variables, program variables are written in sans-serif font and logical variables are written in the standard *italic* font.

Nondeterministic Programs. Consider the following two programs both using the statement $\mathsf{nondet}(n, m)$ to nondeterministically select an integer within the range $[n, m]$.

$$\mathsf{x}^{\mathrm{L}} \coloneqq \mathsf{nondet}(0, 1); \qquad \Big| \qquad \mathsf{y}^{\mathrm{H}} \coloneqq \mathsf{nondet}(0, 2);$$

Example 1. *The low-level program $\mathsf{x}^{\mathrm{L}} \coloneqq \mathit{nondet}(0, 1)$ assigns either 0 or 1 to x^{L}. In contrast, the high-level program (on the right) allows y^{H} to nondeterministically take on any value of 0, 1, or 2. We aim to prove the relational Hoare triple $\langle [\mathsf{y}^{\mathrm{H}} \coloneqq \mathsf{nondet}(0,2)] \rangle\ \mathsf{x}^{\mathrm{L}} \coloneqq \mathsf{nondet}(0,1)\ \langle \mathsf{x}^{\mathrm{L}} = \mathsf{y}^{\mathrm{H}} \wedge [\mathsf{skip}] \rangle$.*

$$\textsc{Low-Focus}\ \frac{\vdash_{\forall} \{P^{\mathrm{L}}\}\ c_1^{\mathrm{L}}\ \{R^{\mathrm{L}}\} \qquad \langle \lfloor R^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}] \rangle\ c_2^{\mathrm{L}}\ \langle Q \rangle}{\langle \lfloor P^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}] \rangle\ c_1^{\mathrm{L}}; c_2^{\mathrm{L}}\ \langle Q \rangle}$$

$$\textsc{High-Focus}\ \frac{\vdash_{\exists} \{P^{\mathrm{H}}\}\ c_1^{\mathrm{H}}\ \{R^{\mathrm{H}}\} \qquad \langle \lfloor F^{\mathrm{L}} \rfloor \wedge \lceil R^{\mathrm{H}} \rceil \wedge [c_2^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle Q \rangle}{\langle \lfloor F^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c_1^{\mathrm{H}}; c_2^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle Q \rangle}$$

$$\textsc{Rel-Wh}\ \frac{\langle \mathcal{P} \wedge \lfloor b^{\mathrm{L}} \rfloor \rangle\ c^{\mathrm{L}}\ \langle \mathcal{P} \rangle}{\langle \mathcal{P} \rangle\ \text{while}\ b^{\mathrm{L}}\ \text{do}\ c^{\mathrm{L}}\ \langle \mathcal{P} \wedge \lfloor \neg b^{\mathrm{L}} \rfloor \rangle}$$

Fig. 6. Core Relational Proof Rules

The relational proof is given below with annotations in the low-level program. We first focus on the low-level program and demonically evaluate the assignment $x^{\mathrm{L}} := \mathrm{nondet}(0, 1)$, where the result value is nondeterministic 0 or 1. Then, we apply rule High-Focus to angelically assign the value of x^{L} to y^{H} in the high-level program. This example shows that, for any demonic execution of the low-level program, the rule High-Focus ensures that the high-level program captures its behavior.

```
// ⟨[y^H := nondet(0, 2)]⟩
x^L := nondet(0, 1);
// low-level step  ⟨∃n.n ∈ {0, 1} ∧ ⌊x^L = n⌋ ∧ [y^H := nondet(0, 2)]⟩
// high-level step  ⟨n ∈ {0, 1} ∧ ⌊x^L = n⌋ ∧ ⌈y^H = n⌉ ∧ [skip]⟩
```

Fig. 7. Relational Proof for the Nondeterministic Example

Simple Sequential Programs. Consider the bitmask/set example introduced in Sec. 1, where the low-level program uses bitwise OR operations to set specific bits in program variable x^{L} while the high-level program directly builds a set s^{H} using set union. We need to prove the triple $\langle [\texttt{set\_union}] \rangle\ \texttt{bit\_mask}\ \langle x^{\mathrm{L}} = \Sigma_{a \in s^{\mathrm{H}}} 2^a \wedge [\mathrm{skip}] \rangle$. Part of the relational proof for this example is shown in Fig. 5a. It begins with a high-level step, applying the rule High-Focus to focus on and evaluate the high-level assignment $s^{\mathrm{H}} = \{\ \}$. Next, the proof proceeds with a low-level step, applying the rule Low-Focus to focus on and evaluate the low-level assignment $x^{\mathrm{L}} = 0$. Then, we observe that the remaining statements in the low-level and high-level programs correspond to each other, and each corresponding pair can be evaluated by focusing rules. This example illustrates that singleton statements, such as an assignment statement, can be independently evaluated on one side through focusing rules Low-Focus and High-Focus.

Loops. As in standard Hoare logic, reasoning about while loops in relational Hoare logic requires a suitable loop invariant. Typical relational Hoare logics provide a while rule like Rel-Wh, which mirrors the standard while rule. By treating programs as resources, a loop invariant should specify the high-level program that represents the remaining iterations of the low-level program's loops.

Example 2. *When additional items need to be recorded in bitmask/set example, we often use loops. In this case, programs iterate through the items in a constant array until index* 8. *In each iteration, the two programs add the corresponding item in arrays to the collection representation (*x^{L} *or* s^{H}*). The goal is to prove the triple* $\langle [\texttt{set\_union\_loop}] \rangle\ \texttt{bit\_mask\_loop}\ \langle x^{\mathrm{L}} = \Sigma_{a \in s^{\mathrm{H}}} 2^a \wedge [\mathrm{skip}] \rangle$.

```
bit_mask_loop                    | set_union_loop
x^L = 0;                         | s^H = { };
i^L = 0;                         | j^H = 0;
while (i^L < 8){                 | while (j^H < 8){
  x^L = x^L | (1 << a[i^L]);     |   s^H = s^H ∪ { a[j^H] };
  i^L = i^L + 1; }               |   j^H = j^H + 1; }
```

Obviously, one iteration of the low-level loop corresponds to one iteration of the high-level loop. Regardless of which iteration the low-level program is on, the high-level program that represents the remaining iterations is always the while statement. We can then use the following loop invariant:

$$\exists\, l\, n.\ \lfloor \mathrm{x}^{\mathrm{L}} = \sum\nolimits_{a \in l} 2^a \wedge \mathrm{i}^{\mathrm{L}} = n \rfloor \wedge \lceil \mathrm{s}^{\mathrm{H}} = l \wedge \mathrm{j}^{\mathrm{H}} = n \rceil \wedge [\text{while } \mathrm{j}^{\mathrm{H}} < 8 \text{ do } \ldots] \tag{3}$$

In the previous examples, the low-level programs are structurally aligned with the high-level programs, enabling smooth verification. When structural alignment is absent, the programs-as-resources approach is still effective in handling such cases. For instance, the following two variants of the low-level program in Example 2 also refine the high-level program of this example. Here one variant extracts the first step from the loop, and the other executes two steps per iteration. Complete proofs for these examples can be found in the extended version [45].

Example 3. *The left low-level program performs initialization steps before executing the loop, and the right low-level program executes two assignments in each loop iteration.*

```
x^L = 0 | (1 << a[0]);
i^L = 1;
while (i^L < 8){
  x^L = x^L | (1 << a[i^L]);
  i^L = i^L + 1; }
```

```
x^L = 0; i^L = 0;
while (i^L < 8){
  x^L = x^L | (1 << a[i^L]);
  x^L = x^L | (1 << a[i^L + 1]);
  i^L = i^L + 2; }
```

Building on the idea of treating high-level programs as resources, relational Hoare logic uses relational Hoare triples as its primary judgments. Then the relational verification processes feature two key components: one is the independent evaluation of singleton statements through focusing rules Low-Focus and High-Focus; the other involves applying rules similar to those in standard Hoare logic for "while" and "if" statements. In this paper, we present an encoding theory that allows relational verification incorporating all these features to be conducted in standard Hoare logic.

3 Reformulating Validity with Configuration Refinement

In this paper, we assume that the programming languages are equipped with denotational semantics.

Definition 6 (Denotational Semantics). *For program $c \in Prog$, $[\![c]\!]_{\mathsf{nrm}}$ denotes its denotational semantics, $(\sigma, \sigma') \in [\![c]\!]_{\mathsf{nrm}}$ if executing c from initial state σ can terminate at state σ'.*

Recall from Def. 1, the multi-step transition $\rightarrow^*$ based on small-step semantics is used to capture the update of the high-level configuration. In this paper, we propose to relax that update with a configuration refinement relation $\hookrightarrow$, which is defined as follows:

Definition 7 (Configuration Refinement). *For any programs c_1 c_2 and any states σ_1 $\sigma_2 \in \Sigma$, configuration (σ_2, c_2) refines (σ_1, c_1), denoted as $(\sigma_1, c_1) \hookrightarrow (\sigma_2, c_2)$, if for any state σ_3, $(\sigma_2, \sigma_3) \in [\![c_2]\!]_{\mathsf{nrm}}$ implies $(\sigma_1, \sigma_3) \in [\![c_1]\!]_{\mathsf{nrm}}$.*

Here, the configuration refinement is defined based on denotational semantics. This choice ensures we maintain a consistent semantics throughout the paper. Nevertheless, if both small-step and denotational semantics characterize the high-level language, then by Prop. 5 one can also give an equivalent small-step definition of configuration refinement. Concretely, the implication that $(\sigma_2^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_2^{\mathrm{H}}]\!]_{\mathsf{nrm}}$ implies $(\sigma_1^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_1^{\mathrm{H}}]\!]_{\mathsf{nrm}}$ can be replaced by that $(\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \rightarrow^* (\sigma_3^{\mathrm{H}}, \mathsf{skip})$ implies $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \rightarrow^* (\sigma_3^{\mathrm{H}}, \mathsf{skip})$. Compared to the multi-step transition $\rightarrow^*$, configuration refinement $\hookrightarrow$ is more relaxed, as evidenced by the following proposition.

Proposition 8. *If $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \rightarrow^* (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$, then $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$.*

Notably, the reverse implication does not always hold. For instance, $(\mathsf{x} = 0, \mathsf{skip}) \hookrightarrow (\mathsf{x} = 1, \mathsf{x} := \mathsf{x} - 1)$ holds, whereas there is no multi-step transition from the left configuration to the right one.

Based on configuration refinement, we adopt an alternative definition for relational Hoare triples.

Definition 9 (Alternative Definition of Relational Hoare Triples). *The relational Hoare triple $\langle \mathcal{P} \rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q} \rangle$ is valid if for any $(\sigma_1^{\mathrm{L}}, \sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \models \mathcal{P}$ and any σ_2^{L} such that $(\sigma_1^{\mathrm{L}}, \sigma_2^{\mathrm{L}}) \in [\![c^{\mathrm{L}}]\!]_{\mathsf{nrm}}$, there exist σ_2^{H} and c_2^{H} such that $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ and $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models \mathcal{Q}$.*

Using this definition, the triple $\langle \mathbb{P} \wedge [c^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle \mathbb{Q} \wedge [\mathsf{skip}] \rangle$ still represents program refinement shown in Fig. 2 since the following proposition holds.

