Journal of Automated Reasoning (2024) 68:18
https://doi.org/10.1007/s10817-024-09706-5

Verifying Programs with Logic and Extended Proof Rules: Deep Embedding vs. Shallow Embedding

Zhongye Wang[1] · Qinxiang Cao[1] · Yichen Tao[1]

Received: 7 May 2023 / Accepted: 9 July 2024 / Published online: 10 August 2024

Abstract
Many foundational program verification tools have been developed to build machine-checked program correctness proofs, a majority of which are based on Hoare logic. Their program logics, their assertion languages, and their underlying programming languages can be formalized by either a shallow embedding or a deep embedding. Tools like early versions of Verified Software Toolchain (before 2018) choose different shallow embeddings to formalize their program logic. But the pros and cons of these different embeddings were not yet well studied. Therefore, we want to study the impact of the program logic's embedding on logic's proof rules in this paper. This paper considers a set of useful extended proof rules, which aided the proof automation in VST, and four different logic embeddings: one deep embedding and three common shallow embeddings. We prove the validity of these extended rules under these embeddings and discuss their main challenges. Furthermore, we propose a method to lift existing shallowly embedded logics to deeply embedded ones to greatly simplify proofs of extended rules in VST. We implemented our theory in VST by lifting the originally shallowly embedded VST to our deeply embedded VST and establishing these extended rules.

Keywords Hoare logic · Program verification · Deep embedding · Shallow embedding · Machine-checked proofs · Separation logic

1 Introduction

Computer scientists have gained great success in developing foundationally sound program verification systems in the past decade. The word *foundationally* means: not only are programs' correctness properties verified, but the correctness proofs and the dependent proof rules are also verified and machine-checked in a proof assistant like Coq [4] or Isabelle [33, 35]. Foundational verification tools like Verified Software Toolchain (VST) [2, 9], Iris

✉ Zhongye Wang
wangzhongye1110@alumni.sjtu.edu.cn

✉ Qinxiang Cao
caoqinxiang@gmail.com

Yichen Tao
taoyc0904@alumni.sjtu.edu.cn

1 Shanghai Jiao Tong University, Shanghai, China

[20], CSimpl [38], CakeML [18], etc. enable their users to verify programs written in real programming languages like C, Rust and OCaml using Hoare-style logic.

Extended Proof Rules

Theoretically, a Hoare logic with compositional rules, the consequence rule, and proof rules for singleton commands (like assignments) is powerful enough to prove all valid judgements [14]. These rules are often referred to as primary rules. Figure 1A shows a sound and complete logic for a toy language with only the skip command, the assignment, the sequential composition, and the if-statement.

But in practice [9], a verification tool can be much easier to use if more rules are added to aid our proofs. Figure 1B shows two extended proof rules for the toy language. SEQ- ASSOC changes the associativity of sequential composition and allows provers to verify a program both from the beginning to the end in the forward style and from the end to the beginning in the backward style; IF- SEQ distributes the command c_3 after an if-block to two branches inside it, and in this way, instead of finding an intermediate assertion that merges post-conditions of two branches c_1 and c_2, provers can directly verify $c_1 ; c_3$ and $c_2 ; c_3$, with no obligation to find a common post-condition for two branches. In this paper, we also incorporate control flow reasoning in program logics, which allows realistic program verification. This induces more extended rules with increased complexity, which are also covered in this paper.

We make a distinction between extended rules and derived rules. Derived rules are directly derived from primary rules with mostly trivial proofs, and these proofs are the same across different embedding of a logic with the same primary rules. Figure 1C shows an example of derived proof rules. HOARE- CONSEQ- PRE is easily derivable from HOARE- CONSEQ and it allows weakening the precondition but keeps the postcondition untouched. This paper focuses on extended rules, whose proofs are usually non-trivial and differ across different

A. Primary Rules.

HOARE-SKIP
$$\vdash \{P\}\texttt{skip}\{P\}$$

HOARE-ASSIGN
$$\vdash \{P[x \mapsto e]\}x = e\{P\}$$

HOARE-SEQ
$$\frac{\vdash \{P\}c_1\{R\} \qquad \vdash \{R\}c_2\{Q\}}{\vdash \{P\}c_1 ; c_2\{Q\}}$$

HOARE-IF
$$\frac{\begin{array}{c}\vdash \{P \land [\![b]\!] = \text{true}\}c_1\{Q\} \\ \vdash \{P \land [\![b]\!] = \text{false}\}c_2\{Q\}\end{array}}{\vdash \{P\}\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\{Q\}}$$

HOARE-CONSEQ
$$\frac{P \vdash P' \qquad \vdash \{P'\}c\{Q'\} \qquad Q' \vdash Q}{\vdash \{P\}c\{Q\}}$$

B. Extended Rules.

SEQ-ASSOC
$$\frac{\vdash \{P\}(c_1 ; c_2) ; c_3\{Q\}}{\vdash \{P\}c_1 ; (c_2 ; c_3)\{Q\}}$$

IF-SEQ
$$\frac{\vdash \{P\}\texttt{if } b \texttt{ then } c_1 ; c_3 \texttt{ else } c_2 ; c_3\{Q\}}{\vdash \{P\}(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) ; c_3\{Q\}}$$

C. Derived Rules.

HOARE-CONSEQ-PRE
$$\frac{P \vdash P' \qquad \vdash \{P'\}c\{Q\}}{\vdash \{P\}c\{Q\}}$$

Fig. 1 An example Hoare logic for a toy language: primary rules and extended rules

embeddings (ways a program logic is formalized). Both derived rules and extended rules are useful in practice and we admit both as necessary components of VST.

Extended Rules and Hoare Logic Embeddings.
To utilize extended rules in program verifications, developers should first prove them in their logics. However, these extended rules may not be easily provable in every foundational verification tool, because different tools may choose various ways, i.e., various embeddings, to formalize their program languages and program logics to obtain unique features. A deeply embedded program logic defines the logic by inductive proof trees, while a shallowly embedded one directly defines Hoare triple's validity using a program's semantics. We further clarify the notion of embeddings in Sects. 3 and 4.

Due to differences in embedding methods, not all these extended rules are valid under each embedding, and challenges we might encounter in their proofs also vary from embedding to embedding. For example, shallowly embedded VST [2, 9] does not provide SEQ- INV (in Fig. 3) due to its significant proof burden while we can develop a simple formal proof of it in our deeply embedded VST. Therefore, in this paper, we (1) *present four mainstream Hoare logic embeddings* (one deep embedding, and three shallow embeddings respectively based on the big-step semantics, the weakest precondition, and the continuation) based on our survey of Hoare-logic-based program verification projects like VST, (2) *identify a set of extended proof rules* that can benefit verification automation in VST; and (3) *formally verify these extended proof rules under these different embeddings*.

Another Contribution: A Lifting Approach
We summarize the main challenges in proofs of these extended rules and find that most proofs of extended proof rules in shallowly embedded program logics are more complicated than those in the deeply embedded one. We claim that it would take more proof effort to equip some verification tools using shallow embeddings (like VST) with these extended proof rules. *Another contribution of our paper is to present a much easier way to extend shallowly embedded program logic with extended proof rules:* we can first lift the shallowly embedded logic into a deeply embedded one with acceptable proof effort. Then, we may derive extended proof rules, which are difficult for the original shallow embedding, using simpler proofs under the deep embedding.

We may not be the first ones to discover and utilize such an approach, but to our knowledge, it has never been formally presented in literature and we believe it can help with the design of foundational verification tools. To evaluate the lifting approach, we lift the shallowly embedded VST [2, 9], a verification framework for C programs, to the deeply embedded VST which supports most extended rules and is the basis of the current VST.

Our survey and formalization techniques can benefit future verification tools in the following ways. Firstly, our surveys and proofs indicate the level of support for extended proof rules under differently embedded program logics. Knowing which extended proof rules a verification tool supports, verification tool users may decide in advance which tool best matches their proof goals. By understanding the advantages and limitations of different embedding methods, verification tool developers can design proof systems satisfying the growing industrial demands. Secondly, our lifting approach can help existing verification tools to improve their framework by implementing extended rules with alleviated proof burden, as we demonstrate in our deeply embedded VST.

Paper Structure
Firstly, in Sect. 2, we fix the primitives of the programming language and primary proof

rules of the program logic that this paper focuses on. Based on this logic, we present three categories of extended proof rules and explain how they benefit verification tools.

We then clarify the concept of deep/shallow embeddings for programming languages, assertion languages, and program logics, along with the settings of this paper in Sect. 3, since they will affect the correctness of extended proof rules. In Sect. 4, we formally define a deep embedding and three shallow embeddings of the Hoare logic supporting control flow reasoning. We also briefly review existing Hoare-logic-based verification frameworks using these embeddings.

Then, in Sect. 5, we present formal proofs of these extended rules under each embedding method and discuss their main challenges. In Sect. 6, we present our approach to lifting a shallowly embedded program logic into a deeply embedded one to avoid otherwise challenging proofs of extended rules. This approach relies on careful choices (explained in Sect. 7) of primary rules.

After that, we discuss various extensions to the Hoare logic and its embedding in Sect. 8. They include separation logic, procedure calls, total correctness, non-determinism, and impredicative assertions.

In Sect. 9, we apply our lifting method to shallowly embedded VST to obtain our deeply embedded VST.

Lastly, we discuss related works of program logic embeddings in Sect. 10 and conclude the paper in Sect. 11.

We formalize all shallow embeddings in Sect. 4 and related proofs in Sect. 5 in a Coq repository [15]. We refer readers to our deeply embedded VST (Sect. 9) for formalizations of the deep embedding and related proofs. Our deeply embedded VST are integrated into the VST repository (https://github.com/PrincetonUniversity/VST).

2 Program Logic and Extended Rules

A verification tool always comes with some very basic primary rules, like compositional rules, the consequence rule, and singleton command rules. However, we could further enrich its capability by adding extended proof rules, some of which will be of great assistance to users for proof simplification and automation.

In this section, we first present in Sect. 2.1 a toy language and a set of primary rules for the program logic that we will use for demonstration throughout this paper. Then, based on this program logic, we introduce in the remainder of this section three categories of useful extended proof rules: transformation rules (Sect. 2.2), structural rules (Sect. 2.3), and inversion rules (Sect. 2.4). We demonstrate some representative rules in each category (Fig. 3) and show their potential usage.