Proposition 10. $(\sigma_1^{\mathrm{H}}, c^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, \mathsf{skip})$ *iff.* $(\sigma_1^{\mathrm{H}}, \sigma_2^{\mathrm{H}}) \in [\![c^{\mathrm{H}}]\!]_{\mathsf{nrm}}$

Intuitively, when the high-level program in the postcondition is not skip, it can be interpreted as a continuation, which represents the remaining abstract behavior that must be completed by the low-level context. This allows triples to capture not just full refinements, but intermediate steps in the control flow. Besides, useful proof rules (such as rules in Sec. 2.2) remain sound. A small benefit of employing this new definition is allowing us to adhere to denotational semantics throughout the paper, thus avoiding the complexity of translating between small-step and denotational semantics when presenting theories. This coherence also benefits our Rocq formalization to rely on one semantics as well. The far greater significance of this definition, however, lies in its decomposition property. That is, we can equivalently break down $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ into implications on *weakest (liberal) preconditions* [15] for an arbitrary set of final states. Formally:

Theorem 11 (Decomposition). *The following two propositions are equivalent:*

(a) $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$

(b) $\forall (X \subseteq \Sigma^{\mathrm{H}}).\ \sigma_1^{\mathrm{H}} \models \mathit{wlp}(c_1^{\mathrm{H}}, X) \Rightarrow \sigma_2^{\mathrm{H}} \models \mathit{wlp}(c_2^{\mathrm{H}}, X)$

Definition 12 (Weakest Precondition). *For any program c and any set $X \subseteq \Sigma$, which also serves as a postcondition over program states, $\mathit{wlp}(c, X)$ gives the weakest precondition such that all terminal states resulting from the execution of program c satisfy X and is defined as:*

$$\sigma \models \mathit{wlp}(c, X) \quad \textit{iff.} \quad \forall \sigma_0.\ (\sigma, \sigma_0) \in [\![c]\!]_{\mathsf{nrm}} \Rightarrow \sigma_0 \models X$$

The decomposition theorem (Theo. 11) plays a central role in our encoding theory, which we will explain later in Sec. 4. Before proving the theorem, we recall a basic but useful set-theoretic equivalence that will be used in the proof:

Proposition 13 (A poset fact). *for any set A B, $B \subseteq A$ iff. for any set C, if $A \subseteq C$ then $B \subseteq C$.*

The left-to-right direction of this fact is obvious, while the right-to-left direction can be proved by instantiating C as A. We now prove Theo. 11, where Fig. 8 illustrates the intuition. According to the definition of configuration refinement, *(a)* states that any terminal state σ^{H} reachable from high-level programs configuration $(\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ must also be reachable from the high-level program configuration $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}})$. This is equivalent to the set inclusion $T' \subseteq T$. Here T' and T represent the terminal state sets for $(\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ and $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}})$ respectively:

$$T \triangleq \{\sigma_3^{\mathrm{H}} \mid (\sigma_1^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_1^{\mathrm{H}}]\!]_{\mathsf{nrm}}\} \qquad T' \triangleq \{\sigma_3^{\mathrm{H}} \mid (\sigma_2^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_2^{\mathrm{H}}]\!]_{\mathsf{nrm}}\}$$

According to Prop. 13, the set inclusion $T' \subseteq T$ implies that for any $X \subseteq \Sigma^{\mathrm{H}}$, if any terminal state in T satisfies X, then every terminal state in T' also satisfies X, which corresponds to *(b)*. To derive *(b)* $\Rightarrow$ *(a)*, we employ a similar approach to the right-to-left proof of Prop. 13 as follows:

Proposition 14. *let X be $\{\sigma_3^{\mathrm{H}} \mid (\sigma_1^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_1^{\mathrm{H}}]\!]_{\mathsf{nrm}}\}$, then proposition (b) implies proposition (a).*

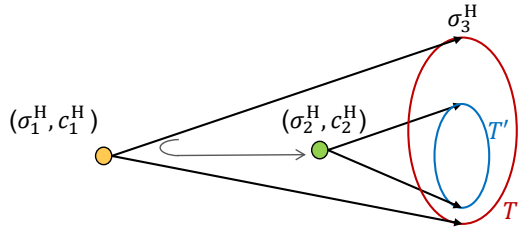

Fig. 8. State Transition Relation

4 Encode Relational Hoare Triples

This section introduces our encoding theory. We first explain the goal and give an intuitive overview via a naive attempt (Sec. 4.1), then formally present the theory and prove its correctness (Sec. 4.2).

4.1 Intuition behind the Encoding Theory: A Naive Attempt

At the core of the encoding theory is to define the assertion encoding $(\!|-|\!)$, which transforms a program-as-resource assertion $\mathcal{P} \subseteq \Sigma^{\mathrm{L}} \times \Sigma^{\mathrm{H}} \times \mathrm{Prog}^{\mathrm{H}}$ into a unary assertion $(\!|\mathcal{P}|\!) \subseteq \Sigma^{\mathrm{L}}$ and ensures:

$$\langle \mathcal{P} \rangle\ c^{\mathrm{L}}\ \langle Q \rangle \text{ is valid} \quad \text{iff.} \quad \vdash_{\forall} \{(\!|\mathcal{P}|\!)\}\ c^{\mathrm{L}}\ \{(\!|Q|\!)\}.$$

To define the encoding, review Def. 9 which states that given any initial states $(\sigma_1^{\mathrm{L}}, \sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \models \mathcal{P}$:

- for any low-level state σ_2^{L} such that $(\sigma_1^{\mathrm{L}}, \sigma_2^{\mathrm{L}}) \in [\![c^{\mathrm{L}}]\!]_{\mathsf{nrm}}$, **there exist high-level state σ_2^{H} and program c_2^{H} such that $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ and $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models Q$.**

In standard Hoare logic, the assertions only concern low-level states σ_1^{L} and σ_2^{L}. If we could remove the angelic part (everything after the exist quantifier) in bold font away from this validity definition, and embed them into the postcondition, then the encoded postcondition will become an assertion over low-level states. Under this transformation, the validity definition corresponds exactly to that of standard Hoare logic. This leads to a naive encoding for the postcondition:

$$(\!|Q|\!)(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \triangleq \lambda \sigma^{\mathrm{L}}.\, \exists \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}.\, (\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \wedge (\sigma^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models Q.$$

Therefore, we have

$$\langle \mathcal{P} \rangle\ c\ \langle Q \rangle \text{ is valid} \quad \text{iff.} \quad \forall \sigma^{\mathrm{H}}\, c^{\mathrm{H}}.\, \{\lambda \sigma^{\mathrm{L}}.(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \in \mathcal{P}\}\ c\ \{(\!|Q|\!)(\sigma^{\mathrm{H}}, c^{\mathrm{H}})\} \text{ is valid}$$

Here, the encoded postcondition $(\!|Q|\!)(\sigma^{\mathrm{H}}, c^{\mathrm{H}})$ is parameterized by σ^{H} and c^{H}, ensuring that the resulting high-level configuration is obtained through a valid configuration refinement from the initial configuration. However, σ^{H} and c^{H} depend on the precondition $\mathcal{P}$. This dependency prevents the naive method from providing a uniform encoding for both preconditions and postconditions. Without a uniform encoding, we cannot derive rules from standard Hoare logic to relational Hoare logic. For example, consider the rule Rel-Wh for addressing while loops in low-level programs. Using the naive encoding, the invariant $\mathcal{P}$ in the precondition and postcondition will be encoded differently. Then this rule can not be derived from the while rule for standard Hoare logic.

Interestingly, although the naive encoding does not work, it suggests applying existential quantification to abstract the high-level state and the high-level program from program-as-resource assertions. Existential quantification provides a straightforward means to encode the ∃ component of the ∀∃ pattern in relational Hoare triples into the ∀ structure of standard Hoare triples. For the ∀ component, existential quantification in the precondition exactly corresponds to some universal property, as rule ExIntro shows. This discussion suggests that program-as-resource precondition $\mathcal{P}$ and postcondition Q can be encoded into:

- $\lambda\sigma_1^{\mathrm{L}}.\, \exists\, \sigma_1^{\mathrm{H}} c_1^{\mathrm{H}}.\, (\sigma_1^{\mathrm{L}}, \sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \models \mathcal{P} \wedge$ some condition
- $\lambda\sigma_2^{\mathrm{L}}.\, \exists\, \sigma_2^{\mathrm{H}} c_2^{\mathrm{H}}.\, (\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models \mathcal{Q} \wedge$ some condition

Here "some condition" is independent of low-level states σ_1^{L} and σ_2^{L}. However, using this encoding form alone leads to an encoded postcondition that becomes disconnected from the encoded precondition. As a result, the encoded relational Hoare triples lack key information to ensure configuration refinement $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$. To address this, we need to introduce a necessary placeholder X. This placeholder serves as the connection between the encoded precondition and the encoded postcondition. Given a placeholder X, a program-as-resource assertion $\mathcal{P}$ can be encoded into:

$$\lambda\sigma^{\mathrm{L}}.\, \exists \sigma^{\mathrm{H}} c^{\mathrm{H}}.\, (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \models \mathcal{P} \wedge \text{some condition}(\sigma^{\mathrm{H}}, c^{\mathrm{H}}, X)$$

4.2 The Encoding Theory

Recall from Theo. 11, it suggests that configuration refinement $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ is equivalent to the requirement that for every subset X, if σ_1^{H} satisfies the weakest precondition of c_1^{H} for X, then σ_2^{H} satisfies the weakest precondition of c_2^{H} for X. Therefore, we can employ the weakest precondition to formalize "some condition" and define the assertion encoding.

Definition 15 (Assertion Encoding). *For any program-as-resource assertion $\mathcal{P} \subseteq \Sigma^{\mathrm{L}} \times \Sigma^{\mathrm{H}} \times \mathrm{Prog}^{\mathrm{H}}$ and subset of high-level states $X \subseteq \Sigma^{\mathrm{H}}$, define the encoding as*

$$\sigma^{\mathrm{L}} \models (\!|\mathcal{P}|\!)_X \quad \textit{iff.} \quad \exists \sigma^{\mathrm{H}} c^{\mathrm{H}}.\, (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}) \models \mathcal{P} \wedge \sigma^{\mathrm{H}} \models \mathit{wlp}(c^{\mathrm{H}}, X).$$

Subsequently, we can encode relational Hoare triples and prove that this encoding is correct.

Theorem 4 (Encoding Relational Triples). *For any low-level statement c^{L}, and assertions $\mathcal{P}$, $\mathcal{Q} \subseteq \Sigma^{\mathrm{L}} \times \Sigma^{\mathrm{H}} \times \mathrm{Prog}^{\mathrm{H}}$:*

$$\underbrace{\langle \mathcal{P} \rangle\ c^{\mathrm{L}}\ \langle \mathcal{Q} \rangle}_{\mathcal{J}} \ \textit{is valid} \quad \textit{iff.} \quad \underbrace{\forall X.\ \vdash_{\forall} \{(\!|\mathcal{P}|\!)_X\}\ c^{\mathrm{L}}\ \{(\!|\mathcal{Q}|\!)_X\}}_{J}$$

Proof. we prove two directions:

- $\Rightarrow$: For any X and σ_1^{L} in $(\!|\mathcal{P}|\!)_X$, there exist σ_1^{H} and c_1^{H} such that $\sigma_1^{\mathrm{H}} \models \mathsf{wlp}(c_1^{\mathrm{H}}, X)$ and $(\sigma_1^{\mathrm{L}}, \sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \models \mathcal{P}$. Then for any state σ_2^{L} such that $(\sigma_1^{\mathrm{L}}, \sigma_2^{\mathrm{L}}) \in [\![c^{\mathrm{L}}]\!]_{\mathsf{nrm}}$, according to relational triple $\mathcal{J}$, there exist σ_2^{H} and c_2^{H} such that $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ and $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models \mathcal{Q}$. Then by Theo. 11, we have $\sigma_2^{\mathrm{H}} \models \mathsf{wlp}(c_2^{\mathrm{H}}, X)$. Therefore, according to Def. 15 we derive $\sigma_2^{\mathrm{L}} \models (\!|\mathcal{Q}|\!)_X$.
- $\Leftarrow$: Based on Prop. 14, for any $(\sigma_1^{\mathrm{L}}, \sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \models \mathcal{P}$, we instantiate X as the high-level terminal states set $\{\sigma_3^{\mathrm{H}} \mid (\sigma_1^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_1^{\mathrm{H}}]\!]_{\mathsf{nrm}}\}$. By the definition of assertion encoding, $\sigma_1^{\mathrm{L}} \models (\!|\mathcal{P}|\!)_X$. According to standard triple J, for any σ_2^{L} such that $(\sigma_1^{\mathrm{L}}, \sigma_2^{\mathrm{L}}) \in [\![c^{\mathrm{L}}]\!]_{\mathsf{nrm}}$, we have $\sigma_2^{\mathrm{L}} \models (\!|\mathcal{Q}|\!)_X$. By the definition of assertion encoding, there exist σ_2^{H} and c_2^{H} such that $\sigma_2^{\mathrm{H}} \models \mathsf{wlp}(c_2^{\mathrm{H}}, X)$ and $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models \mathcal{Q}$. According to the definition of weakest preconditions, for any σ_3^{H} such that $(\sigma_2^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_2^{\mathrm{H}}]\!]_{\mathsf{nrm}}$, $\sigma_3^{\mathrm{H}} \models X$. That is for any σ_3^{H} such that $(\sigma_2^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_2^{\mathrm{H}}]\!]_{\mathsf{nrm}}$, we have $(\sigma_1^{\mathrm{H}}, \sigma_3^{\mathrm{H}}) \in [\![c_2^{\mathrm{H}}]\!]_{\mathsf{nrm}}$. Therefore, we finally derive $(\sigma_1^{\mathrm{H}}, c_1^{\mathrm{H}}) \hookrightarrow (\sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}})$ and $(\sigma_2^{\mathrm{L}}, \sigma_2^{\mathrm{H}}, c_2^{\mathrm{H}}) \models \mathcal{Q}$.