2.1 The Toy Language and the Program Logic

The While-CF language (*While*-language with *c*ontrol *f*low commands) in Fig. 2 includes the assignment statement, $x = e$, which assigns the value of e to variable x, and empty command `skip` which does nothing. It includes three basic structural commands: the sequential composition $c_1 ; c_2$, the if-statement `if` e `then` c_1 `else` c_2, and the for-loop statement `for`(;;c_2) c_1 in C language style, where c_1 is the loop body and c_2 is the increment step after the execution of each loop iteration. This for-loop statement itself does not exit the loop without the help of `break` and `continue` commands, which are responsible for manipulating control flows

While-CF Language

$$\begin{array}{rl} & x, y \in \text{program-variable} \quad a \in \text{logic-variable} \\ & v \in \text{value} \quad b \in \text{bool expression} \quad e \in \text{expression} \quad \sigma \in \text{state} \\ c \in \text{command} := & \texttt{skip} \mid x = e \mid c_1 \,;\, c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \\ \mid & \texttt{for}(;;\, c_2)\ c_1 \mid \texttt{break} \mid \texttt{continue} \end{array}$$

Assertion Language

$$\begin{array}{rl} P, Q, R \in \text{assertion} := & \top \mid \bot \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \to Q \\ \mid & \forall a.P \mid \exists a.P \mid [\![e]\!] = v \mid P[x \mapsto e] \mid \cdots \end{array}$$

Primary Proof Rules

HOARE-SKIP
$$\vdash \{P\}\texttt{skip}\{P, [\bot, \bot]\}$$

HOARE-BREAK
$$\vdash \{P\}\texttt{break}\{\bot, [P, \bot]\}$$

HOARE-CONTINUE
$$\vdash \{P\}\texttt{continue}\{\bot, [\bot, P]\}$$

HOARE-SEQ
$$\frac{\vdash \{P\}c_1\{Q', [\vec{R}]\} \qquad \vdash \{Q'\}c_2\{Q, [\vec{R}]\}}{\vdash \{P\}c_1 \,;\, c_2\{Q, [\vec{R}]\}}$$

HOARE-LOOP
$$\frac{\vdash \{P\}c_1\{I, [Q, I]\} \qquad \vdash \{I\}c_2\{P, [Q, \bot]\}}{\vdash \{P\}\texttt{for}(;;\, c_2)\ c_1\{Q, [\bot, \bot]\}}$$

HOARE-IF
$$\frac{\vdash \{P \wedge [\![b]\!] = \texttt{true}\}c_1\{Q, [\vec{R}]\} \qquad \vdash \{P \wedge [\![b]\!] = \texttt{false}\}c_2\{Q, [\vec{R}]\}}{\vdash \{P\}\texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2\{Q, [\vec{R}]\}}$$

HOARE-ASSIGN
$$\vdash \{P[x \mapsto e]\}x = e\{P, [\bot, \bot]\}$$

HOARE-CONSEQUENCE
$$\frac{\begin{array}{c} \vdash \{P'\}c\{Q', [R'_{\text{brk}}, R'_{\text{con}}]\} \\ P \vdash P' \qquad Q' \vdash Q \qquad R'_{\text{brk}} \vdash R_{\text{brk}} \qquad R'_{\text{con}} \vdash R_{\text{con}} \end{array}}{\vdash \{P\}c\{Q, [R_{\text{brk}}, R_{\text{con}}]\}}$$

Fig. 2 The While-CF programming language and primary proof rules

in the loop. Although the language is named While-CF, we use the for-loop instead of the while-loop because the for-loop will yield more interesting extended proof rules, and it is also the choice of the VST. It is also non-trivial because a continue command in the for-loop body c_1 does not skip the increment step c_2, which cannot be easily encoded by a while-loop.

Figure 2 shows some terms of the assertion language. It includes basic first-order logic terms and terms to support express evaluation and variable assignment. The term $[\![e]\!]$ denotes the value of e evaluated on given state σ, i.e., eval(e, σ), and $[\![e]\!] = v$ means expression e evaluates to value v. $P[x \mapsto e]$ denotes the assertion after substitution of all occurrences of x by the value of e evaluated on the current state. We use the same notation for the boolean expression b.

Primary proof rules are listed in Fig. 2. Intuitively and informally, $\vdash \{P\}c\{Q, [R_{\text{brk}}, R_{\text{con}}]\}$ (we use $[\vec{R}]$ as syntax sugar for $[R_{\text{brk}}, R_{\text{con}}]$ in the rest of the paper) means that: starting from any program state satisfying pre-condition P,

- *(safety)* the execution of the command c never cause an error,
- *(correctness)* and if the execution of c terminates and its termination is caused by `break` or `continue`, the post-state will satisfy corresponding control flow post-conditions R_{brk}

and R_{con}, respectively. Otherwise, if the program terminates naturally, the post-state will satisfy the normal post-condition Q.

The HOARE- SKIP, HOARE- BREAK rule, and HOARE- CONTINUE are rules for singleton control flow commands that introduce the pre-condition into corresponding control flow post-conditions. The singleton command rule HOARE- ASSIGN.[1] asserts that the value of variable x used in the pre-condition will be changed in the post-condition. The HOARE- SEQ rule splits the proof of two sequentially composed commands. The HOARE- LOOP rule uses two triples of the loop body and the increment step to build a specification for the for-loop statement. The HOARE- IF rule splits the proof of the if-statement into two branches to prove them separately. The HOARE- CONSEQUENCE rule allows provers to strengthen the pre-condition and weaken the post-condition.

Although this programming language is a simple toy example, we may extend it with heap manipulation commands and function invocations as the paper will discuss in Sect. 8. Other primary rules related to separation logic and function invocation can also be included. But to demonstrate our results about extended rules in Sects. 2 and 5, the language and proof rules in Fig. 2 are sufficient. These primary rules are expressive enough to reason about loop control flows and also present in real verification tools like VST [2].

In Fig. 2, the assertion language and the program logic are presented in a deep embedding style using the syntax of assertions and derivation rules, but it is only for the purpose of demonstration. However, both of them can also be defined using shallow embeddings directly using their interpretation and semantics. We will soon investigate different embeddings of them in Sect. 3.

There are lots of useful extended rules that can be implemented above the program logic in Fig. 2, and we organize them into three categories and present some representatives of them in Fig. 3. We explain in detail the usage of these extended rules in the following sections.

2.2 Transformation Rules

It is often the case that we know two different programs are semantically equivalent and we should be able to substitute one of them for the other in a Hoare triple while preserving the triple's validity. Rules like IF- SEQ, LOOP- NOCONTINUE, and LOOP- UNROLL1[2] allow such semantic preserving transformations, and provers can transform the program into one that is easier to verify and sometimes automate its proof. *We classify these extended rules that transform the command in a Hoare triple as transformation rules.*

The proof rule IF- SEQ. The program $(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ ;\ c_3$ and $\texttt{if}\ b\ \texttt{then}\ c_1\ ;\ c_3\ \texttt{else}\ c_2\ ;\ c_3$ behaves similarly and we can use IF- SEQ to transform the one into the other.

Example 1 For a concise demonstration, we first start with a toy example (for which exists another workaround instead of using IF- SEQ and we will discuss it soon), where we want to prove the following for some given m, e, c_2, Q, $\vec{R}$,

[1] Here we use the backward version of HOARE- ASSIGN There is another forward version of HOARE- ASSIGN, $\vdash \{P\}x = e\{\exists v.\ P[x \mapsto v] \wedge [\![x]\!] = [\![e[x \mapsto v]]\!]\}$, which is often used in forward symbolic execution. Both versions are equivalent with the presence of the HOARE- CONSEQUENCE rule. We use both versions in this paper without notice.

[2] As far as we know, Andrew W. Appel and the development team of VST have integrated these transformation rules into VST to improve proof automation since 2017.

Extended Rules — Transformation Rules

$$\text{IF-SEQ}\ \frac{\vdash \{P\}\texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3\{Q,[\vec{R}]\}}{\vdash \{P\}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;c_3\{Q,[\vec{R}]\}}$$

$$\overset{\text{LOOP-NOCONTINUE}}{\frac{c_1, c_2 \text{ contain no } \texttt{continue} \qquad \vdash \{P\}\texttt{for}(;;\texttt{skip})\ (c_1\,;c_2)\{Q,[\vec{R}]\}}{\vdash \{P\}\texttt{for}(;;c_2)\ c_1\{Q,[\vec{R}]\}}}$$

$$\text{LOOP-UNROLL1}\ \frac{\begin{array}{c}\vdash \{P\}c_1\{P_1,[R_{\text{brk}},P_1]\}\\ \vdash \{P_1\}c_2\{P_2,[R_{\text{brk}},P_2]\} \qquad \vdash \{P_2\}\texttt{for}(;;c_2)\ c_1\{Q,[\vec{R}]\}\end{array}}{\vdash \{P\}\texttt{for}(;;c_2)\ c_1\{Q,[\vec{R}]\}}$$

Extended Rules — Structural Rules

$$\overset{\text{HOARE-EX}}{\frac{\text{forall } x.\ \vdash \{P(x)\}c\{Q,[\vec{R}]\}}{\vdash \{\exists x.\,P(x)\}c\{Q,[\vec{R}]\}}} \qquad \text{NOCONTINUE}\ \frac{\begin{array}{c}c \text{ contains no } \texttt{continue}\\ \vdash \{P\}c\{Q,[R_{\text{brk}},R_{\text{con}}]\}\end{array}}{\vdash \{P\}c\{Q,[R_{\text{brk}},R'_{\text{con}}]\}}$$

Extended Rules — Inversion Rules

$$\overset{\text{SEQ-INV}}{\frac{\vdash \{P\}c_1\,;c_2\{Q,[\vec{R}]\}}{\begin{array}{c}\text{exists } Q'.\ \vdash \{P\}c_1\{Q',[\vec{R}]\}\\ \text{and } \vdash \{Q'\}c_2\{Q,[\vec{R}]\}\end{array}}} \qquad \overset{\text{LOOP-INV}}{\frac{\vdash \{P\}\texttt{for}(;;c_2)\ c_1\{Q,[\bot,\bot]\}}{\begin{array}{c}\text{exists } I_1,I_2.\ \vdash \{I_1\}c_1\{I_2,[Q,I_2]\}\\ \text{and } \vdash \{I_2\}c_2\{I_1,[Q,\bot]\} \text{ and } P \vdash I_1\end{array}}}$$

$$\text{IF-INV}\ \frac{\vdash \{P\}\texttt{if}\,e\,\texttt{then}\,c_1\,\texttt{else}\,c_2\{Q,[\vec{R}]\}}{\vdash \{P \wedge [\![e]\!] = \texttt{true}\}c_1\{Q,[\vec{R}]\} \text{ and } \vdash \{P \wedge [\![e]\!] = \texttt{false}\}c_2\{Q,[\vec{R}]\}}$$

Fig. 3 Three categories of extended rules and their representatives

$$\vdash \{\underbrace{\exists n.\,[\![x]\!] = n \times m \wedge [\![y]\!] = m}_{P}\}(\texttt{if}\ b\ \texttt{then break else}\ \underbrace{z = x/y}_{c_1})\,;c_2\{Q,[\vec{R}]\}$$

Regularly, we need to explicitly provide an intermediate assertion Q' and show[3]

$$\frac{\dfrac{\dfrac{\dfrac{\cdots}{P \wedge [\![b]\!] \vdash R_{\text{brk}}}}{\vdash \{P \wedge [\![b]\!]\}\texttt{break}\{Q',[\vec{R}]\}} \quad \dfrac{\cdots}{\vdash \{P \wedge \neg[\![b]\!]\}c_1\{Q',[\vec{R}]\}}}{\vdash \{P\}(\texttt{if}\ b\ \texttt{then break else}\ c_1)\{Q',[\vec{R}]\}} \qquad \dfrac{\cdots}{\vdash \{Q'\}c_2\{Q,[\vec{R}]\}}}{\vdash \{P\}(\texttt{if}\ b\ \texttt{then break else}\ c_1)\,;c_2\{Q,[\vec{R}]\}} \tag{1}$$

[3] We abbreviate assertions $[\![b]\!] = \text{true}$ as $[\![b]\!]$ and $[\![b]\!] = \text{false}$ as $\neg[\![b]\!]$ for the sake of space.

But when we move c_2 inside the if-statement by IF- SEQ, we only need to prove

$$\dfrac{\dfrac{\dfrac{\cdots}{P \wedge [\![b]\!] \vdash R_{\mathrm{brk}}}}{\vdash \{P \wedge [\![b]\!]\}\texttt{break}\,;c_2\{Q', [\vec{R}]\}} \quad \dfrac{\cdots}{\vdash \{P \wedge \neg[\![b]\!]\}c_1\,;c_2\{Q, [\vec{R}]\}}}{\dfrac{\vdash \{P\}\texttt{if}\ b\ \texttt{then}\ \texttt{break}\,;c_2\ \texttt{else}\ c_1\,;c_2\{Q, [\vec{R}]\}}{\vdash \{P\}(\texttt{if}\ b\ \texttt{then}\ \texttt{break}\ \texttt{else}\ c_1)\,;c_2\{Q, [\vec{R}]\}}} \tag{2}$$

As the reader may wonder, to prove the sequential composition $c_1; c_2$ in (2), we still need to prove the last two goals for c_1 and c_2 in (1). The difference here is that when verifying (2), we can symbolically execute c_1 along with c_2 from the beginning to the end. But in (1), we need to symbolically execute two branches to a unifying post-condition Q' before executing c_2, while symbolic executions of two branches might produce different free variables in their postconditions and it brings troubles. We explain this difficulty in detail in the following paragraphs.

Symbolic execution in theorem provers is a technique for proof automation. It simulates the execution of the program and applies the execution's effect to the pre-condition according to proof rules. VST's proof automation relies heavily on such a technique. For example, symbolic execution can automatically transform the proof goal of the false branch (sequentially composed with a skip command for demonstration) in (1) as (3) shows (arrows represent symbolic execution steps by labeled proof rules).

$$\begin{aligned}
&\{\exists n.\, [\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots\}z = x/y\,;\texttt{skip}\{Q', [\vec{R}]\}\\
\xrightarrow{\text{HOARE- EX}}\ &\{[\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots\}z = x/y\,;\texttt{skip}\{Q', [\vec{R}]\}\\
\xrightarrow[\text{HOARE- SEQ}]{\text{HOARE- ASSIGN}}\ &\{[\![z]\!] = n \wedge [\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots\}\texttt{skip}\{Q', [\vec{R}]\}\\
\xrightarrow[\text{HOARE- SKIP}]{}\ &([\![z]\!] = n \wedge [\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots) \vdash Q'
\end{aligned} \tag{3}$$

During the symbolic execution, it will apply HOARE- EX (demonstrated in Sect. 2.3) to eliminate quantifiers in the pre-condition and introduce the bounded variable as a free one into the context to reason about it. On the contrary, symbolic execution cannot automatically determine which free variables should be reverted back into the assertion as bounded ones in which order so that the following execution steps can proceed.

Back to the example, symbolic execution of c_1 in (3) will introduce n into the context, which is necessary for forwarding $z = x/y$. To achieve a unifying Q' in (1), we need to revert n back into Q' in the symbolic execution of c_1, since the other branch does not introduce n and it remains bounded. In other words, the Q' in (3) should become $\exists n.\, [\![z]\!] = n \wedge [\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots$, instead of the final assertion in (3). However, in (2), after we introduce n in the execution of c_1, we do not need to revert it back and when c_2 access z and n during its execution, we do not need to introduce n again. IF- SEQ reduces the complexity of symbolic execution in this case.

Remark As we have mentioned, the toy example can be resolved without applying IF- SEQ. As the proof scheme in (4) indicates, we can extract n in P before we start the proof of the sequential composition and in this way, there is no need to revert n back again when finding the post-conditions of if-branches.

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash \{\cdots\}\texttt{if}\ b\ \texttt{then break else}\ c_1\{\cdots\}} \qquad \dfrac{\cdots}{\vdash \{\cdots\}c_2\{\cdots\}}}{\vdash \{[\![x]\!] = n \times m \wedge [\![y]\!] = m\}(\texttt{if}\ b\ \texttt{then break else}\ c_1)\,;c_2\{Q, [\vec{R}]\}}}{\vdash \{\exists n. [\![x]\!] = n \times m \wedge [\![y]\!] = m\}(\texttt{if}\ b\ \texttt{then break else}\ c_1)\,;c_2\{Q, [\vec{R}]\}} \tag{4}$$

However, this does not mean the example is meaningless, because in many cases, the existential quantifier in the precondition cannot and should not be eliminated in advance. For example, the precondition below is an assertion to state the storage of a linked list in the heap. To prove the Hoare triple below for some $x, l, e, c_1', c_2, Q, \vec{R}$, we are unable to eliminate all quantifiers in listrep by HOARE- EX without a fixed length of list l. In practice, we should be able to prove the triple below without such knowledge and keep the listrep predicate folded until reaching a test-empty operation to the linked list, and then should we unfold the predicate once and extract quantifiers. IF- SEQ is the only option in these cases.

$$\begin{aligned}&\vdash \{\mathrm{listrep}(x, l)\}(\texttt{if}\ b\ \texttt{then break else}\ c_1')\,;c_2\{Q, [\vec{R}]\}\\ &\text{where } \mathrm{listrep}(x, l) \triangleq (l = \mathrm{nil} \wedge \texttt{emp})\\ &\qquad\qquad\qquad\qquad \vee (\exists y, v, l'.\, l = v{::}l' \wedge x \mapsto (v, y) * \mathrm{listrep}(y, l'))\end{aligned}$$

The proof rule LOOP- NOCONTINUE. Another exemplary transformation rule in Fig. 3 is LOOP- NOCONTINUE, which allows merging loop body and incremental step into one command when neither of them contains `continue`. In this way, users only need to provide one loop invariant for the loop body instead of two (one before the loop body and one before the increment step) when reasoning about a loop with no `continue`.

Example 2 This example combines LOOP- NOCONTINUE and IF- SEQ. The program divides x by y twice but separates two divisions in the incremental step and the loop body.

$$\vdash \underbrace{\left\{\exists n. \begin{array}{r}[\![x]\!] = m^{2n}\\ \wedge [\![y]\!] = m\end{array}\right\}}_{P} \texttt{for(;;}\underbrace{x = z/y}_{c_2}\texttt{)}\ \texttt{if}\ \underbrace{x > 1}_{e}\ \begin{array}{l}\texttt{then break}\\ \texttt{else}\ \underbrace{z = x/y}_{c_1}\end{array} \left\{\underbrace{[\![x]\!] = 1}_{Q}, [\bot]\right\}$$

The precondition P can serve as a loop invariant, but regularly, we still need to find another loop invariant after the loop body but before the incremental step. However, by the symbolic execution (5), it first removes the incremental step and with `skip` as the incremental step, it is obvious that another loop invariant is the same as P. Then, it can open the loop and prove that the loop body obeys the invariant. With the help of IF- SEQ, it can automatically reach two easily provable proof goals without manual assistance.

$$\begin{aligned}&\vdash \{P\}\texttt{for(;;}c_2\texttt{)}\ \texttt{if}\ b\ \texttt{then break else}\ c_1\{Q, [\bot]\}\\ \xrightarrow[\text{NOCONTINUE}]{\text{LOOP-}}\ &\vdash \{P\}\texttt{for(;;skip)}\ (\texttt{if}\ b\ \texttt{then break else}\ c_1)\,;c_2\{Q, [\bot]\}\\ \xrightarrow[\text{IF- SEQ}]{\text{HOARE- LOOP}}\ &\vdash \{P\}\texttt{if}\ b\ \texttt{then break}\,;c_2\ \texttt{else}\ (c_1\,;c_2)\{P, [P, Q]\}\\ \xrightarrow[\text{HOARE- BREAK}]{\text{HOARE- IF}}\ &P \wedge \neg[\![b]\!] \vdash Q \quad\text{and}\quad \vdash \{P\}c_1\,;c_2\{P, [P, Q]\}\end{aligned} \tag{5}$$

The proof rule LOOP- UNROLL1. One more transformation rule we want to mention here is the LOOP- UNROLL1 rule. It allows peeling the first iteration of a for-loop and proving it as a separate triple. This is especially useful in two cases: when the first iteration of a loop does something different from the remaining iterations, e.g., initialization of some data, and the loop invariant for the remaining iterations can be greatly simplified without the first iteration; and when the loop only runs constant number of iterations, it may be easier to simply unfold it into a sequence of loop bodies and use symbolic execution to prove them automatically.

Although the LOOP- UNROLL1 rule in Fig. 3 does not perfectly match our criterion for transformation rules, their essence is the same. The LOOP- UNROLL1 rule has its counterpart, the following WHILE- UNROLL1 for the while command, in the While language without control flow statements. It unrolls the first iteration of the while-loop into an if-statement.

$$\textsc{while- unroll1}\ \frac{\vdash \{P\}\texttt{if}\ b\ \texttt{then}\ c\,;\texttt{while}(b)\ c\ \texttt{else skip}\{Q\}}{\vdash \{P\}\texttt{while}(b)\ c\{Q\}}$$

If there are control flow commands like `break` or `continue`, we cannot simply transform the for-loop into an if-statement because these commands will jump out of the scope of the original loop and interfere with executions outside the loop. As a result, we need to split the if-statement in WHILE- UNROLL1 into several Hoare triples in LOOP- UNROLL1 and specify control flow post-conditions for the loop body and the increment command.