□

5 Syntactic Encoding of Decomposed Assertions

Up to this point, we have treated assertions semantically as subsets of program states and configurations, and presented the assertion encoding $(\!|-|\!)_X$ semantically. This semantic formulation has helped highlight the foundational ideas behind our encoding theory and its soundness guarantees. In this section, to make our encoding theory more explicit and to apply our encoding theory within

an actual verification tool, we present a syntactic encoding that translates decomposed program-as-resource assertions into unary assertions over low-level states. We begin by establishing equivalent transformations that describe how decomposed assertions behave under the semantic encoding $(\!|-|\!)_X$ (Sec. 5.1). These equivalences justify the structure of our syntactic encoding. We then define the syntax for decomposed assertions and present the syntactic encoding (Sec. 5.2).

Using existential logical variables and pure logical conditions, a program-as-resource assertion can typically be written into components concerning low-level states, high-level states, and the high-level program. This decomposition allows for concise focusing rules in real verification tasks.

$$\exists \vec{a}.\, B(\vec{a}) \wedge \lfloor P^{\mathrm{L}}(\vec{a}) \rfloor \wedge \lceil P^{\mathrm{H}}(\vec{a}) \rceil \wedge [c^{\mathrm{H}}]$$

5.1 Equivalent Transformations for Encoded Decomposed Assertions

Based on assertion encoding $(\!|-|\!)_X$ (Def. 15), when encoding a decomposed assertion, the existential variables $\vec{a}$, pure logical conditions $B(\vec{a})$, and unary assertion $P^{\mathrm{L}}(\vec{a})$ about the low-level states can be lifted out. This is captured by the following assertion transformation theorem:

Theorem 16 (Encoded Assertion Transformation).

(a) $(\!|\exists a. \mathcal{P}(a)|\!)_X \iff \exists a.\, (\!|\mathcal{P}(a)|\!)_X$.
(b) $(\!|B \wedge \mathcal{P}|\!)_X \iff B \wedge (\!|\mathcal{P}|\!)_X$.
(c) $(\!|\lfloor P^{\mathrm{L}} \rfloor \wedge \mathcal{P}|\!)_X \iff P^{\mathrm{L}} \wedge (\!|\mathcal{P}|\!)_X$.
(d) $(\!|\mathcal{P}_1 \vee \mathcal{P}_2|\!)_X \iff (\!|\mathcal{P}_1|\!)_X \vee (\!|\mathcal{P}_2|\!)_X$.

Here we include a transformation *(d)* for disjunction to account for the fact that program-as-resource assertions may include disjunctions of decomposed forms. These equivalences show that the encoding of a decomposed assertion can be rewritten as follows:

$$(\!|\exists \vec{a}.\, B(\vec{a}) \wedge \lfloor P^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}]|\!)_X \iff \exists \vec{a}.\, B(\vec{a}) \wedge P^{\mathrm{L}}(\vec{a}) \wedge (\!|\lceil P^{\mathrm{H}}(\vec{a}) \rceil \wedge [c^{\mathrm{H}}]|\!)_X.$$

The final component, $(\!|\lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}]|\!)_X$, captures the execution behavior of the high-level program. Crucially, it is pure: it does not mention or depend on any low-level program state. In the following, we use this insight to define a syntactic encoding based on a pure logical predicate, the *execution predicate* $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$, which represents this component in the low-level assertion language.

5.2 Syntactic Encoding for Decomposed Assertions

While the previous subsection defined the encoding semantically, we now make it more concrete by introducing a class of syntactic decomposed assertions and a corresponding syntactic encoding. We begin with the syntax of basic assertions. A pure logical predicate B is a first-order formula over logical variables, built from user-defined predicates p (such as equality or reachability) using standard connectives and quantifiers. Our encoding is parametric over the assertion languages, so we do not fix the syntax of low-level assertions P^{L} or high-level assertions P^{H}. We assume both include pure predicates B, user-defined predicates p over low-level (e^{L}) or high-level expressions (e^{H}), conjunction ($\wedge$), and the $\exists$ quantifier to introduce logical variables. These languages can be extended as needed, e.g., with universal quantification ($\forall$) or separation logic constructs such as the empty heap predicate (emp) and separating conjunction ($*$).

$$\begin{array}{rcl} B & ::= & \mathsf{True} \mid \mathsf{False} \mid p(\vec{a}) \mid FO(B) \\ P^{\mathrm{L}} & ::= & B \mid p(\vec{e}^{L}) \mid P^{\mathrm{L}} \wedge P^{\mathrm{L}} \mid \exists a.\, P^{\mathrm{L}}(a) \mid \ldots \\ P^{\mathrm{H}} & ::= & B \mid p(\vec{e}^{H}) \mid P^{\mathrm{H}} \wedge P^{\mathrm{H}} \mid \exists a.\, P^{\mathrm{H}}(a) \mid \ldots \end{array}$$

Building on the syntax of basic assertions, we define the syntax of program-as-resource assertions as finite disjunctions of decomposed forms, in which $[c^{\mathrm{H}}]$ specifies the remaining high-level

program to be fulfilled, and $\lfloor P^{\mathrm{L}} \rfloor$ and $\lceil P^{\mathrm{H}} \rceil$ specify properties over the low-level and high-level states, respectively.

$$\mathcal{P} \quad ::= \quad \bigvee_i (\exists \vec{a}.\, B_i(\vec{a}) \wedge \lfloor P_i^{\mathrm{L}}(\vec{a}) \rfloor \wedge \lceil P_i^{\mathrm{H}}(\vec{a}) \rceil \wedge [c_i^{\mathrm{H}}])$$

Based on the syntax of program-as-resource assertions and the semantic equivalences established in Theo. 16, we now define how such assertions are transformed into unary assertions over low-level states. To express the high-level behavior within a unary assertion, we introduce the *execution predicate* $\mathsf{Exec}_X(P^{\mathrm{H}}(\vec{a}),c^{\mathrm{H}})$ to represent $(\!|\lceil P^{\mathrm{H}}(\vec{a}) \rceil \wedge [c^{\mathrm{H}}]|\!)_X$.

Definition 17 (Syntactic Encoding). *Given a high-level postcondition X, the encoding of a syntactic decomposed assertion into a unary assertion over low-level states is defined as follows:*

$$\begin{aligned} \mathsf{Enc}_X(\exists \vec{a}.\, B(\vec{a}) \wedge \lfloor P^{\mathrm{L}}(\vec{a}) \rfloor \wedge \lceil P^{\mathrm{H}}(\vec{a}) \rceil \wedge [c^{\mathrm{H}}]) &\triangleq \exists \vec{a}.\, B(\vec{a}) \wedge P^{\mathrm{L}}(\vec{a}) \wedge \mathsf{Exec}_X(P^{\mathrm{H}}(\vec{a}),c^{\mathrm{H}}) \\ \mathsf{Enc}_X(\mathcal{P}_1 \vee \mathcal{P}_2) &\triangleq \mathsf{Enc}_X(\mathcal{P}_1) \vee \mathsf{Enc}_X(\mathcal{P}_2) \end{aligned}$$

The semantic interpretation of this syntactic encoding aligns with the equivalences in Theo. 16. That is, for any decomposed program-as-resource assertion $\mathcal{P}$ and high-level postcondition X:

$$\mathsf{Enc}_X(\mathcal{P}) \Leftrightarrow (\!|\mathcal{P}|\!)_X.$$

By transforming program-as-resource assertions into unary assertions over low-level states, this encoding enables relational reasoning to be conducted using standard Hoare logic. For instance, in the bitmask/set example, the relational triple (4) can be written in decomposed form as (6), which then is encoded into a standard triple (6) for any high-level postcondition X.

$$\langle [\texttt{set\_union}] \rangle\ \texttt{bit\_mask}\ \langle \mathrm{x}^{\mathrm{L}} = \sum_{a \in \mathrm{s}^{\mathrm{H}}} 2^a \wedge [\texttt{skip}] \rangle \tag{4}$$

$$\langle [\texttt{set\_union}] \rangle\ \texttt{bit\_mask}\ \langle \exists l.\, \lfloor \mathrm{x}^{\mathrm{L}} = \sum_{a \in l} 2^a \rfloor \wedge \lceil \mathrm{s}^{\mathrm{H}} = l \rceil \wedge [\texttt{skip}] \rangle \tag{5}$$

$$\{\mathsf{Exec}_X(\mathsf{True},\texttt{set\_union})\}\ \texttt{bit\_mask}\ \{\exists l.\, \mathsf{Exec}_X(\mathrm{s}^{\mathrm{H}} = l,\ [\texttt{skip}]) \wedge \mathrm{x}^{\mathrm{L}} = \sum_{a \in l} 2^a\} \tag{6}$$

In the remainder of the paper, we adopt the notation $(\!|\mathcal{P}|\!)_X$ uniformly, as it is interchangeable with $\mathsf{Enc}_X(\mathcal{P})$ when $\mathcal{P}$ is a decomposition assertion, which is the typical case in practical verification.

Note that the execution predicate is a pure logical predicate, and we can extend the low-level assertion language to include it:

$$\bar{P}^L ::= B \mid \mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}}) \mid p(\vec{e}^L) \mid \bar{P}^L \wedge \bar{P}^L \mid \exists a.\, \bar{P}^L(a) \mid \bar{P}^L \vee \bar{P}^L \mid \ldots$$

$\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$ expresses that there exists an initial state satisfying P^{H} such that any final state after executing c^{H} satisfies X. Its semantic interpretation can be defined as follows:

Definition 18 (Semantic Interpretation of the Execution Predicate). *For any high-level assertion P^{H}, postcondition X, and program c^{H},*

$$\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})\ \textit{holds} \quad \textit{iff.} \quad \textit{there exists}\ \sigma^{\mathrm{H}}\ \textit{such that}\ \sigma^{\mathrm{H}} \models P^{\mathrm{H}} \wedge \mathit{wlp}(c^{\mathrm{H}}, X)$$

In the next section, we will present proof rules for reasoning about $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$.