2.3 Structural Rules

We classify extended rules that transforms pre-/post-conditions in proof goals as structural rules. These rules allow provers to adjust pre-/post-conditions into forms that permit more organized proofs.

A typical case is the use of the primary rule HOARE- CONSEQUENCE[4]. For example, in the last step of symbolic execution (5) from the last section, we need to prove the following assumption for the then-branch in order to apply HOARE- IF to verify the if-statement, where direct application of HOARE- BREAK does not work since the pre-condition does not match the break post-condition P and other two post-conditions are not $\perp$. We will use HOARE- CONSEQUENCE to weaken the pre-condition into Q and strengthen the post-condition P into $\perp$ to match the form of HOARE- BREAK. In this way, we can directly apply HOARE- BREAK to prove the new triple.

$$\frac{P \wedge \neg[\![b]\!] \vdash Q \qquad \perp \vdash P \qquad \vdash \{Q\}\texttt{break}\{\perp, [\perp, Q]\}}{\vdash \{\underbrace{(\exists n.[\![x]\!] = m^{2n} \wedge [\![y]\!] = m)}_{P} \wedge [\![x]\!] \leq 1\}\texttt{break}\{P, [P, \underbrace{[\![x]\!] = 1}_{Q}]\}}$$

In this section, we mainly discuss two structural rules in Fig. 3, HOARE- EX and NOCONTINUE, which we consider as extensions of HOARE- CONSEQUENCE. When separation logic is taken into consideration in Sect. 8.1.1, we will encounter two more structural rules, FRAME and HYPO- FRAME, and we postpone their discussions until then.

The proof rule HOARE- EX. As we have already seen in Sect. 2.2, it is a common situation in that we need to prove a Hoare triple whose precondition is existentially quantified, e.g.,

[4] However, we do not formalize HOARE- CONSEQUENCE rule as an extended rule but as a primary rule due to historical reasons. It is part of the primary rules in Cook's soundness and completeness proof of the Hoare logic [14] and is admitted as a primary rule of any Hoare logic ever since.

loop invariants are usually existentially quantified.[5] If provers can eliminate those existential quantifiers and extract bounded variables and related pure facts in preconditions into the meta-logic context, they can then apply domain-specific theories to those extracted variables — they are not buried in assertions any longer. HOARE- EX enable us to perform such extraction.

Example 3 In the symbolic execution example (3) in Sect. 2.2 (rephrased below), the first step is applying HOARE- EX to remove the existential quantifier. To verify the assignment $z = x/y$, we know the result of z will be n and it must be a free variable so that we can perform substitution of z by n according to HOARE- ASSIGN.

$$
\begin{aligned}
&\{\exists n.\, [\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots\} z = x/y\, ; \texttt{skip}\{Q', [\vec{R}]\} \\
\xrightarrow{\text{HOARE- EX}}\ &\{[\![x]\!] = n \times m \wedge [\![y]\!] = m \wedge \cdots\} z = x/y\, ; \texttt{skip}\{Q', [\vec{R}]\} \\
\ldots
\end{aligned}
$$

HOARE- EX is different from and sometimes cannot be derived directly from HOARE-CONSEQUENCE. HOARE- CONSEQUENCE changes pre-/post-conditions by entailments of the assertion logic, while HOARE- EX is a direct entailment between two Hoare triples in the meta-logic.

The proof rule NOCONTINUE. This rule allows us to modify the continue assertion arbitrarily since the program will never exit by a continue. We can use it as an enhancement to the consequence rule when we need to change the continue assertion. We can use this proof rule to support derivations of loop-related extended rules. For example, as we will see later in Fig. 4, we can prove LOOP- NOCONTINUE using the NOCONTINUE and inversion rules, which is the proof taken by VST.

2.4 Inversion Rules

Compositional rules allow us to combine proofs for separate modules into one proof for a larger program specification. But in some cases, we would like to extract information for these modules from the complete specification for other purposes. *We classify extended rules that extract premises of a primary rule from its goal as inversion rules.*

Figure 3 shows two example inversion rules, the SEQ- INV and LOOP- INV. SEQ- INV is the reversed sequencing rule, where the triple about c_1 ; c_2 gives us the triples of c_1 and c_2. LOOP-INV is the reversed loop rule, which gives us the intermediate loop invariant. They are typical rules that reproduce the premises in compositional rules from the original conclusion. In the following sections, we discuss only the SEQ- INV rule because it is the most representative inversion rule and other ones have similar conclusions.

Example 4 With inversion rules, we can more easily reorganize proof trees in deeply embedded logic, and in our deeply embedded VST, they have been used extensively in proving other extended rules like transformation rules and structural rules. For example, in Fig. 4, the transformation rule LOOP- NOCONTINUE can be proven given SEQ- INV and LOOP- INV.

[5] This is often necessary because logical variables utilized in the proof of the loop body usually are not exposed to the proof outside the loop. These variables should not be introduced before the proof of loop body and we need existential quantifiers to introduce this variable in the loop invariant.

$$\begin{array}{cl}
c_1, c_2 \text{ has no } \texttt{continue} & \\
\vdash \{P\}\texttt{for}(;;\texttt{skip})\ c_1 ; c_2\{Q, [\vec{R}]\} & \\ \hline
& \textsc{loop-inv} \\
\vdash \{I_1\}\texttt{skip}\{P, [Q, \bot]\} \qquad \vdash \{P\}c_1 ; c_2\{I_1, [Q, P]\} & \\ \hline
& \textsc{skip-inv}\ \&\ \textsc{conseq} \\
\vdash \{P\}c_1 ; c_2\{P, [Q, P]\} & \\ \hline
& \textsc{seq-inv} \\
\vdash \{P\}c_1\{I_2, [Q, P]\} \qquad \vdash \{I_2\}c_2\{I_1, [Q, P]\} & \\ \hline
& \textsc{nocontinue} \\
\vdash \{P\}c_1\{I_2, [Q, P]\} \qquad \vdash \{I_2\}c_2\{I_1, [Q, \bot]\} & \\ \hline
& \textsc{hoare-loop} \\
\vdash \{P\}\texttt{for}(;;c_2)\ c_1\{Q, [\bot, \bot]\} & \\ \hline
& \textsc{conseq} \\
\vdash \{P\}\texttt{for}(;;c_2)\ c_1\{Q, [\vec{R}]\} &
\end{array}$$

Fig. 4 A derivation of LOOP- NOCONTINUE based on inversion rules and structural rules

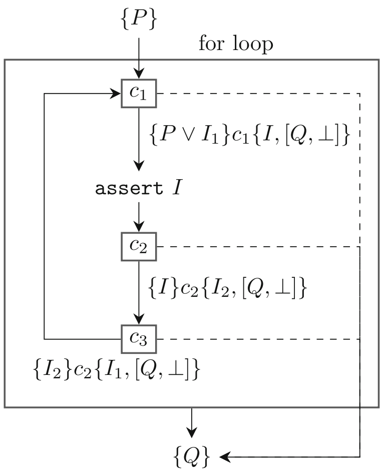
{P}
for loop
c_1
$\{P \vee I_1\}c_1\{I, [Q, \bot]\}$
assert I
c_2
$\{I\}c_2\{I_2, [Q, \bot]\}$
c_3
$\{I_2\}c_2\{I_1, [Q, \bot]\}$
{Q}

Fig. 5 Control flow graph of Example 5

We first extract information from the original triple by inversion rules and use NOCONTINUE to adjust the continue post-condition of the increment step c_2 to $\bot$ so that we can subsequently apply HOARE- LOOP to obtain the triple of the new loop.

Example 5 More crucially, inversion rules enable the extraction of premises from shallowly embedded triples without unfolding the definition of the triple and users can reorganize existing "proofs" to produce new triples. This is especially useful when one want to add more automation to annotation-based verification and is used in VST-A [49] to better utilize existing annotations.

For example, consider an annotated program $\texttt{for}(;;c_3)\ c_1; \texttt{assert}\ \ I; c_2$, where $\texttt{assert}\ \ I$ asserts that the assertion I holds between executions of c_1 and c_2 and it can guide prover's verification. Usually, the main burden of verifying a loop is to find the loop invariant I_1 and I_2 as indicated in the control flow graph in Fig. 5. But for this program, if we can check $\vdash \{P\}c_1\{I, [Q, \bot]\}$ and $\vdash \{I\}c_2; c_3; c_1\{I, [Q, \bot]\}$ to be true, then the following derivation directly proves the triple $\vdash \{P\}\texttt{for}(;;c_3)\ c_1; \texttt{assert}\ \ I; c_2\{Q, [\bot, \bot]\}$ for the loop.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \vdash \{P\}c_1\{I,[Q,\bot]\}
      \qquad
      \begin{array}{c}
      \dfrac{\vdash \{I\}c_2;\, c_3;\, c_1\{I,[Q,\bot]\}}{\vdash \{I\}c_2\{I_2,[Q,\bot]\} \qquad \vdash \{I_2\}c_3\{I_1,[Q,\bot]\}}\ \text{SEQ- INV} \\
      \vdash \{I_1\}c_1\{I,[Q,\bot]\}
      \end{array}
    }{
      \vdash \{P \vee I_1\}c_1\{I,[Q,\bot]\} \qquad \vdash \{I\}c_2\{I_2,[Q,\bot]\} \qquad \vdash \{I_2\}c_3\{I_1,[Q,\bot]\}
    }\ \text{DISJ}
  }{
    \vdash \{P \vee I_1\}c_1;\, \texttt{assert}\ I;\, c_2\{I_2,[Q,\bot]\} \qquad \vdash \{I_2\}c_3\{P \vee I_1,[Q,\bot]\}
  }\ \text{SEQ}
}{
  \dfrac{\vdash \{P \vee I_1\}\texttt{for}(;;c_3)\ c_1;\, \texttt{assert}\ I;\, c_2\{Q,[\bot,\bot]\}}{\vdash \{P\}\texttt{for}(;;c_3)\ c_1;\, \texttt{assert}\ I;\, c_2\{Q,[\bot,\bot]\}}\ \text{CONSEQ}
}\ \text{LOOP}
$$

With the help of SEQ- INV, two intermediate assertions I_1 and I_2 are extracted for free. By re-ordering commands into a new loop body $c_2; c_3; c_1$ using I as the invariant, $P \vee I_1$ and I_2 become our loop invariants and we reconstructed Hoare triples for three paths in Fig. 5. As a result, we have verified the triple of the for-loop without any additional proof. Notice that the triples we need to provide as the premise of this derivation can be verified symbolic execution without the need to manually construct intermediate assertions I_1 and I_2. This approach of using SEQ- INV to rearrange program orders helps simplify the proof of this kind of loop by making it easier to apply symbolic executions.