6 Encoded Proof Rules and Reasoning with the Execution Predicate

In this section, we show how relational reasoning can be carried out within standard Hoare logic, building on our encoding. We first encode core relational proof rules as derivable standard rules and introduce proof rules for the execution predicate $\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}})$ (Sec. 6.1). Then we demonstrate examples to show how to leverage our encoding to conduct relational proofs in standard Hoare logic (Sec. 6.2). Together, these results show that standard Hoare logic, extended with the execution predicate and related proof rules, suffices for $\forall\exists$ relational proofs. Finally, we show that vertical composition rules can also be encoded to support compositional reasoning (Sec. 6.3).

$$\textsc{Low-Focus}\quad \frac{\vdash_\forall \{P^{\mathrm{L}}\}\ c_1^{\mathrm{L}}\ \{R^{\mathrm{L}}\} \qquad \langle \lfloor R^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}] \rangle\ c_2^{\mathrm{L}}\ \langle Q \rangle}{\langle \lfloor P^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c^{\mathrm{H}}] \rangle\ c_1^{\mathrm{L}}; c_2^{\mathrm{L}}\ \langle Q \rangle} \qquad \frac{\vdash_\forall \{P^{\mathrm{L}}\}\ c_1^{\mathrm{L}}\ \{R^{\mathrm{L}}\} \qquad \vdash_\forall \{\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}}) \wedge R^{\mathrm{L}}\}\ c_2^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}{\vdash_\forall \{\mathsf{Exec}_X(P^{\mathrm{H}},c^{\mathrm{H}}) \wedge P^{\mathrm{L}}\}\ c_1^{\mathrm{L}}; c_2^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}$$

$$\textsc{Choice}\quad \frac{\langle \mathcal{P} \rangle\ c_1^{\mathrm{L}}\ \langle Q \rangle \qquad \langle \mathcal{P} \rangle\ c_2^{\mathrm{L}}\ \langle Q \rangle}{\langle \mathcal{P} \rangle\ \mathsf{choice}(c_1^{\mathrm{L}}, c_2^{\mathrm{L}})\ \langle \mathcal{P} \rangle} \qquad \frac{\vdash_\forall \{(\!|\mathcal{P}|\!)_X\}\ c_1^{\mathrm{L}}\ \{(\!|Q|\!)_X\} \qquad \vdash_\forall \{(\!|\mathcal{P}|\!)_X\}\ c_2^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}{\vdash_\forall \{(\!|\mathcal{P}|\!)_X\}\ \mathsf{choice}(c_1^{\mathrm{L}}, c_2^{\mathrm{L}})\ \{(\!|Q|\!)_X\}}$$

$$\textsc{Rel-Wh}\quad \frac{\langle \mathcal{P} \wedge \lfloor b^{\mathrm{L}} \rfloor \rangle\ c^{\mathrm{L}}\ \langle \mathcal{P} \rangle}{\langle \mathcal{P} \rangle\ \mathsf{while}\ b^{\mathrm{L}}\ \mathsf{do}\ c^{\mathrm{L}}\ \langle \mathcal{P} \wedge \lfloor \neg b^{\mathrm{L}} \rfloor \rangle} \qquad \frac{\vdash_\forall \{(\!|\mathcal{P}|\!)_X \wedge b^{\mathrm{L}}\}\ c^{\mathrm{L}}\ \{(\!|\mathcal{P}|\!)_X\}}{\vdash_\forall \{(\!|\mathcal{P}|\!)_X\}\ \mathsf{while}\ b^{\mathrm{L}}\ \mathsf{do}\ c^{\mathrm{L}}\ \{(\!|\mathcal{P}|\!)_X \wedge \neg b^{\mathrm{L}}\}}$$

Fig. 9. Low-Level Dependent Relational Rules and Their Corresponding Encoded Rules

To support the rules presented in this section, we assume both the low-level and high-level languages are standard imperative languages. Their syntax includes the following constructs:

$$c \quad ::= \quad \mathsf{skip} \mid \mathsf{x} := e \mid \mathsf{x} := \mathsf{nondet}(e, e) \mid \mathsf{assume}\ b \mid \mathsf{choice}(c,\ c) \mid \mathsf{while}\ b\ \mathsf{do}\ c \mid c; c$$

In programs, expressions e refer only to program variables and constants. The non-deterministic assignment $\mathsf{x} := \mathsf{nondet}(e_1, e_2)$ assigns to x an arbitrary value between e_1 and e_2. We include statement assume b, which behaves like skip when b holds and blocks otherwise, and a nondeterministic choice construct $\mathsf{choice}(c_1,\ c_2)$, which executes either c_1 or c_2 nondeterministically. Then standard if b then c_1 else c_2 can be written as $\mathsf{choice}(\mathsf{assume}\ b; c_1,\ \mathsf{assume}\ \neg b; c_2)$.

6.1 Proof Rules

Low-Level Rules. Relational Hoare logic provides proof rules depending on low-level statements, such as sequencing (Low-Focus), loops (Rel-Wh), and nondeterministic choice (Choice). These rules are transformed into standard Hoare rules using our encoding theory, as shown in Fig. 9. For example, the rule Low-Focus is encoded into a special case of sequencing rule Seq. Theoretically, our encoding theory explains the similarities in the proof rules of relational Hoare logic and standard Hoare logic. Practically, our encoding theory also suggests proof rules in standard Hoare logic can be used to prove program refinement in a similar way to relational Hoare logic.

High-Level Rules. Relational Hoare logic also provides proof rules for independently evaluating high-level programs. These high-level steps can be encoded into the consequence rule Conseq-Pre in standard Hoare logic by updating the execution predicate as follows:

$$\frac{\mathsf{Exec}_X(P^{\mathrm{H}},c_1^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(R^{\mathrm{H}},c_2^{\mathrm{H}}) \qquad \vdash_\forall \{\mathsf{Exec}_X(R^{\mathrm{H}},c_2^{\mathrm{H}}) \wedge F^{\mathrm{L}}\}\ c^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}{\vdash_\forall \{\mathsf{Exec}_X(P^{\mathrm{H}},c_1^{\mathrm{H}}) \wedge F^{\mathrm{L}}\}\ c^{\mathrm{L}}\ \{(\!|Q|\!)_X\}}$$

As shown in Fig. 10, rule High-Focus allows one to angelically execute part of the high-level program in isolation. Our encoding reflects this rule by allowing a successful angelic evaluation of c_1^{H} to update $\mathsf{Exec}_X(P_1{}^{\mathrm{H}},c_1^{\mathrm{H}};c_2^{\mathrm{H}})$ into $\mathsf{Exec}_X(P_2{}^{\mathrm{H}},c_2^{\mathrm{H}})$. The rest of Fig. 10 presents proof rules for updating the execution predicate, covering the basic imperative constructs. For example, angelic nondeterminism is captured by two rules: rule Exec-Nondet allows any value v within the admissible range to be picked, while rule Exec-ChoiceL permits the left branch of a nondeterministic choice to be selected (a symmetric right-branch rule is omitted here). Sequential composition is handled by two rules: Exec-Pure allows rewriting the first subcommand without modifying state; Exec-Seq ensures that sequential composition can be evaluated step by step. For loops, the pair of

Relational high-level rule and its corresponding encoded rule

$$\textsc{High-Focus}\ \frac{\vdash_\exists \{P^{\mathrm{H}}\}\ c_1^{\mathrm{H}}\ \{R^{\mathrm{H}}\} \qquad \langle \lfloor F^{\mathrm{L}} \rfloor \wedge \lceil R^{\mathrm{H}} \rceil \wedge [c_2^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle Q \rangle}{\langle \lfloor F^{\mathrm{L}} \rfloor \wedge \lceil P^{\mathrm{H}} \rceil \wedge [c_1^{\mathrm{H}}; c_2^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle Q \rangle} \qquad \frac{\vdash_\exists \{{P_1}^{\mathrm{H}}\}\ c_1^{\mathrm{H}}\ \{{P_2}^{\mathrm{H}}\}}{\mathsf{Exec}_X({P_1}^{\mathrm{H}}, c_1^{\mathrm{H}}; c_2^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X({P_2}^{\mathrm{H}}, c_2^{\mathrm{H}})}$$

Proof rules for updating the execution predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$

$$\textsc{Exec-Assign}\ \frac{}{\mathsf{Exec}_X(P^{\mathrm{H}}[e^{\mathrm{H}}/\mathrm{x}^{\mathrm{H}}], \mathrm{x}^{\mathrm{H}} := e^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{skip})} \qquad \textsc{Exec-ChoiceL}\ \frac{}{\mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{choice}(c_1^{\mathrm{H}}, c_2^{\mathrm{H}})) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, c_1^{\mathrm{H}})}$$

$$\textsc{Exec-Nondet}\ \frac{P^{\mathrm{H}}[v/\mathrm{x}^{\mathrm{H}}] \Rightarrow e_1^{\mathrm{H}} \le v \le e_2^{\mathrm{H}}}{\mathsf{Exec}_X(P^{\mathrm{H}}[v/\mathrm{x}^{\mathrm{H}}], \mathrm{x}^{\mathrm{H}} := \mathsf{nondet}(e_1^{\mathrm{H}}, e_2^{\mathrm{H}})) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{skip})} \qquad \textsc{Exec-Assume}\ \frac{P^{\mathrm{H}} \Rightarrow b^{\mathrm{H}}}{\mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{assume}\ b^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{skip})}$$

$$\textsc{Exec-Pure}\ \frac{\forall X.\ \mathsf{Exec}_X(P^{\mathrm{H}}, c_1^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, c_0^{\mathrm{H}})}{\forall X.\ \mathsf{Exec}_X(P^{\mathrm{H}}, c_1^{\mathrm{H}}; c_2^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, c_0^{\mathrm{H}}; c_2^{\mathrm{H}})} \qquad \textsc{Exec-While-End}\ \frac{P^{\mathrm{H}} \Rightarrow \neg b^{\mathrm{H}}}{\mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{while}\ b^{\mathrm{H}}\ \mathsf{do}\ c^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{skip})}$$

$$\textsc{Exec-Seq}\ \frac{\forall X.\ \mathsf{Exec}_X(P^{\mathrm{H}}, c_1^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(Q^{\mathrm{H}}, \mathsf{skip})}{\forall X.\ \mathsf{Exec}_X(P^{\mathrm{H}}, c_1^{\mathrm{H}}; c_2^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(Q^{\mathrm{H}}, c_2^{\mathrm{H}})} \qquad \textsc{Exec-While-Unroll}\ \frac{P^{\mathrm{H}} \Rightarrow b^{\mathrm{H}}}{\mathsf{Exec}_X(P^{\mathrm{H}}, \mathsf{while}\ b^{\mathrm{H}}\ \mathsf{do}\ c^{\mathrm{H}}) \Rightarrow \mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}}; \mathsf{while}\ b^{\mathrm{H}}\ \mathsf{do}\ c^{\mathrm{H}})}$$

Fig. 10. Proof Rules for Evaluating High-Level Programs

rules Exec-While-Unroll and Exec-While-End express that the execution predicate unrolls the loop as long as the guard b^{H} holds and terminates when the guard fails.

6.2 Proofs with the Execution Predicate

To demonstrate how our encoding enables relational proofs within standard Hoare logic, we revisit the examples from Sec. 2.2, and prove them using standard Hoare rules and the execution predicate.

Nondeterministic Programs. Consider the two nondeterministic programs from Example 1. Our goal is to prove the following standard Hoare triple:

$$\{\mathsf{Exec}_X(\mathsf{True}, \mathrm{y}^{\mathrm{H}} := \mathsf{nondet}(0,2))\}\ \mathrm{x}^{\mathrm{L}} := \mathsf{nondet}(0,1)\ \{\exists n.\ \mathsf{Exec}_X(\mathrm{y}^{\mathrm{H}} = n, \mathsf{skip}) \wedge \mathrm{x}^{\mathrm{L}} = n\}.$$

We begin by evaluating the low-level assignment $\mathrm{x}^{\mathrm{L}} := \mathsf{nondet}(0,1)$ demonically. This yields an postcondition: $\exists n \in \{0,1\}.\ \mathsf{Exec}_X(\mathsf{True}, \mathrm{y}^{\mathrm{H}} := \mathsf{nondet}(0,2)) \wedge \mathrm{x}^{\mathrm{L}} = n$. Next, we apply rule Exec-Nondet and angelically pick the same value n to update the execution predicate. Then we establish the desired postcondition. This standard Hoare logic proof, along with updating the execution predicate, mirrors the relational reasoning presented in Sec. 2.2.

Simple Sequential Programs. Recall the bitmask/set example, where the high-level program builds a set and the low-level program constructs a bitmask. Our goal is to prove:

$$\{\mathsf{Exec}_X(\mathsf{True}, \mathtt{set\_union})\}\ \mathtt{bit\_mask}\ \{\exists l.\ \mathsf{Exec}_X(\mathrm{s}^{\mathrm{H}} = l, \mathsf{skip}) \wedge \mathrm{x}^{\mathrm{L}} = \sum\nolimits_{a \in l} 2^a\}.$$

As shown in Fig. 5b, we begin with the high-level assignment $\mathrm{s}^{\mathrm{H}} := \emptyset$, updating the execution predicate using rules Exec-Seq and Exec-Assign. This yields the updated predicate: $\mathsf{Exec}_X(\mathrm{s}^{\mathrm{H}} = \emptyset, \mathrm{s}^{\mathrm{H}} := \mathrm{s}^{\mathrm{H}} \cup a_0;\ \mathrm{s}^{\mathrm{H}} := \mathrm{s}^{\mathrm{H}} \cup a_1)$. Next, we handle the low-level assignment $\mathrm{x}^{\mathrm{L}} := 0$, and apply the sequencing and assignment rules from standard Hoare logic. Then the goal becomes$\{\mathsf{Exec}_X(\mathrm{s}^{\mathrm{H}} = \emptyset,$

$\mathsf{s}^{\mathrm{H}} := \mathsf{s}^{\mathrm{H}} \cup a_0;\ \mathsf{s}^{\mathrm{H}} := \mathsf{s}^{\mathrm{H}} \cup a_1) \wedge \mathsf{x}^{\mathrm{L}} = 0\}\ \mathsf{x}^{\mathrm{L}} := \mathsf{x}^{\mathrm{L}} | (1 << a_0);\ \mathsf{x}^{\mathrm{L}} := \mathsf{x}^{\mathrm{L}} | (1 << a_1)\ \{\exists\, l.\ \mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = l, \mathsf{skip}) \wedge \mathsf{x}^{\mathrm{L}} = \sum_{a \in l} 2^a\}$. We proceed similarly for the remaining assignments: updating the execution predicate on the high-level side and applying standard Hoare rules on the low-level side. The final postcondition is $\mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = a_0, a_1, \mathsf{skip}) \wedge \mathsf{x}^{\mathrm{L}} = 2^{a_0} + 2^{a_1}$ if $a_0 \neq a_1$, and $\mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = a_0, \mathsf{skip}) \wedge \mathsf{x}^{\mathrm{L}} = 2^{a_0}$ otherwise, both of which imply the desired result.

Loops. When more items need to be recorded, we use a loop-based version as shown in Example 2. Our goal is to prove the following standard Hoare triple:

$$\{\mathrm{Exec}_X(\mathsf{True}, \mathtt{set\_union\_loop})\}\ \mathtt{bit\_mask\_loop}\ \{\exists\, l.\ \mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = l, \mathsf{skip}) \wedge \mathsf{x}^{\mathrm{L}} = \sum\nolimits_{a \in l} 2^a\}.$$

As before, we can evaluate the initial assignments to s^{H}, j^{H}, x^{L}, and i^{L} using standard Hoare rules along with the Exec-Seq and Exec-Assign rules to update the execution predicate. This reduces the goal to $\{\mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = \emptyset \wedge \mathsf{j}^{\mathrm{H}} = 0, \mathsf{while}\ \mathsf{j}^{\mathrm{H}} < 8\ \mathsf{do}\ \ldots) \wedge \mathsf{x}^{\mathrm{L}} = 0 \wedge \mathsf{i}^{\mathrm{L}} = 0\}\ \mathsf{while}\ \mathsf{i}^{\mathrm{L}} < 8\ \mathsf{do}\ \ldots\ \{\exists\, l.\ \mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = l, \mathsf{skip}) \wedge \mathsf{x}^{\mathrm{L}} = \sum_{a \in l} 2^a\}$. We then apply the standard while rule with the following invariant, which corresponds to the relational invariant (3):

$$\exists\, n\, l.\ \mathrm{Exec}_X(\mathsf{s}^{\mathrm{H}} = l \wedge \mathsf{j}^{\mathrm{H}} = n, \mathsf{while}\ \mathsf{j}^{\mathrm{H}} < 8\ \mathsf{do}\ \ldots) \wedge \mathsf{x}^{\mathrm{L}} = \sum\nolimits_{a \in l} 2^a \wedge \mathsf{i}^{\mathrm{L}} = n$$

This invariant expresses that both programs have processed the first n elements of the array. The set s^{H} contains the first n items, and the integer x^{L} encodes the same set as a bitmask. As long as the guard $\mathsf{i}^{\mathrm{L}} < 8$ holds, we know $\mathsf{j}^{\mathrm{H}} < 8$ also holds, allowing us to apply Exec-While-Unroll to unroll the high-level while statement. Each loop iteration then updates both sides independently as before. Once the loop exits, Exec-While-End applies on the high-level side, reducing its program to skip and yielding the desired postcondition.

6.3 Encoded Vertical Composition Rules and Compositional Reasoning

Relational Hoare logic supports two forms of vertical composition: combining a refinement proof with a standard proof of the high-level program (VC-FC), and composing two refinement proofs (VC-Refine). Here the linking operators $\odot$ and $\circ$ used in the rules are defined as follows:

Definition 19 (Assertion Linking). *For any binary assertion $\mathbb{P} \subseteq \Sigma^{\mathrm{L}} \times \Sigma^{\mathrm{H}}$ and any unary assertion $P^{\mathrm{H}} \subseteq \Sigma^{\mathrm{H}}$:*

$$\sigma^{\mathrm{L}} \models \mathbb{P} \odot P^{\mathrm{H}} \quad \textit{iff.} \quad \exists\, \sigma^{\mathrm{H}}.\ (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}) \models \mathbb{P} \wedge \sigma^{\mathrm{H}} \models P^{\mathrm{H}}.$$

For any binary assertions $\mathbb{P}_1 \subseteq \Sigma_1 \times \Sigma_2$ and $\mathbb{P}_2 \subseteq \Sigma_2 \times \Sigma_3$:

$$(\sigma_1, \sigma_3) \models \mathbb{P}_1 \circ \mathbb{P}_2 \quad \textit{iff.} \quad \exists\, \sigma_2.\ (\sigma_1, \sigma_2) \models \mathbb{P}_1 \wedge (\sigma_2, \sigma_3) \models \mathbb{P}_2.$$

The operator $\odot$ links a binary assertion with a unary assertion by existentially projecting over the high-level state, while $\circ$ composes two binary assertions by joining over their shared intermediate state. Both operators are special cases of the relational join operator [1].

Under our encoding, rules VC-FC and VC-Refine are transformed into the following forms:

$$\frac{\forall X.\ \vdash_\forall \{(\!|\mathbb{P} \wedge [c^{\mathrm{H}}]|\!)_X\}\ c^{\mathrm{L}}\ \{(\!|\mathbb{Q} \wedge [\mathsf{skip}]|\!)_X\} \qquad \vdash_\forall \{P^{\mathrm{H}}\}\ c^{\mathrm{H}}\ \{Q^{\mathrm{H}}\}}{\vdash_\forall \{\mathbb{P} \odot P^{\mathrm{H}}\}\ c^{\mathrm{L}}\ \{\mathbb{Q} \odot Q^{\mathrm{H}}\}}$$

$$\frac{\forall X_1.\ \vdash_\forall \{(\!|\mathbb{P}_1 \wedge [c_2]|\!)_{X_1}\}\ c_1\ \{(\!|\mathbb{Q}_1 \wedge [\mathsf{skip}]|\!)_{X_1}\} \qquad \forall X_2.\ \vdash_\forall \{(\!|\mathbb{P}_2 \wedge [c_3]|\!)_{X_2}\}\ c_2\ \{(\!|\mathbb{Q}_2 \wedge [\mathsf{skip}]|\!)_{X_2}\}}{\forall X.\ \vdash_\forall \{(\!|\mathbb{P}_1 \circ \mathbb{P}_2 \wedge [c_3]|\!)_X\}\ c_1\ \{(\!|\mathbb{Q}_1 \circ \mathbb{Q}_2 \wedge [\mathsf{skip}]|\!)_X\}}$$

The two encoded rules resemble specialized forms of the standard consequence rule. In particular, both can be derived using the consequence rule together with the rule ExIntro in standard Hoare logic. For the first rule, the key idea is extracting the initial high-level state σ_0^{H} from the existential

quantification in $\mathbb{P} \odot P^{\mathrm{H}}$, and instantiating X as the set of high-level states of executing c^{H} starting from σ_0^{H}. The second rule is then proved by instantiating X_2 as X and applying the first rule.

These rules become convenient when used with decomposed assertions. For example, a refinement judgment that expresses c^{L} refines c^{H} often appears as follows.

$$\langle \exists\, u. \lfloor \mathsf{storePre}^{\mathrm{L}}(u) \rfloor \wedge \lceil \mathsf{storePre}^{\mathrm{H}}(u) \rceil \wedge [c^{\mathrm{H}}] \rangle\, c^{\mathrm{L}}\, \langle \exists\, v. \lfloor \mathsf{storePost}^{\mathrm{L}}(v) \rfloor \wedge \lceil \mathsf{storePost}^{\mathrm{H}}(v) \rceil \wedge [\mathsf{skip}] \rangle$$

Here, the logical variables u and v serve to relate the low-level and high-level states and represent the initial and final data, respectively. For instance, in the bitmask/set example, the relational triple can be written as $\langle \lfloor \mathsf{True} \rfloor \wedge \lceil \mathsf{True} \rceil \wedge [\mathtt{set\_union}] \rangle\, \mathtt{bit\_mask}\, \langle \exists\, v.\, \lfloor \mathsf{x}^{\mathrm{L}} = \sum_{a \in v} 2^a \rfloor \wedge \lceil \mathsf{s}^{\mathrm{H}} = v \rceil \wedge [\mathsf{skip}] \rangle$. Under encoding, this form enables a convenient vertical composition with a high-level standard triple as follows, where the precondition assumes a constraint B_1 on the data stored in the state and the postcondition asserts that constraint B_2 will hold after execution.

$$\frac{\begin{array}{c} \forall X.\ \vdash_\forall \left\{ \exists\, u. \begin{array}{c} \mathsf{Exec}_X(\mathsf{storePre}^{\mathrm{H}}(u), c^{\mathrm{H}}) \\ \wedge\ \mathsf{storePre}^{\mathrm{L}}(u) \end{array} \right\}\, c^{\mathrm{L}}\, \left\{ \exists\, v. \begin{array}{c} \mathsf{Exec}_X(\mathsf{storePost}^{\mathrm{H}}(v), c^{\mathrm{H}}) \\ \wedge\ \mathsf{storePost}^{\mathrm{L}}(v) \end{array} \right\} \\ \vdash_\forall \{\exists\, u.\, B_1(u) \wedge \mathsf{storePre}^{\mathrm{H}}(u)\}\ c^{\mathrm{H}}\ \{\forall\, v.\, \mathsf{storePost}^{\mathrm{H}}(v) \Rightarrow B_2(v)\} \\ \forall\, u.\, B_1(u) \Rightarrow \mathsf{inhabitant}(\mathsf{storePre}^{\mathrm{H}}(u)) \end{array}}{\vdash_\forall \{\exists\, u.\, B_1(u) \wedge \mathsf{storePre}^{\mathrm{L}}(u)\}\ c^{\mathrm{L}}\ \{\exists\, v.\, B_2(v) \wedge \mathsf{storePost}^{\mathrm{L}}(v)\}}$$

The side condition $\forall\, u.\, B_1(u) \Rightarrow \mathsf{inhabitant}(\mathsf{storePre}^{\mathrm{H}}(u))$ ensures that, for any logical variable u such that $B_1(u)$, there exists some high-level state satisfying $\mathsf{storePre}^{\mathrm{H}}(u)$. It reflects the existential quantification of $\mathbb{P} \odot P^{\mathrm{H}}$ in the encoded VC-FC rule. Intuitively, this requirement lies in that to derive properties of the low-level program based on the high-level one, we need to ensure that there exists a valid high-level execution satisfying the properties.

Example 4. *Assume we have proved that* `set_union` *ensures* s^{H} *is non-empty after termination, as described by the triple* $\{\mathsf{True}\}\ \mathtt{set\_union}\ \{\mathsf{s}^{\mathrm{H}} \neq \emptyset\}$. *The goal is to prove that* `bit_mask` *ensures* x^{L} *is larger than 0 after termination, as described by the triple* $\{\mathsf{True}\}\ \mathtt{bit\_mask}\ \{\mathsf{x}^{\mathrm{L}} > 0\}$.

We apply the vertical composition rule with $B_1(u) \triangleq \mathsf{True}$ and $B_2(v) \triangleq v \neq \emptyset$. This establishes a low-level triple $\{\mathsf{True}\}\ \mathtt{bit\_mask}\ \{\exists\, v.\, v \neq \emptyset \wedge \mathsf{x}^{\mathrm{L}} = \sum_{a \in v} 2^a\}$, whose postcondition implies $\mathsf{x}^{\mathrm{L}} > 0$.