Summary
In summary, inversion rules have a one-to-one correspondence to primary rules. For each primary rule, we can derive an inversion rule that produces the premises of it from the conclusion. Users often apply inversion rules to assumptions to extract information from them and use this information to aid their proofs. On the contrary, transformation rules and structural rules do not have such correspondence and are applied directly to the proof goals (the Hoare triple specification) to generate sub-goals with easier proofs. Transformation rules change the program in the specification but keep original pre-/post-conditions, while structural rules only have effect on pre-/post-conditions but not the program.

Although the discussion of extended rules in this paper focuses on a programming language with control flow commands, most of these rules are still meaningful in a language that does not feature the control flow commands. For example, IF- SEQ and HOARE- EX do not depend on the control flow post-condition. LOOP- UNROLL1 also has a counterpart WHILE- UNROLL for loops without break and continue. For any programming language and program logic, it will always have its own set of inversion rules corresponding to its primary rules. Moreover, we believe adding other features to the language and the logic will yield more extended rules, but they are not well-studied and are not the focus of this paper.

3 Nomenclature

We have presented the program logic and extended rules in Sect. 2, but did not mention how they are formalized at all and we cannot check the correctness of these extended rules until we formally define the program logic. This section clarifies the notion of deep embedding and shallow embedding, which are two different approaches to formalizing languages and logics. In short, deep embeddings formalize structures while shallow embeddings formalize underlying semantics.

A program-logic-based foundational verification tool contains at least three elements where we could choose different embeddings: *the programming language*, *the assertion language* that describes program state properties, and *the program logic* that reasons about functional correctness of a program. Different verification projects choose different combi-

Table 1 Choices between shallow embedding and deep embedding

Verification projects (in historical order)	Programming language	Assertion	Program logic
CAP [46], XCAP [31]	Deep	Shallow	Deep
Software Foundations Vol. 2 [36]	Deep	Shallow	Shallow (BigS.) and Deep
FCSL[40]	Shallow	Shallow	Shallow (WP)
μC [45]	Deep	Deep	Deep
Simpl [39], CSimpl [38]	Deep	Shallow	Deep
VST (before Sep. 2018) [2]	Deep	Shallow	Shallow (Cont.)
VST (after Sep. 2018) [9]	Deep	Shallow	Deep
Iris [20]	Deep	Shallow	Shallow (WP)
Software Foundations Vol. 6 [12]	Deep	Shallow	Shallow (BigS.)

$$x : \text{variable\_name} \qquad \sigma : \text{prog\_state} \qquad \text{prog\_state} = \text{variable\_name} \to \mathbb{Z}$$

Deeply Embedded Program	Shallowly Embedded Program
$e \in \text{expr} := \text{Const}(n) \mid \text{Var}(v) \mid \text{Add}(e_1, e_2)$	$\text{expr} \triangleq \text{prog\_state} \to \mathbb{Z}$
$\text{eval}(\text{Const}(n), \sigma) = n$	$\text{Const} = \lambda n. \lambda \sigma. n$
$\text{eval}(\text{Var}(x), \sigma) = \sigma(x)$	$\text{Var} = \lambda x. \lambda \sigma. \sigma(x)$
$\text{eval}(\text{Add}(e_1, e_2), \sigma) = \text{eval}(e_1, \sigma) + \text{eval}(e_2, \sigma)$	$\text{Add} = \lambda e_1. \lambda e_2. \lambda \sigma. e_1(\sigma) + e_2(\sigma)$

Fig. 6 Example: deeply/shallowly embedded expressions

nations of embeddings for these three elements as Table 1 shows. For shallow embeddings of program logics, we could further divide them into sub-categories: big-step-based (BigS.), weakest-precondition-based (WP), and continuation-based (Cont.), which we will discuss in detail in Sect. 4.

In the rest of this section, we explain what are shallow/deep embeddings for a programming language, an assertion language, and a program logic.

3.1 Embeddings of Programming Languages

When formalizing a language (e.g. a programming language or an assertion language), a deep embedding formalizes its syntax tree first and defines its meaning separately. In contrast, a shallow embedding uses the language's intrinsic semantics as its definition directly. For example, Fig. 6 shows their differences in defining simple program expressions that only contain integer variables and addition. The addition operation is defined as a syntax tree constructor in the deeply embedded one and we use another evaluation function to define the semantics of all expressions. In the shallowly embedded one, the operation is directly defined as a function that computes the summation of its operands' evaluation results or a relation from the expression to the final result.

Most foundational verification tools including VST choose to use deeply embedded programming languages. The deeply embedded languages separate the syntax and semantics of programs and would allow structural induction over the program's syntax tree in some proofs. Comparably, typical shallowly embedded languages use functions from initial program states to ending states, or binary relations between initial states and ending states, to represent programs. In other words, programs are formalized in proof assistants by their denotations instead of syntax trees, which makes it difficult to equip them with structural inductions. It is also difficult (although not impossible [40]) for a shallowly embedded programming language to support concurrency and other extensions. For example, to implement concurrency, a deeply embedded programming language can syntactically parallel-compose two programs and then choose small-step semantics for describing concurrency since one can interleave parallel steps that programs take. However, it is less convenient for the denotational shallow embedding to implement such an interleaving. In the denotational shallow embedding, a program expression itself is a big-step denotation that hides the intermediate states of its execution, so we cannot directly define the program with other threads interleaving at these intermediate states.

In some cases, developers would also use mixed embeddings of their programming languages [40, 45]: these languages use deep embedding to formalize structural compositions and use shallow embedding for singleton commands. For example, in the μC framework [45] for verifying OS kernels, Xu et. al formalize atomic operations γ via a shallow embedding and formalize operation compositions (e.g., sequential composition $s_1; s_2$ and non-deterministic choice $s_1 + s_2$) via a deep embedding. Thus, a command s is defined by both embeddings in the formula below.

$$\begin{aligned} s &\in \text{Command}{:=}\gamma \mid s_1; s_2 \mid s_1 + s_2 \mid \textbf{end} \mid \cdots \\ \gamma &\in \text{Abstract State} \rightarrow \text{Abstract State} \rightarrow \text{Prop} \end{aligned}$$

Remark In a shallowly embedded language, transformation rules become trivial and meaningless. For example, in IF- SEQ rule, we have two programs defined below.

$$\begin{aligned} \texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3 &\triangleq \lambda\sigma.\,\mathsf{match\ eval}(b,\sigma)\ \mathsf{with} \mid \texttt{false} \Rightarrow c_3(c_1(\sigma)) \\ &\qquad\qquad\qquad\qquad\quad \mid \texttt{true} \Rightarrow c_3(c_2(\sigma)). \\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;c_3 &\triangleq \lambda\sigma.\,c_3(\mathsf{match\ eval}(b,\sigma)\ \mathsf{with} \mid \texttt{false} \Rightarrow c_1(\sigma) \\ &\qquad\qquad\qquad\qquad\quad \mid \texttt{true} \Rightarrow c_2(\sigma)). \end{aligned}$$

They are exactly the same function in the meta-language and will satisfy the same Hoare triple. Proofs of structural rules and inversion rules for a shallowly embedded language are similar to their proofs in a big-step-based shallowly embedded program logic (Sect. 4.1) since both uses big-step semantics to define the semantics. In this paper, we will mainly discuss deeply embedded programming languages.

3.2 Embeddings of Assertion Languages

To reason about a program's effect on program states, a Hoare-style program logic uses an assertion language to describe program state. A shallowly embedded assertion is a predicate in the meta-logic that directly defines the set of states. A deeply embedded assertion language specifies syntax trees of assertions first and then defines how to interpret assertions as sets of states. For example, Fig. 2 defines a deep embedding of an assertion language. Figure 7 defines a shallow embedding of the assertion language (only a snippet of it), which is also the interpretation for the deeply embedded one.

Unlike programming languages, we observe that shallowly embedded assertion languages are generally preferred by program verification framework developers, as Table 1 indicates that almost all existing works rely on a shallow assertion language. The reason is that the shallowly embedded assertion often has more expressiveness. For one fixed deeply embedded assertion language, we need to define its interpretation in the meta-logic. This interpretation is a meta-logic function from assertions' syntax trees to sets of program states, i.e.,

$$\begin{aligned} [\![e]\!] = v &:= (\lambda\sigma.\, \mathrm{eval}(e,\sigma) = v) & \top &:= \lambda\sigma.\, \mathrm{True} \\ P[x \mapsto e] &:= (\lambda\sigma.\, P[x/\mathrm{eval}(e,\sigma)]\,\sigma) & \bot &:= \lambda\sigma.\, \mathrm{False} \\ P \wedge Q &:= (\lambda\sigma.\, (P\,\sigma) \text{ and } (Q\,\sigma)) & &\cdots \end{aligned}$$

Fig. 7 A shallowly embedded assertion language: for clarity, we write $\sigma \vDash P$ for $P\ \sigma$ to denote that σ satisfies the assertion in the rest of the paper

$[\![\cdot]\!] \in \text{Assertion}_{\text{deep}} \to \text{state} \to \text{Prop}$. Thus, the expressiveness of this deep embedding is no stronger than the expressiveness of "sets of states", as which the assertion language is formalized in a shallow embedding, i.e., $\text{Assertion}_{\text{shallow}} \triangleq \text{state} \to \text{Prop}$.

The cause of different embedding choices for the programming language and the assertion language is the difference in the object they describe. Assertion languages describe static objects, i.e. an assertion is always used to determine a set of program states. Thus, it is suitable to directly model assertions as sets. Comparably, programming languages describe dynamic objects and transitions of program states. Verification projects for different real programming languages need to embody different features of those languages, e.g., concurrency. However, there exists no simple shallow embedding of a programming language that can describe these features concisely, e.g., we have mentioned in the previous part that denotational shallow embeddings are difficult to express concurrency. Therefore, formalizing different programming languages' syntax in deep embeddings is preferable to directly using shallow embeddings.

In this paper, we will stick to the shallow embedding of assertion languages defined in Fig. 7.

3.3 Embeddings of Program Logics

When formalizing logics[6] (e.g. the propositional logic or a Hoare logic), a shallowly embedded logic defines a statement to be *valid* directly using semantics. We mainly consider partial correctness in this paper, and one common definition of shallowly embedded Hoare triple is: $\vDash \{P\}c\{Q\}$ iff. for any initial state σ_1, if σ_1 satisfies assertion P, then the execution of c from σ_1 does not cause an error; and for and any ending state σ_2 reachable from σ_1 in the execution of c, σ_2 satisfies Q. Verification tool developers can prove many properties about valid triples and use them as "proof rules" to assist users in program verification. In practice, there are different ways to shallowly embed a program logic, which we discuss in Sects. 4.1, 4.2, 4.3.

In contrast, a deeply embedded logic formalizes the proof tree inductively by giving admitted proof rules, and we say a statement is *provable* under the logic, denoted by $\vdash \{P\}c\{Q\}$, iff. it can be constructed from these proof rules. Users of a verification tool can use these proof rules to build a proof tree of their program specifications.

To ensure proof rules given in a deeply embedded logic are consistent with program behaviors, one needs to additionally prove the logic sound, that is, every *provable* statement is also *valid*, i.e., for any statement S, if $\vdash S$ then $\vDash S$. In simple and common cases, this soundness theorem can be proved by induction over proof trees, i.e. it suffices to prove that every proof rule will always generate valid Hoare triples from valid triples. Using the sequential rule (HOARE- SEQ) as an example,

[6] In this paper, we mainly discuss the embedding of "program logic" and will use "logic" for short without ambiguity.

$$\text{HOARE- SEQ}\ \frac{\vdash \{P\}c_1\{Q\} \qquad \vdash \{Q\}c_2\{R\}}{\vdash \{P\}c_1; c_2\{R\}}$$

the induction step is to prove:

$$\vDash \{P\}c_1\{Q\} \text{ and } \vDash \{Q\}c_2\{R\} \text{ imply } \vDash \{P\}c_1; c_2\{R\} \tag{6}$$

In comparison, there is no counterpart of this soundness property when using a shallowly embedded logic. To use this sequential rule in a shallowly embedded logic, one needs to directly prove property (6). This fact implies that the implementation of a sound deeply embedded program logic always accompanies an underlying shallowly embedded one.

In some nontrivial cases, one has to introduce an auxiliary validity definition $\Vdash S$ in order to prove the soundness of logic in two steps: (1) for any triple S, prove $\vdash S$ implies $\Vdash S$ by induction over the proof tree; (2) show that $\Vdash S$ does imply $\vDash S$ by semantic analysis. For example, Brookes's concurrent separation logic soundness proof [7] uses this technique. In comparison, a shallow embedding strategy will directly formalize the auxiliary validity, and establish "proof rules" based on it. We discuss this soundness proof technique later in Sect. 8.1.2.

In conclusion, we can choose arbitrary combinations of shallow/deep embeddings among the programming language, the assertion language, and the program logic to instantiate a foundational verification tool. In this paper, we mainly focus on a deeply embedded programming language and a shallowly embedded assertion language, but consider different embeddings of program logics which we will discuss soon in Sect. 4.

4 Different Embeddings of Program Logics

As we have mentioned, there exists different ways to embed a program logic. In this section, we present three different shallow embeddings (Sects. 4.1, 4.2, 4.3) and a deep embedding (Sect. 4.4) to formalize the program logic in Fig. 2. Under each formalization, the chosen primary rules in Fig. 2 are sound. For each embedding method, we also demonstrate which existing Hoare-logic-based verification projects and how do their program logic fit into the category as Table 1 shows.

Meanwhile, there also exists other non-Hoare-logic-based verification approaches, which we discuss in Sect. 4.5.

Verification projects we review in this section all have certain unique features and complex mechanisms and it is difficult to cover all of them, e.g., support for concurrency. Some also do not support control flow reasoning and we do not discuss how to extend them with control flows to support our toy language. We only discuss some basics of their embeddings and related features that can classify them into each category.

4.1 Big-step (BigS) Based Shallow Embedding

Figure 8 describes notations defining the big-step semantics of the While-CF language, where $(c, \sigma_1) \Downarrow (\text{ek}, \sigma_2)$ states that from program state σ_1, the program c may terminate with exit kind ek, which could be normal exit, denoted by ϵ, or break, continue exit, and the state will be changed to σ_2. We use $(c, \sigma) \Uparrow$ to denote that an error would occur for the execution of c from state σ. The big-step semantics is defined recursively, e.g., the semantics of the

$$\text{ek} \in \text{exit\_kind} := \epsilon \mid \text{brk} \mid \text{con} \qquad \sigma \in \text{state}$$
$$\text{Big-step Relation: } (c, \sigma_1) \Downarrow (\text{ek}, \sigma_2) \qquad \text{Error: } (c, \sigma) \Uparrow$$

$$\textsc{Seq1}\ \frac{(c_1, \sigma_1) \Downarrow (\epsilon, \sigma_3) \quad (c_2, \sigma_3) \Downarrow (\text{ek}, \sigma_2)}{(c_1\,;c_2, \sigma_1) \Downarrow (\text{ek}, \sigma_2)} \qquad \textsc{Seq2}\ \frac{\text{ek} \neq \epsilon \quad (c_1, \sigma_1) \Downarrow (\text{ek}, \sigma_2)}{(c_1\,;c_2, \sigma_1) \Downarrow (\text{ek}, \sigma_2)} \qquad \textsc{Seq\_Fail1}\ \frac{(c_1, \sigma) \Uparrow}{(c_1\,;c_2, \sigma) \Uparrow}$$

$$\textsc{Seq\_Fail2}\ \frac{(c_1, \sigma_1) \Downarrow (\epsilon, \sigma_2) \qquad (c_1, \sigma_2) \Uparrow}{(c_1\,;c_2, \sigma_1) \Uparrow}$$

Fig. 8 Notations for Big-step semantics

sequencing command is Seq1 and Seq2. For the sake of space, we put the full definition (which is standard) in Appendix A.

Based on this big-step semantics, a (partial correctness) triple is defined to be valid, $\vDash_b \{P\}c\{Q, [R_{brk}, R_{con}]\}$, iff. for any σ_1 satisfying precondition P, (1) the execution of c from σ_1 is safe and does not cause error, and (2) if the execution terminates, the ending program state satisfies the corresponding post-condition.

$$\begin{aligned}\vDash_b \{P\}c\{Q, [R_{brk}, R_{con}]\} \text{ iff. } & \text{for all } \sigma_1 \vDash P, \neg\,(c, \sigma_1) \Uparrow \\ & \text{and for all ek}, \sigma_2, \text{if } (c, \sigma_1) \Downarrow (\text{ek}, \sigma_2) \\ & \quad \text{then ek} = \epsilon \text{ implies } \sigma_2 \vDash Q \\ & \quad \text{and ek} = \text{brk implies } \sigma_2 \vDash R_{brk} \\ & \quad \text{and ek} = \text{con implies } \sigma_2 \vDash R_{con}\end{aligned}$$

Related Projects

Klein et al. [8] formalizes an imperative functional programming language in Isabelle/HOL, which is a shallowly embedded one using the state monad in HOL. Based on this language, Lammich [24] builds a logic in big-step-based shallow embedding in Isabelle/HOL. The logic embedding is almost identical to the one discussed above, but it does not consider control flows and only requires the ending state of the normal exit to satisfy the post-condition. Thus, its definition does not have the last two conjuncts above. Based on Lammich's logic [24], Zhan [48] verifies imperative implementations of some data structures in Isabelle/HOL using its auto2 prover [47]. Nipkow's Hoare logic in Isabelle/HOL [32, 33] also uses big-step embedding. Many verification has been performed based on these logics in Isabelle, e.g., Lammich and Nipkow [25] proves the correctness of priority search tree and Prim's and Dijkstra's algorithm in Isabelle.

Cook's famous Hoare logic's soundness and completeness proof [14] uses big-step-based shallow embedding as the logic's validity definition and proves many properties including logic's soundness and completeness w.r.t. it.

Software foundation [36] is a famous textbook for teaching Coq formalization. Its second volume introduces Hoare logics both in the big-step-based embedding and the deep embedding, where the former is also the validity definition of the latter. Its sixth volume introduces a separation logic built with the big-step-based embedding. The simplicity of the big-step-based embedding makes it a good introduction to separation logic for beginners.

However, similar to the situation of shallowly embedded language (Sect. 3.1), the big-step semantics struggles to support complex semantic features like concurrency. Indeed, among

$$\kappa \in \text{continuation} := \epsilon \mid \text{KSeq}(c) \cdot \kappa \mid \text{KLoop}_1(c_1, c_2) \cdot \kappa \mid \text{KLoop}_2(c_1, c_2) \cdot \kappa$$

$$((\texttt{for}(;;\, c_2)\ c_1), \kappa, \sigma) \to_c (c_1\, ;\, \texttt{continue}, \text{KLoop}_1(c_1, c_2) \cdot \kappa, \sigma) \tag{1}$$

$$(\texttt{skip}, \text{KLoop}_2(c_1, c_2) \cdot \kappa, \sigma) \to_c (c_1\, ;\, \texttt{continue}, \text{KLoop}_1(c_1, c_2) \cdot \kappa, \sigma) \tag{2}$$

$$(\texttt{continue}, \text{KLoop}_1(c_1, c_2) \cdot \kappa, \sigma) \to_c (c_2, \text{KLoop}_2(c_1, c_2) \cdot \kappa, \sigma) \tag{3}$$

$$(\texttt{break}, \text{KLoop}_1(c_1, c_2) \cdot \kappa, \sigma) \to_c (\texttt{skip}, \kappa, \sigma) \tag{4}$$

$$(\texttt{continue}, \text{KSeq}(c) \cdot \kappa, \sigma) \to_c (\texttt{continue}, \kappa, \sigma) \tag{5}$$

$$(\texttt{break}, \text{KSeq}(c) \cdot \kappa, \sigma) \to_c (\texttt{break}, \kappa, \sigma) \tag{6}$$

Fig. 9 Notations for Small-step semantics

concurrent program verification projects we have surveyed, none of them is based on the big-step semantics.

4.2 Weakest Precondition (WP) Based Shallow Embedding

Another shallow embedding method is to embed the logic using the weakest precondition defined by small-step semantics. We want to emphasize that the weakest precondition here is directly defined by semantics instead of encoding by some existing Hoare logic.

We use $(c, \kappa, \sigma) \to_c (c', \kappa', \sigma')$ to describe a small-step in command reduction and $\to_c^*$ to describe the reflexive transitive closure of $\to_c$. The small-step reduction is a binary relation between triples of the focused term[7] c, the continuation (control stack) κ, and the program state σ. The control flow commands make it slightly different from textbook definitions. We mainly follow Comp-Cert Clight's definition style and list the most important semantic rules in Fig. 9.