7 Case Studies

In Sec. 6.2, we have already demonstrated how standard Hoare reasoning applies to several simple examples, including nondeterminism, sequencing, and loops. Beyond these, we have formalized a collection of more realistic case studies in Rocq, covering heap mutation, abstract data structures, and function calls:

- **Mergesort**: A low-level implementation over singly linked lists refines an abstract version over lists. This captures the correspondence between heap-based lists and pure lists.
- **KMP String Matching**: An array-based KMP implementation refines a list-based program. The refinement proof involves reasoning about array index manipulation.
- **Binary Search Trees**: A loop-based insertion program over a mutable tree refines a recursive version over an abstract binary tree model. The two programs differ in control structure but preserve the same search behavior.
- **Depth-First Search**: A recursive DFS program that traverses a graph represented by adjacency lists refines an iterative version over an abstract graph model. The relational specification ensures both programs explore the same set of reachable nodes.

In all examples, the low-level programs are written in a heap-based imperative language. Their specifications are written in a standard separation logic extended with the execution predicate.

```
struct list{ int data; struct list *next; };
struct list * merge^L(struct list *x^L,
                      struct list *y^L)
{ struct list * r^L;
  struct list ** t^L = &r^L;
  while (true) {
    if (x^L == NULL) {*t^L = y^L; return r^L;}
    else if (y^L == NULL)
      {*t^L = x^L; return r^L;}
    else if (x^L -> data < y^L -> data)
      { *t^L = x^L;
        t^L = &(x^L->next); x^L = x^L->next; }
    else { *t^L = y^L;
        t^L = &(y^L->next); y^L = y^L->next; } } }
```

```
merge^H(p^H, q^H) ≜
  r^H := empty;
  while (p^H ≠ empty ∨ q^H ≠ empty) {
    if (p^H = empty) then r^H := r^H q^H
    else if (q^H = empty) then r^H := r^H p^H
    else {z_1^H := head(p^H);
      z_2^H := head(q^H);
      choice(assume (z_1^H ≤ z_2^H));
          p^H := tail(p^H);
          r^H := r^H z_1^H,
          assume (z_2^H ≤ z_1^H));
          q^H := tail(q^H);
          r^H := r^H z_2^H } };
  return r^H
```

Fig. 11. Low-Level Program (Left) Merges Two Singly Linked Lists with Explicit Pointer Manipulation, and High-Level Program (Right) Merges Two Abstract Lists.

The assertion language extends that in Sec. 5 with standard heap assertions as follows, where emp denotes the empty heap, $e_1^L \mapsto e_2^L$ specifies a singleton heap mapping address e_1^L to value of e_2^L, and $\bar{P}_1^L * \bar{P}_2^L$ asserts that the heap can be split into two disjoint parts satisfying $\bar{P}_1^L$ and $\bar{P}_2^L$, respectively.

$$\bar{P}^L ::= B \mid \mathsf{Exec}_X(P^H, c^H) \mid \mathsf{emp} \mid e^L \mapsto e^L \mid p(\vec{e}^L) \mid \bar{P}^L \wedge \bar{P}^L \mid \exists a.\, \bar{P}^L(a) \mid \bar{P}^L \vee \bar{P}^L \mid \bar{P}^L * \bar{P}^L$$

Notably, as supported by the syntactic encoding in Sec. 5.2, the inclusion of separation logic constructs in low-level assertions enables the application of the standard frame rule:

$$\frac{\{P^L\}\ c^L\ \{Q^L\}}{\{P^L * F^L\}\ c^L\ \{Q^L * F^L\}}$$

This is because we reason entirely within standard Hoare logic, using the low-level proof rules together with updating rules for the execution predicate. In contrast, the high-level programs are written using standard first-order assertions without separation logic. We believe that extending separation logic to high-level programs has limited benefit for sequential verification, though we have a supplementary discussion of such an extension in our extended version [45].

In the remainder of this section, we present a detailed proof of the merging algorithm from the mergesort example. While we have formalized the full refinement between the low-level and high-level implementations of mergesort in Rocq, we focus here on the merge step, as it is sufficient to show how standard Hoare logic with the execution predicate can be used to prove the refinement between heap-based list manipulation and abstract list concatenation.

7.1 Heap-Based Merge Refines List-Based Merge

Merging two lists is naturally specified nondeterministically: when head elements are equal, either can be chosen. In contrast, real-world implementations (e.g., in C language) make this choice deterministically, as shown in Fig. 11. Our goal is to verify that such a deterministic, heap-based implementation refines a nondeterministic list-based program.

On the left of Fig. 11 is the low-level implementation merge^L, which merges two sorted singly linked lists, x^L and y^L, into a new list. The key idea is to maintain a pointer-to-pointer variable t^L that always refers to the tail of the result list being constructed. On the right is the high-level version

$\mathsf{merge}^{\mathrm{H}}$, which operates over abstract lists using standard list operations: $l_1 l_2$ for concatenation, $l_1 z_1$ to append an element, $\mathsf{head}(l)$ to access the first element, and $\mathsf{tail}(l)$ to get the rest. Besides differing in data representation, the two implementations also differ in behavior. First, when both input lists are empty, the high-level program skips the loop, while the low-level version enters. Second, when both head elements are equal, the high-level program allows either element to be selected, while the low-level program always chooses the element from y^{L}. Their refinement can be expressed as a relational Hoare triple, where the separation logic predicate $\mathsf{sll}(p, l)$ relates a linked list segment starting at address p in the heap to an abstract list l:

$$\left\langle \exists\, l_x, l_y.\ \begin{array}{l} \lfloor \mathsf{sll}(\mathrm{x}^{\mathrm{L}}, l_x) * \mathsf{sll}(\mathrm{y}^{\mathrm{L}}, l_y) \rfloor \wedge \\ \lceil \mathrm{p}^{\mathrm{H}} = l_x \wedge \mathrm{q}^{\mathrm{H}} = l_y \rceil \wedge [\mathsf{merge}^{\mathrm{H}}(\mathrm{p}^{\mathrm{H}}, \mathrm{q}^{\mathrm{H}})] \end{array} \right\rangle \mathsf{merge}^{\mathrm{L}}(\mathrm{x}^{\mathrm{L}}, \mathrm{y}^{\mathrm{L}}) \left\langle \exists\, l_r.\ \begin{array}{l} \lfloor \mathsf{sll}(\mathrm{r}^{\mathrm{L}}, l_r) \rfloor \wedge \\ \lceil \mathrm{r}^{\mathrm{H}} = l_r \rceil \wedge [\mathsf{skip}] \end{array} \right\rangle$$

This triple states that if the initial heap contains linked lists at x^{L} and y^{L} corresponding to high-level variables p^{H} and q^{H}, then the low-level program produces a list r^{L} that matches r^{H}.

With the execution predicate, we can express the refinement as a standard Hoare triple:

$$\left\{ \exists\, l_x, l_y.\ \begin{array}{l} \mathsf{Exec}_X(\mathrm{p}^{\mathrm{H}} = l_x \wedge \mathrm{q}^{\mathrm{H}} = l_y, \mathsf{merge}^{\mathrm{H}}(\mathrm{p}^{\mathrm{H}}, \mathrm{q}^{\mathrm{H}})) \\ \wedge\ \mathsf{sll}(\mathrm{x}^{\mathrm{L}}, l_x) * \mathsf{sll}(\mathrm{y}^{\mathrm{L}}, l_y) \end{array} \right\} \mathsf{merge}^{\mathrm{L}}(\mathrm{x}^{\mathrm{L}}, \mathrm{y}^{\mathrm{L}}) \left\{ \exists\, l_r.\ \begin{array}{l} \mathsf{Exec}_X(\mathrm{r}^{\mathrm{H}} = l_r, \mathsf{skip}) \\ \wedge\ \mathsf{sll}(\mathrm{r}^{\mathrm{L}}, l_r) \end{array} \right\}$$

As in Sec. 6.2, we begin by applying standard Hoare rules together with Exec-Seq and Exec-Assign to handle the initial setup before entering the loop. The core of the proof then lies in establishing a loop invariant that relates the current state of the low-level heap to the high-level state. Intuitively, the invariant should track three components: the remaining portions of the input lists, and the portion of the result list that has already been constructed from pointer r^{L} to pointer t^{L}. To express the result list, we introduce a predicate $\mathsf{sllbseg}(p, q, l)$ to describe a list segment l that begins at the address stored at pointer p and ends at pointer q. The loop invariant is then stated as:

$$\exists\, l_x, l_y, l_r, p_t.\ \begin{array}{l} \mathsf{Exec}_X(\mathrm{p}^{\mathrm{H}} = l_x \wedge \mathrm{q}^{\mathrm{H}} = l_y \wedge \mathrm{r}^{\mathrm{H}} = l_r, \mathsf{while}\ \ldots\ \mathsf{do}\ \ldots) \\ \wedge\ \mathsf{sll}(\mathrm{x}^{\mathrm{L}}, l_x) * \mathsf{sll}(\mathrm{y}^{\mathrm{L}}, l_y) * \mathsf{sllbseg}(\&\mathrm{r}^{\mathrm{L}}, \mathrm{t}^{\mathrm{L}}, l_r) * \mathrm{t}^{\mathrm{L}} \mapsto p_t \end{array}$$

We verify that the loop invariant is preserved across all paths through the loop body. When either input list is empty, we assume without loss of generality that $\mathrm{x}^{\mathrm{L}} = \mathtt{NULL}$ (the case where $\mathrm{y}^{\mathrm{L}} = \mathtt{NULL}$ is symmetric). In the low-level program, $*\mathrm{t}^{\mathrm{L}}$ is assigned to y^{L} and b^{L} is set to 1. We apply the standard assignment rule to handle these assignments, and use the frame rule to preserve the parts of the heap that are unchanged (e.g., the already constructed result list). This results in a heap satisfying the invariant with $\mathrm{t}^{\mathrm{L}} \mapsto \mathrm{y}^{\mathrm{L}}$ and $\mathrm{b}^{\mathrm{L}} = 1$. On the high-level side, the execution predicate remains unchanged, and the loop invariant holds. When both input lists are non-empty, we assume that $l_x = z_x l_x'$ and $l_y = z_y l_y'$. For the high-level program, we apply the rule Exec-While-Unroll to unroll one iteration of the loop. Then, Exec-Assign allows us to update the execution predicate into $\mathsf{Exec}_X(\mathrm{p}^{\mathrm{H}} = l_x \wedge \mathrm{q}^{\mathrm{H}} = l_y \wedge \mathrm{r}^{\mathrm{H}} = l_r \wedge \mathrm{z}_1^{\mathrm{H}} = z_x \wedge \mathrm{z}_2^{\mathrm{H}} = z_y, \mathsf{choice}(\ldots, \ldots); \mathsf{while}\ \ldots\ \mathsf{do}\ \ldots)$. If $z_x < z_y$, the low-level program appends the node from x^{L} to the result list. On the heap, we unfold $\mathsf{sll}(\mathrm{x}^{\mathrm{L}}, l_x)$ and transfer its head node to the list segment described by $\mathsf{sllbseg}(\&\mathrm{r}^{\mathrm{L}}, \mathrm{t}^{\mathrm{L}}, l_r)$, extending it to $l_r z_x$. The pointer t^{L} is then updated to point to the next insertion point, which is exactly the new x^{L}. These steps are handled by standard separation logic reasoning with the assignment and frame rules. For the high-level program, we apply Exec-Pure and Exec-ChoiceL to select the left branch. We then use Exec-Assume to validate the condition, and Exec-Assign to update both p^{H} and r^{H} appropriately. This ensures that the execution predicate and heap remain in sync with the invariant. If $z_x \geq z_y$, the argument is symmetric: the low-level program appends the node from y^{L}, and the high-level program selects the right branch of the nondeterministic choice.

The loop exits when either l_x = empty or l_y = empty. Hence, the high-level loop condition ($\mathsf{p}^{\mathrm{H}} \neq$ empty $\vee$ $\mathsf{q}^{\mathrm{H}} \neq$ empty) must be false, and rule Exec-While-End can reduce the high-level program in the execution predicate to skip. For the low-level side, we consider l_x = empty (the case where l_x = empty is symmetric). Then $\mathsf{sll}(\mathsf{x}^{\mathrm{L}}, l_x)$ simplifies to emp, and combining the remaining heaps gives $\mathsf{sll}(\mathsf{r}^{\mathrm{L}}, l_r l_y)$, which matches the final postcondition in the desired Hoare triple.

8 Discussion

This paper has modified the validity of the relational Hoare triple with configuration refinement. Does this modification change the meaning of the refinement represented in relational triples? Under our modified definition, some triples previously invalid become valid. For example, the triple $\langle \mathsf{x} = \mathsf{y} \wedge [\mathsf{skip}] \rangle\ \mathsf{x} := \mathsf{x} + 1\ \langle \mathsf{x} = \mathsf{y} \wedge [\mathsf{y} := \mathsf{y} - 1] \rangle$ is valid under configuration refinement. This is because we no longer require updates of high-level program configuration to precisely track program reductions. Instead, we allow rewriting the high-level configuration into a new one that ultimately produces the same final results. However, for both relational Hoare triples based on multi-step transitions and configuration refinement, when used to express refinement, the high-level program in the postcondition must terminate (e.g., a skip statement). Thus, under both definitions, the relational triple $\langle \mathbb{P} \wedge [c^{\mathrm{H}}] \rangle\ c^{\mathrm{L}}\ \langle \mathbb{Q} \wedge [\mathsf{skip}] \rangle$ always represents that the low-level program c^{L} refines the high-level program c^{H}, as shown in Fig. 2. Besides, we have formalized proof rules in Rocq, confirming that the proof theory itself is preserved.

Does the final states set X outside the encoded standard Hoare triple introduce any complications? Simply put: No. Although our theory (as shown in Theo. 4) transforms a relational Hoare triple into a standard Hoare triple universally quantified by a subset X, this final-state set X is merely a placeholder. In practice, proving refinement reduces to proving a standard Hoare triple where we do not need to reason about what X is in the verification process. For instance, the examples in Sec. 6.2 and our case studies show that X is just a placeholder during verification. In fact, universally quantified Hoare triples (not universally quantified assertions) are widely used in deduction-based program verification. For example, in the C program verification tool VST, a C function's specification has the form: $\forall a.\ \{P(x, y, z, \cdots, a)\}\ \mathsf{ret} = f(x, y, z, \cdots, z)\ \{Q(\mathsf{ret}, a)\}$, where a serves as a logical variable.

What existing verification infrastructure from standard Hoare logic can be reused, and how? Our encoding reduces verifying ∀∃ relational properties to proving standard Hoare triples about low-level programs. Consequently, any proof strategy or tool compatible with standard Hoare logic and the execution predicate can be reused. For example, tactics provided by VST-floyd [9]—such as the `forward` tactic for automatically forwardly stepping through code—remain fully reusable. Provers can thus rely on well-tested automation for local reasoning steps (e.g., pointer manipulations in the low-level program). In particular, we have integrated the execution predicate into a standard Hoare logic tool [47] and applied it in case studies. In these relational proofs, we reuse the tool's symbolic executor, verification condition generator, and logic entailment infrastructure.

Does reasoning with the execution predicate increase complexity relative to existing relational approaches, and is it amenable to automation? We argue it does not introduce additional complexity. Owing to our encoding theory, relational proof rules can be encoded into standard rules that incorporate the execution predicate. In prior relational works, updating the high-level configuration involves an angelic triple, as shown in rule High-Focus. In our encoding, updating the execution predicate is accomplished by a consequence rule, and can also be captured by an angelic triple. So, any strategies that can automate angelic Hoare triples can be adapted to handle $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$. For instance, verifier Hypra [11] provides a `hint` annotation to guide the instantiation of existential

quantifiers in the SMT solver and thereby facilitates proof automation. We believe a similar strategy can be applied to automate updating the execution predicate. Furthermore, we provide update rules, such as Exec-Pure in Fig. 10, to evaluate the high-level program within the execution predicate. Existing ∀∃ relational approaches also require high-level side rules, such as rule PURE-R in ReLoC [19]. In particular, our update rules resemble these high-level rules. Thus, our encoding neither fundamentally changes the way people reason about program refinement nor adds complexity, and it should be readily automated with existing proof infrastructure.

Since the ∀∃ relational Hoare logic can already be encoded into the Hoare logic using ghost states and invariants, why propose a separate encoding based on standard Hoare logic. In principle, a foundationally formalized verification tool could choose either standard Hoare logic or the Hoare logic with ghost states and invariants as its base framework. Our main contribution is theoretical and shows a different way of unifying relational Hoare logic with standard Hoare logic. While employing the Hoare logic with ghost states and invariants is common in concurrent program verification, standard Hoare logic sometimes enjoys better meta-theoretical properties in some sense. For example, the conjunction rule typically remains valid in standard Hoare logic but may fail for Hoare logics with ghost states. Moreover, the soundness of standard Hoare logic is also easier to formalize, compared to the Hoare logic with ghost states and invariants. For example, as presented in Iris [24], the weakest precondition definition involving ghost states and invariants (in its Sec. 7.3) is significantly more complex than the one without them (in its Sec. 6.3).

9 Related Work

To the best of our knowledge, this work provides the first theory that encodes the ∀∃ relational Hoare logic into a standard Hoare logic framework. By introducing the execution predicate $\mathsf{Exec}_X(P^{\mathrm{H}}, c^{\mathrm{H}})$ along with its associated proof rules, our encoding allows developers to leverage standard Hoare logic for relational reasoning without modifying the core framework or re-verifying its soundness. In this section, we discuss related works including approaches to encoding one type of relational Hoare logic into some Hoare logic and frameworks for the ∀∃ relational Hoare logic.

9.1 Logic Encoding

Although our primary contribution focuses on encoding ∀∃ relational properties, there are alternative relational logics. For instance, as discussed in the introduction, when dealing with deterministic programs, two-safety properties suffice to establish refinement. Below, we also discuss how such logics can also be encoded within some Hoare logic frameworks.

Encoding Two-Safety. Barthe [7] proposed self-composition, which enables proving 2-safety properties in standard Hoare logic. This method is both straightforward and effective: for two programs, c_1 and c_2, variables in c_2 are renamed to ensure disjoint program states, producing a renamed program c_2'. The two programs can then be sequentially composed as $c_1; c_2'$. Therefore, the 2-safety property for programs c_1 and c_2 is reduced to some property of all possible executions of program $c_1; c_2'$, allowing reasoning within standard Hoare logic. D'Osualdo et al. [16] employed the weakest liberal precondition assertion [9] (wlp) and introduced a Logic for Hyper-triple Composition (LHC) to provide various composition rules for reasoning about k-safety properties—a generalization of ∀∀ relational Hoare logic. Focusing on 2-safety, the hyper-triple $\mathbb{P} \Rightarrow \mathsf{wlp}([c^{\mathrm{L}}, c^{\mathrm{H}}], \mathbb{Q})$ used in their work precisely encodes the ∀∀ relational Hoare logic. In contrast, this paper focuses

[9]The weakest precondition assertion wp used in their paper is referred to as weakest liberal precondition wlp in this paper and other works [15].

on representing the ∀∃ relational Hoare logic using standard Hoare logic. This task introduces additional complexities, particularly when addressing nondeterministic behavior.

Encoding Data Refinement. Roever and Engelhardt [13] explored how to prove a low-level program refines a high-level specification within a generalized version of Hoare logic. To be precise, their framework extends standard Hoare logic by introducing *specification statements* in the form $P \rightsquigarrow Q$. The statement $P \rightsquigarrow Q$ is used to represent the most general program c that satisfies the Hoare triple $\{P\}\ c\ \{Q\}$. They demonstrated that the refinement relationship between a low-level program c^{L} and a high-level specification $P^{\mathrm{H}} \rightsquigarrow Q^{\mathrm{H}}$ can be encoded as a Hoare triple of the low-level program c^{L}. For example, they proved that given a binary assertion $\mathbb{R}$, the forward simulation between a low-level program c^{L} and a high-level specification $P^{\mathrm{H}} \rightsquigarrow Q^{\mathrm{H}}$, denoted as

$$c^{\mathrm{L}} \subseteq^{L}_{\mathbb{R}} P^{\mathrm{H}} \rightsquigarrow Q^{\mathrm{H}}$$

can be encoded as the following Hoare triple [10]:

$$\{\lambda\sigma^{\mathrm{L}}.\, \exists\sigma^{\mathrm{H}}.\, (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}) \models \mathbb{R} \land \sigma^{\mathrm{H}} \models P^{\mathrm{H}}\}\ c^{\mathrm{L}}\ \{\lambda\sigma^{\mathrm{L}}.\, \exists\sigma^{\mathrm{H}}.\, (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}) \models \mathbb{R} \land \sigma^{\mathrm{H}} \models Q^{\mathrm{H}}\} \tag{7}$$

In their framework, specification statements play a central role, and the goal is to prove that a low-level program refines a high-level specification. Instead, our encoding theory does not require specification statements and focuses on establishing refinement directly between a low-level program and a high-level program. Such direct refinement is crucial because a high-level program can be used to prove various specifications and to enable further refinements. For instance, a recursive DFS implementation using heaps can be incrementally refined into a recursive DFS without heaps, and later into an iterative DFS without heaps [27]. Note that that specifying the high-level specification as $(\sigma^{\mathrm{H}} = \sigma^{\mathrm{H}}_0) \rightsquigarrow (\sigma^{\mathrm{H}}_0, \sigma^{\mathrm{H}}) \in [\![c^{\mathrm{H}}]\!]_{\mathsf{nrm}}$ implies that the encoding (7) captures exactly the refinement of low-level program c^{L} and high-level program c^{H}:

$$\{\lambda\sigma^{\mathrm{L}}.\, (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}_0) \models \mathbb{R}\}\ c^{\mathrm{L}}\ \{\lambda\sigma^{\mathrm{L}}.\, \exists\sigma^{\mathrm{H}}.\, (\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}) \models \mathbb{R} \land (\sigma^{\mathrm{H}}_0, \sigma^{\mathrm{H}}) \in [\![c^{\mathrm{H}}]\!]_{\mathsf{nrm}}\}$$

This encoding still suffers from the same limitation as the naive approach in Sec. 4. Precisely, since it treats pre- and postconditions differently, it is unable to support powerful relational proof rules (e.g., Rel-Seq and Rel-Wh). Overcoming this limitation requires a uniform encoding for preconditions and postconditions, which our coding theory achieves. Besides, the refinement relationship between a low-level program c^{L} and a high-level program $P^{\mathrm{H}} \rightsquigarrow Q^{\mathrm{H}}$ can be proved using the encoded vertical composition rule.

Encoding ∀∃ Properties into Product Programs. In the context of product programs, Barthe [6] has incorporated ∀∃ pattern into the program semantics to support refinement reasoning. They introduced an asymmetric concept of product programs through control-flow graphs. This concept involves the use of universal quantification over the execution traces of the low-level program and existential quantification over those of the high-level program. However, in this paper, the ∀∃ pattern is hidden in the encoded assertions and we employ the denotational semantics to capture observational behaviors.

Encoding ∀∃ Properties into a Hoare logic with Ghost States and Invariants. In concurrent program verification, Turon et al. [41] proposed treating high-level programs as resources, and Iris [24, 25] established a $(\forall\exists)^{\omega}$ separation Hoare logic with ghost states and invariant. Iris employs weakest preconditions as primitive and a Hoare triple $\{P\}\ c\ \{Q\}$ can be written as $P \Rightarrow \mathsf{wp}(c, Q)$ [11]. When

[10]This encoding requires that $\mathbb{R}$ is total and functional. That is for any low-level state σ^{L}, there exists a unique high-level state σ^{H} such that $(\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}) \models R$.

[11]The triple is actually defined as $\Box(P \mathrel{-\!\!*} \mathsf{wp}(c, Q))$. We omit the persistence modality $\Box$ and replace the separating implication $\mathrel{-\!\!*}$ with $\Rightarrow$ for simplicity.

ghost states and invariants are involved, we write $\mathrm{wp}_I^{\mathrm{gst}}(c, Q)$ to denote the weakest precondition. Intuitively, as Fig. 12 [12] shows, $\mathrm{wp}_I^{\mathrm{gst}}(c, Q)$ is the largest set of physical/ghost-state pairs such that:

- if program c can reduce, then we open the invariant I and demonically obtain a physical state σ and a matching ghost state σ^{G} such that $(\sigma, \sigma^{\mathrm{G}}) \models I$. For any possible small-step transition $(\sigma, c) \to (\sigma_2, c_2)$, we can angelically update σ^{G} to some new ghost state σ_2^{G} in a way that preserves the invariant I. This update process continues for c_2.
- if program c ends, then postcondition Q holds.