(1) When the focused term begins with a for-loop, the loop is pushed into the control flow stack and the loop body (followed by `continue`) is loaded into the focused term.
(2) When the increment step for the previous iteration finishes, the loop body is loaded for the next iteration.[8]
(3) When the focused term reduces to `continue`, the increment step c_2 is loaded and the innermost loop's continuation is updated to Kloop$_2$.
(4) When the focused term reduces to `break`, the innermost loop is popped out of the continuation and the program is set to `skip` to continue execute the remaining control stack.
(5), (6) When the focused term reduces to `continue` or `break`, any KSeq continuation at the top of the continuation will be skipped. As a result, if an execution from (c, κ, σ) terminates, it must terminate at $(\texttt{skip}, \epsilon, \_)$, $(\texttt{break}, \epsilon, \_)$, or $(\texttt{continue}, \epsilon, \_)$.

The shallowly embedded weakest pre-condition can be defined on this small-step semantics. We use $\sigma \models \mathsf{WP}(c, \kappa)\{Q, [\vec{R}]\}$ to denote that a program state σ satisfies the weakest

[7] The focused term is the next command to execute and is so named in some literature.

[8] $\text{KLoop}_1(c_1, c_2)$ means in-loop-body, and $\text{KLoop}_2(c_1, c_2)$ means in-increment-step.

pre-condition for the focused program c and the continuation κ with post-conditions Q and $[\vec{R}]$. The WP is defined as the largest set such that $\sigma \models \mathsf{WP}(c, \kappa)\{Q, [\vec{R}]\}$ iff.

- *Terminal Case:* κ is ϵ and (1) $c = \texttt{skip}$ and $\sigma \models Q$ or (2) $c = \texttt{break}$ and $\sigma \models R_{\text{brk}}$ or (3) $c = \texttt{continue}$ and $\sigma \models R_{\text{con}}$;
- *Preservation Case:* Or there exists some c', κ', σ' that (c, κ, σ) can be further reduced to, i.e. $(c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma')$, and for any c', κ', σ' such that $(c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma')$, the judgement $\sigma' \models \mathsf{WP}(c', \kappa')\{Q, [\vec{R}]\}$ holds.

The terminal case ensures that the ending program state will satisfy post-conditions, and the preservation case guarantees that it can always step forward and eventually reach an ending state satisfying post-conditions.

A triple is defined to be valid under the weakest-precondition-based shallow embedding, $\models_w \{P\}c\{Q\}$, iff. for any state σ satisfying precondition P, we have $\sigma \models \mathsf{WP}(c, \epsilon)\{Q, [\vec{R}]\}$.

Define WP Using Bourbaki–Witt Fixed Point Thereom

We may implement the above weakest-precondition's definition using the Bourbaki–Witt fixed point theorem [6, 44], which is used by many existing frameworks to define the weakest-precondition-based embedding and is used in our Coq formalization.

Theorem 1 (Bourbaki–Witt fixed point theorem) *If $(L, \leq)$ is a non-empty complete ordered chain, and $f : L \rightarrow L$ satisfies $f(x) \geq x$ forall x, then f has a fixed point. For some $x_0 \in L$, let $g : \mathbb{N} \rightarrow L$ be a function with*

$$g(0) = x_0 \qquad g(n+1) = f(g(n)),$$

then $\lim_{n \rightarrow \infty} g(n)$ *is a fixed point.*

Here, the type L is sets of program configuration, i.e., $\mathcal{P}(\text{state} \times \text{command} \times \text{continuation})$, and order $\leq$ is $\supseteq$. The function f is defined as below, which removes configurations that will either cause an error or contradict with the post-conditions from the input set x.

$$\begin{aligned} f(x) \triangleq x \cap & \left\{ (\sigma, c, \kappa) \;\middle|\; \begin{array}{r} ((\kappa = \epsilon \wedge c = \texttt{skip}) \Rightarrow \sigma \models Q) \\ \wedge((\kappa = \epsilon \wedge c = \texttt{break}) \Rightarrow \sigma \models Q_{\text{brk}}) \\ \wedge((\kappa = \epsilon \wedge c = \texttt{continue}) \Rightarrow \sigma \models Q_{\text{con}}) \end{array} \right\} \\ \cap & \left\{ (\sigma, c, \kappa) \;\middle|\; \text{reducible}(c, \kappa, \sigma) \wedge \forall \sigma', c', \kappa'. \begin{pmatrix} (c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma') \\ \Rightarrow (\sigma', c', \kappa') \in x \end{pmatrix} \right\} \end{aligned}$$

where

$$\text{reducible}(c, \kappa, \sigma) \text{ iff. } \exists c', \kappa', \sigma'. (c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma')$$

To utilize Theorem 1, we only need to find a function g satisfying these conditions and use its limit as the fixed point. We define the function g as

$$\lambda n. \lambda \sigma, c, \kappa. \forall m \leq n. \sigma \models \text{WP}(m, c, \kappa)\{Q, [\vec{R}]\}$$

where

$$\begin{aligned} \sigma \models \text{WP}(m, c, \kappa)\{Q, [\vec{R}]\} \text{ iff. } & \left(\kappa = \epsilon \wedge \begin{pmatrix} (c = \texttt{skip} \wedge \sigma \models Q) \vee \\ (c = \texttt{break} \wedge \sigma \models R_{\text{brk}}) \vee \\ (c = \texttt{continue} \wedge \sigma \models R_{\text{con}}) \end{pmatrix} \right) \\ \vee & \left(0 < m \Rightarrow \begin{pmatrix} \text{reducible}(c, \kappa, \sigma) \wedge \\ \forall c', \kappa', \sigma'. \begin{pmatrix} (c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma') \Rightarrow \\ \sigma' \models \text{WP}(m-1, c', \kappa')\{Q, [\vec{R}]\} \end{pmatrix} \end{pmatrix} \right) \end{aligned} \tag{7}$$

One may prove that $g(0) \in L$ and $g(n+1) = f(g(n))$ and define the weakest precondition by taking the limit of $g(n)$ as below.

$$\sigma \vDash \mathrm{WP}(c, \kappa)\{Q, [\vec{R}]\} \text{ iff. } \forall n \in \mathbb{N}.\, \sigma \vDash \mathrm{WP}(n, c, \kappa)\{Q, [\vec{R}]\}$$

If the reader is familiar with the step-indexing technique [3], they may find that a meta-logic equipped with step-indices (e.g., Iris [20]) can also define the weakest precondition as the Bourbaki–Witt fixed point. In a step-indexed world, one may add an index number to the interpretation of a predicate and use a later modality to lower the index by one.

$$\begin{aligned} \sigma \vDash P &\iff \forall n.\, \sigma \vDash P_n \\ \sigma \vDash (\triangleright P)_n &\iff n = 0 \vee \sigma \vDash P_{n-1} \end{aligned}$$

To use step indexing to define the weakest precondition, the number of execution steps is considered as the index. For example, the ordinal number n in $\sigma \vDash \mathrm{WP}(n, c, \kappa)\{Q, [\vec{R}]\}$ represents the number of execution steps during which this predicates guarantees the program to not produce an error. The weakest precondition's definition can also be simplified using these step-indexing notations.

$$\begin{aligned} \sigma \vDash \mathrm{WP}(c, \kappa)\{Q, [\vec{R}]\} \text{ iff. } & \left(\kappa = \epsilon \wedge \begin{pmatrix} (c = \texttt{skip} \wedge \sigma \vDash Q) \vee \\ (c = \texttt{break} \wedge \sigma \vDash R_{\text{brk}}) \vee \\ (c = \texttt{continue} \wedge \sigma \vDash R_{\text{con}}) \end{pmatrix} \right) \\ & \vee \left(\begin{array}{r} \text{reducible}(c, \kappa, \sigma) \wedge \\ \forall c', \kappa', \sigma'. \begin{pmatrix} (c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma') \\ \Rightarrow \sigma' \vDash \triangleright \mathrm{WP}(c', \kappa')\{Q, [\vec{R}]\} \end{pmatrix} \end{array} \right) \end{aligned}$$

Define WP Using Tarski Fixed Point Thereom

The other way to define the weakest-precondition is through the Tarski fixed point theorem [42].

Theorem 2 (Tarski fixed point theorem) *For a complete lattice* $(L, \leq)$ *and a monotone function* $f : L \rightarrow L$*, the set of all fixed points of* f *is also a complete lattice with* $\sup\{x \in L \mid x \leq f(x)\}$ *as the greatest fixed point.*

Here, the type L is still sets of program configuration, and the order $\leq$ is the subset relation $\subseteq$. The function $f(x)$ increases (thus it guarantees monotonicity) the input configuration set x with configurations that can become configurations in x after one step of execution, and if the input set is empty, it returns configurations already satisfying the post-conditions Q and $\vec{R}$.

$$\begin{aligned} f(x) \triangleq x \cup & \left\{ (\sigma, c, \epsilon) \;\middle|\; \begin{array}{r} (c = \texttt{skip} \wedge \sigma \vDash Q) \\ \vee (c = \texttt{break} \wedge \sigma \vDash R_{\text{brk}}) \\ \vee (c = \texttt{continue} \wedge \sigma \vDash R_{\text{con}}) \end{array} \right\} \\ \cup & \left\{ (\sigma, c, \kappa) \;\middle|\; \text{reducible}(c, \kappa, \sigma) \wedge \forall \sigma', c', \kappa'. \begin{pmatrix} (c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma') \\ \Rightarrow (\sigma', c', \kappa') \in x \end{pmatrix} \right\} \end{aligned}$$

One may simply applies the Tarski fixed point theorem by taking the supremum of the chain $\emptyset \leq f(\emptyset) \leq f(f(\emptyset)) \leq \cdots$ as the weakest-precondition definition.

This approach is equivalent to directly using a co-inductive definition in Coq. While we did not find any work that uses co-inductive definitions for weakest-precondition, Liang [26] and Xu et al. [45] uses co-inductively defined simulations when formalizing their relational Hoare logic. We also proved in our Coq formalization that co-inductively defined weakest-preconditions are equivalent to the one defined using the Bourbaki–Witt fixed point.