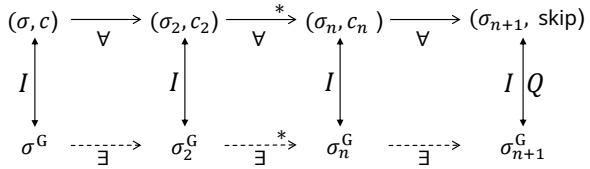

Fig. 12. Hoare Triples with Ghost States and Invariants

Subsequent frameworks [19, 37, 38] leverage this $(\forall\exists)^\omega$ Hoare logic to encode the $\forall\exists$ relational Hoare logic for verifying program refinement. A common strategy is to instantiate the ghost state as a high-level program configuration and set the invariant $\mathcal{I}$ as all possible configurations from an initial configuration $(\sigma_0^{\mathrm{H}}, c_0^{\mathrm{H}})$ [19].

$$\mathcal{I} \triangleq \lambda\sigma^{\mathrm{L}}, \sigma^{\mathrm{H}}, c^{\mathrm{H}}.\, (\sigma_0^{\mathrm{H}}, c_0^{\mathrm{H}}) \to^* (\sigma^{\mathrm{H}}, c^{\mathrm{H}})$$

Since this invariant imposes constraints only on the ghost state rather than the physical state, the low-level program's $\forall$ transitions do not depend on the previous $\exists$ updates of the high-level configuration. Then $\exists$ can be moved behind, $\forall$ can be moved forward, and all $\forall$s and all $\exists$s can be gathered together, reducing the resulting weakest precondition to a $\forall\exists$ pattern. Then the refinement between programs c^{L} and c^{H} is encoded as $\exists\sigma_0^{\mathrm{H}}, c_0^{\mathrm{H}}.\, \mathcal{I} \Rightarrow \mathrm{wp}_{\mathcal{I}}^{\mathrm{gst}}(c^{\mathrm{L}}, \mathrm{StateMatch})$. In contrast, this paper aims to encode the $\forall\exists$ relational Hoare logic into the $\forall$-structured standard Hoare logic.

9.2 Relational Hoare Logic Frameworks

Relational Hoare logic, introduced by Benton [8], extends standard Hoare logic to relate two programs. Many relational verification frameworks have been proposed to verify relational properties, including the $\forall\forall$ properties [5, 26, 32, 44, 48] and the $\forall\exists$ properties [6, 10]. To handle function calls in relational program verification, Song et al. [36] introduced the judgment $S \vdash c^{\mathrm{L}} \sqsubseteq c^{\mathrm{H}}$, which represents that c^{L} refines c^{H} when the modules are used according to the specification S. They proposed to insert assume and assert statements in programs to encode the preconditions and postconditions of functions, and demonstrated that these inserted statements can be safely erased. In our work, since we reduce relational Hoare logic to standard Hoare logic, function calls can be straightforwardly handled due to the direct application of the call rules in standard Hoare logic.

Some relational frameworks [5, 14, 32, 34] employ relational Hoare quadruples as the primary judgment and derive various proof rules for quadruples. Notably, quadruples representing program refinement, denoted as $\{\mathbb{P}\}\ c^{\mathrm{L}} \preceq c^{\mathrm{H}}\ \{\mathbb{Q}\}$, also exhibit a $\forall\exists$ pattern: for any execution of the low-level program c^{L}, there exist an execution of the high-level program c^{H} to capture it. Then quadruples can be transformed into equivalent relational Hoare triples, which in turn can be encoded into standard Hoare triples. However, these works do not investigate how to encode the $\forall\exists$-structured relational quadruples directly into $\forall$-structured standard triples. Besides, some works [2, 33] based

[12] The logic in Iris is a separation logic; however, for simplicity, frames have been omitted in this figure.

on quadruples focus on the alignment problem between two programs. We believe that approaches for alignment in these frameworks can also be adapted to relational Hoare triples.

Hyper Hoare logic [12] extends standard Hoare logic by employing assertions over sets of states as preconditions and postconditions. This work supports universal and existential quantifiers over these sets and can be used to reason about the ∀∃ properties of a single program. However, this work does not address the reduction of ∀∃ relational Hoare logic to standard Hoare logic.

Various verification tools, including Dafny [29], Verus [28], VeriFast [23], and Why3 [17], employ ghost states and ghost code to assist in proving properties of the real (i.e., compiled or executable) code. These tools ensure that real code with ghost extensions simulates the original real code after the ghost elements are erased. In other words, they do not ensure that a low-level (real) program refines a high-level (ghost) program and then prove the correctness of that high-level program.

IronFleet [21], built on Dafny, verifies that the pre- and post-states of the implementation match a transition in a high-level (abstract) state machine. This approach treats the high-level program as a relation over pre- and post-states, similar to the $P \rightsquigarrow Q$ specifications of Roever and Engelhardt [13]. The high-level programs in both approaches lack structural constructs or intermediate states. Consequently, when the low-level implementation includes loops, it is difficult to relate intermediate iterations to specific control points in the high-level program. Refinement proofs in both approaches are indeed standard Hoare proofs showing that the execution of the low-level program preserves a relational specification between its initial and final states.

9.3 Other Related Work

Incorrectness logic [35], which is proposed to reason about the presence of bugs, shows a ∀∃ pattern. Its judgment $[P]\ c\ [Q]$ describes that for any state σ within postcondition Q, there exists a valid execution from some state in precondition P, which is exactly the reverse direction of the angelic Hoare triple $\vdash_\exists \{P\}\ c\ \{Q\}$ in this paper.

10 Conclusion and Future Work

In this paper, we introduce the first encoding theory that reduces the ∀∃ relational Hoare logic to the ∀-structured standard Hoare logic. This encoding theory shows that, with our execution predicate, standard Hoare logic can verify program refinement. Consequently, relational proofs can be built on existing formalizations of standard Hoare logic and their associated proof strategies.

However, our syntactic encoding targets decomposed assertions, which requires users to explicitly write relational decomposed assertions when employing standard Hoare logic with our execution predicate to verify ∀∃ properties. Additionally, this work does not yet address concurrency or liveness properties, which are essential in many verification scenarios.

In the future, we aim to adapt the encoding theory to support reasoning about refinements that involve more complex observational behaviors. We also plan to extend the encoding to accommodate additional language features, including concurrent programs.

Acknowledgments

We thank David A. Naumann, Zhenjiang Hu, Zhongye Wang, Xinyi Wan, Yiyuan Cao, and Zhiyi Wang for their valuable feedback. We are also grateful to the anonymous reviewers for their constructive comments and suggestions. This work was supported by NSF China 62472274.

Data-Availability Statement

An artifact supporting this paper is available on Zenodo [46]. It provides a Rocq-based formalization of our encoding theory, machine-checked proofs, and case studies (BST, DFS, mergesort, and KMP) to demonstrate relational verification in standard Hoare logic.

References

[1] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. 1979. The Theory of Joins in Relational Databases. *ACM Trans. Database Syst.* 4, 3 (1979), 297–314. doi:10.1145/320083.320091

[2] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 573–603. doi:10.1145/3571213

[3] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. doi:10.1007/978-3-642-19718-5_1

[4] Andrew W. Appel. 2022. Coq's vibrant ecosystem for verification engineering (invited talk). In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 2–11. doi:10.1145/3497775.3503951

[5] Anindya Banerjee, Ramana Nagasamudram, David A. Naumann, and Mohammad Nikouei. 2022. A Relational Program Logic with Data Abstraction and Dynamic Framing. *ACM Trans. Program. Lang. Syst.* 44, 4 (2022), 25:1–25:136. doi:10.1145/3551497

[6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7734)*, Sergei N. Artëmov and Anil Nerode (Eds.). Springer, 29–43. doi:10.1007/978-3-642-35722-0_3

[7] G. Barthe, P.R. D'Argenio, and T. Rezk. 2004. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* 100–114. doi:10.1109/CSFW.2004.1310735

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

[9] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. doi:10.1007/S10817-018-9457-5

[10] Martin Clochard, Claude Marché, and Andrei Paskevich. 2020. Deductive verification with ghost monitors. *Proc. ACM Program. Lang.* 4, POPL (2020), 2:1–2:26. doi:10.1145/3371070

[11] Thibault Dardinier, Anqi Li, and Peter Müller. 2024. Hypra: A Deductive Program Verifier for Hyper Hoare Logic. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1279–1308. doi:10.1145/3689756

[12] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI, Article 207 (June 2024), 25 pages. doi:10.1145/3656437

[13] Willem-Paul de Roever and Kai Engelhardt. 1998. *Simulation and Hoare Logic.* Cambridge University Press, 132–145. doi:10.1017/CBO9780511663079.008

[14] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational ∀ ∃ Properties. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 67–87. doi:10.1007/978-3-031-21037-2_4

[15] Edsger Wybe Dijkstra. 1997. *A Discipline of Programming* (1st ed.). Prentice Hall PTR, USA.

[16] Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 135 (Oct. 2022), 26 pages. doi:10.1145/3563298

[17] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: where programs meet provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 125–128. doi:10.1007/978-3-642-37036-6_8

[18] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf

[19] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *CoRR* abs/2006.13635 (2020). arXiv:2006.13635 https://arxiv.org/abs/2006.13635

[20] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498689

[21] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. doi:10.1145/2815400.2815428

[22] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. doi:10.1145/363235.363259

[23] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55. doi:10.1007/978-3-642-20398-5_4

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. doi:10.1145/2676726.2676980

[26] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 327–337. doi:10.1145/1542476.1542513

[27] Peter Lammich and René Neumann. 2015. A Framework for Verifying Depth-First Search Algorithms. In *Proceedings of the 2015 Conference on Certified Programs and Proofs* (Mumbai, India) *(CPP '15)*. Association for Computing Machinery, New York, NY, USA, 137–146. doi:10.1145/2676724.2693165

[28] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (April 2023), 30 pages. doi:10.1145/3586037

[29] K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. doi:10.1007/978-3-642-17511-4_20

[30] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. doi:10.1145/2491956.2462189

[31] Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 385–399. doi:10.1145/2837614.2837635

[32] Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* 4, POPL (2020), 4:1–4:33. doi:10.1145/3371072

[33] Ramana Nagasamudram and David A. Naumann. 2021. Alignment Completeness for Relational Hoare Logics. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. doi:10.1109/LICS52264.2021.9470690

[34] David A. Naumann. 2020. Thirty-Seven Years of Relational Hoare Logic: Remarks on Its Principles and History. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12477)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 93–116. doi:10.1007/978-3-030-61470-6_7

[35] Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. doi:10.1145/3371078

[36] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL (2023), 1121–1151. doi:10.1145/3571232

[37] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 80–95. doi:10.1145/3453483.3454031

[38] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 909–936. doi:10.1007/978-3-662-54434-1_34

[39] Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem.. In *SAS (Lecture Notes in Computer Science, Vol. 3672)*, Chris Hankin and Igor Siveroni (Eds.). Springer, 352–367. http://dblp.uni-trier.de/db/conf/sas/sas2005.html#TerauchiA05

[40] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28.

[41] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 377–390. doi:10.1145/2500365.2500600

[42] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 343–356. doi:10.1145/2429069.2429111

[43] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 100–115. doi:10.1145/3497775.3503689

[44] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.* 2, POPL (2018), 56:1–56:29. doi:10.1145/3158144

[45] Shushu Wu, Xiwei Wu, and Qinxiang Cao. 2025. Encode the ∀∃ Relational Hoare Logic into Standard Hoare Logic: Extended Version. arXiv:2504.17444 [cs.PL] https://arxiv.org/abs/2504.17444

[46] Shushu Wu, Xiwei Wu, and Qinxiang Cao. 2025. EncRelTheory: EncRelTheory and Case Studies. doi:10.5281/zenodo.16927168

[47] Xiwei Wu, Yueyang Feng, Xiaoyang Lu, Tianchuan Lin, Kan Liu, Zhiyi Wang, Shushu Wu, Lihan Xie, Chengxi Yang, Hongyi Zhong, Naijun Zhan, Zhenjiang Hu, and Qinxiang Cao. 2025. QCP: A Practical Separation Logic-based C Program Verification Tool. arXiv:2505.12878 [cs.PL] https://arxiv.org/abs/2505.12878

[48] Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334. doi:10.1016/J.TCS.2006.12.036

Received 2025-03-26; accepted 2025-08-12