Related Projects
Fine-grained Concurrent Separation Logic (FCSL) [40] is a framework for verifying concurrent programs. It uses mixed embedding (deep and shallow embedding) for its programming language and a weakest-precondition-based shallow embedding for its program logic. Its program is defined by an action tree, where each edge, the atomic action, is defined by a transition between states, i.e., a shallow embedding. In the action tree's syntax definition below, ω is a leaf node indicating the divergence of a thread and ret v is a leaf node indicating the termination of a thread with return value v. The concatenation $a :: k$ of the action a and the context k is the sequential execution a and k with a's result as arguments. Similarly, parallel composition $(t_1 \| t_2)::k$ first interleaves executions of tree t_1 and t_2 and passes their results to the context k.

$$\begin{aligned} t, t_1, t_2 \in \text{Tree} &:= \omega \mid \text{ret } v \mid a::(k : \text{Val} \to \text{Tree}) \\ &\quad\mid\ (t_1 \| t_2)::(k : \text{Val} \times \text{Val} \to \text{Tree}) \\ a \in \text{Action} &:= \text{state} \to \text{state} \to \text{Prop} \end{aligned}$$

The definition below is FCSL's [40] embedding of their Hoare triple (we omit details about concurrency). The always predicate (8) is similar to our weakest precondition's definition, which asserts that the memory safety at each step and at each step the assertion P should hold for the program state and the action tree. This assertion of always predicate in the Hoare triple (10) ensures that the post-condition holds at leaf nodes. It also uses the Bourbaki–Witt fixed point approach to define the weakest precondition. The k in (8) and (9) serves the purpose of an ordinal number.

$$\begin{aligned} &\text{always}^k\ \sigma\ t\ Q \text{ iff. memsafe}(\sigma, t) \wedge Q(\sigma, t) \wedge \\ &\qquad \forall \sigma', t'.\, (k > 0 \wedge ((\sigma, t) \to (\sigma', t'))) \Rightarrow \text{always}^{k-1}\ \sigma'\ t'\ Q \qquad (8) \\ &\text{always}\ \sigma\ t\ Q \text{ iff. } \forall k \in \mathbb{N}.\, \text{always}^k\ \sigma\ t\ Q \qquad (9) \\ &\vDash \{P\}c\{Q\} \text{ iff. } \forall \sigma.\, \sigma \vDash P \Rightarrow \text{always}\ \sigma\ t\ (\lambda \sigma, t.\, \forall v.\, t = \text{Ret } v \Rightarrow Q(\sigma, v)) \qquad (10) \end{aligned}$$

Iris is a higher order concurrent separation logic for verifying the correctness of functional programs. Its framework is parameterized with a programming language and uses the weakest-precondition-based embedding. When concurrency is not involved (Sect. 6.3.2 of [20]), Iris embeds their logic by weakest pre-conditions below as a separation logic proposition in Iris (iProp).

$$\begin{aligned} \mathsf{WP}\, e\, \{\Phi\} \triangleq\ & (e \in \mathit{Val} \wedge \Phi(e)) \qquad (11) \\ & \vee \big(\forall \sigma.\, e \notin \mathit{Val} \wedge S(\sigma) -\!\!* \big(\text{reducible}(e, \sigma) \\ & \wedge \triangleright \forall e', \sigma'.\, \big((e, \sigma) \to_t (e', \sigma')\big) -\!\!* \big(S(\sigma') * \mathsf{WP}\, e'\, \{\Phi\}\big)\big)\big) \qquad (12) \end{aligned}$$

The first disjunct (11) asserts that if the expression e is a value, then the evaluation has terminated and the program state and the evaluation result e should satisfy Φ. The second disjunct (12) specifies behaviors when e is not a terminal and should be further reduced. $S(\sigma)$ injects program state σ into an iProp. The disjunct (12) will first consume a piece of memory injected by σ and asserts that expression e is reducible with this piece of memory (reducible(e, σ)). Then, for any new expressions e' and σ' it can reduce to by small-step semantics $\to_t$, we can still have the memory $S(\sigma')$ that it reduces to and the new expression

e' still preserves the weakest pre-condition. Iris's propositions iProp uses the step-index to solve the circularity when defining higher-ordered ghost states, which happens to be a suitable choice of the ordinal number in the definition of the co-inductive weakest precondition. Iris's Hoare triple[9] $\{P\}e\{\Phi\}$ is embedded as $P \vdash \mathsf{WP}\ e\ \{\Phi\}$, which conforms with our prototypical weakest-precondition-based embedding.

FCSL and Iris both support concurrent program verifications through mechanisms of angelic updates (view shifts in Iris). Iris is also extended to support prophecy variables [21]. However, these features are out of this paper's scope. For simplicity, definitions in this section only consider sequential programs and remove features supporting concurrency. For the complete definition, readers may refer to their original papers [20, 40].

4.3 Continuation (Cont.) based Shallow Embedding

Continuation-based shallow embedding defines the Hoare triple through Hoare tuple $\{P\}(c, k)$, pronounced P guards (c, k). An assertion P guards the execution of a program c and a continuation κ, $\{P\}(c, \kappa)$, iff. for any program state σ satisfying the pre-condition P, we have $\text{safe}(c, \kappa, \sigma)$, which is the largest set of configurations with following properties.

- *Terminal Case:* κ is ϵ, and c is `skip`, `break`, or `continue`;
- *Preservation Case:* Or (c, κ, σ) can be further reduced by $\rightarrow_c$, and for any (c', κ', σ') it reduces to, we should have $\text{safe}(c', \kappa', \sigma')$.

One could use either fixed point formalization in Sect. 4.2 for defining $\text{safe}(c, \kappa, \sigma)$.

A triple $\{P\}c\{Q, [\vec{R}]\}$ is valid iff. for arbitrary continuation κ,

$$\text{if} \quad \begin{cases} \{Q\}(\texttt{skip}, \kappa) \\ \{R_{\text{brk}}\}(\texttt{break}, \kappa) \\ \{R_{\text{con}}\}(\texttt{continue}, \kappa) \end{cases} \quad \text{then} \quad \{P\}(c, \kappa).$$

It states that for any continuation that safely executes from different post-conditions with corresponding exit kind, it should safely executes after program c's execution from the pre-condition.

Related Projects

The original shallowly embedded VST [2] uses a deeply embedded programming language and a program logic in continuation-based shallow embedding. The Hoare triples in shallowly embedded VST extend our definition with function invocations and concurrent separation logic. Besides these features, which we discuss later in Sect. 9, its embedding is almost identical to the prototypical definition above.

4.4 Deep Embeddings

As clarified in Sect. 3, a deep embedding of program logic is a proof system with inductively defined proof trees. Different deep embeddings of program logics with fixed languages and forms of judgements have different sets of admitted proof rules. For example, we may consider a proof system with primary proof rules from Fig. 2 as a deep embedding. In Coq, one can

[9] It is worth mentioning that Iris primarily provides mechanisms to directly reason about the weakest pre-conditions, which makes it less like a Hoare-logic-based framework, and our conclusions may not apply to their logic. We discuss the weakest-precondition-based verification method in Sect. 4.5.2.

define the deep embedding with an inductive relation shown below. Here, `provable P c Q Rb Rc` stands for the Hoare triple $\vdash \{P\}c\{Q, [R_b, R_c]\}$.

```
Inductive provable: Assertion → com → Assertion (* normal post *) →
 Assertion (* break post *) → Assertion (* continue post *) → Prop :=
| hoare_skip: forall P, provable P skip P bot bot
| hoare_break: forall P, provable P break bot P bot
| hoare_continue: forall P, provable P continue bot bot P
| hoare_seq: forall c1 c2 P P’ Q Rb Rc,
   provable P c1 P’ Rb Rc → provable P’ c2 Q Rb Rc →
   provable P (c1;c2) Q Rb Rc
| hoare_loop: forall c1 c2 P I Q,
   provability P c1 I Q I → provable I c2 P Q bot →
   provable P (for(;;c2) c1) Q bot bot
| ⋯.
```

An arbitrary set of admitted proof rules may not be correct, thus we need to prove the soundness of such a proof system. To do so, we need to define the validity of a logic judgement, i.e., the corresponding shallowly embedded logic. We can prove the soundness by showing that each proof rule is valid in the shallow embedding and the shallowly embedded logic is sound. We can easily prove primary proof rules in Fig. 2 to be valid under all three shallow embeddings in Sect. 4.1, 4.2, 4.3.

Related Projects

Simpl [39] is a tool in Isabelle/HOL for verifying sequential programs, which has been used to verify seL4 code [22]. It uses a deeply embedded programming language and program logic. Its judgement is $\vdash \{P\}c\{Q, Q_{\mathrm{ek}}\}$ with the following validity definition.

$$\begin{aligned}\vDash \{P\}c\{Q, Q_{\mathrm{ek}}\} \triangleq \forall \sigma_1, \sigma_2.\, \sigma_1 \in \text{Normal } P \Rightarrow \langle c, \sigma_1\rangle \Downarrow \sigma_2 \Rightarrow \\ \sigma_2 \in \text{Normal } Q \cup \text{Abrupt } Q_{\mathrm{ek}}\end{aligned}$$

The above definition says after the execution from program states satisfying the precondition, the resulting state satisfies the normal and abrupt post-conditions depending on its exit kind. Both Normal Q and Abrupt Q_{ek} exclude any faulty state, ensuring the safety of the program. Simpl also has mechanisms to ignore certain errors in the program logic and to support total correctness, which we omit here.

CSimpl [38] is a framework for concurrent program verification based on rely guarantee reasoning. Its programming language is a deeply embedded imperative language supporting concurrency. Its logic is deeply embedded by giving inference rules of the logic with judgement $\vdash \{P\}c\{Q, Q_{\mathrm{ek}}\}$, where P is the pre-condition and Q, Q_{ek} are normal and control flow post-conditions respectively. It uses the rely-guarantee method to reason about concurrency. We omit details related to the usage of rely-guarantee in its formalization and refer readers to its original paper. To show the soundness of its proof system, CSimpl defines judgement's validity in three steps.

- It uses assum(c, P) to denote a set of small-step reduction steps of c (lists of intermediate commands and states) from pre-condition P.
- It uses comm(Q, Q_{ek}) to denote a set of reduction steps that have terminal state satisfying post-conditions Q, Q_{ek}.
- The judgement $\vdash \{P\}c\{Q, Q_{\mathrm{ek}}\}$ is valid iff. assum(c, P) is a subset of comm(Q, Q_{ek}). The inclusion implies any execution of c from pre-condition P will terminate in post-conditions Q, Q_{ek}.

The definition of the set comm(Q, Q_{ek}) is almost identical to the weakest precondition in Sect. 4.2. In conclusion, CSimpl uses deeply embedded logic with the weakest-precondition-based shallow embedding as its validity.

Certified assembly program (CAP) [46] and XCAP [31] use deeply embedded language and deeply embedded logic with judgement $\vdash \{P\}\, c$ to verify assembly programs. The logic guarantees that an assembly program c can execute safely if the program state satisfies the pre-condition P. Both work formalize the logic with a set of inference rules as its deep embedding and prove the soundness under the continuation-based shallow embedding. Both work's objective is to verify the program's safety instead of functional correctness, therefore the judgement and logic look different from the one in our paper.

Xu et al. [45] develop μC, a framework for verifying preemptive operating systems. The framework uses deeply embedded program logic. The logic is proven sound w.r.t. a weakest-precondition-based shallow embedding, which is similar to those in Sect. 4.2.

Software foundation [36] introduces a toy example of a deeply embedded Hoare logic with big-step-based shallow embedding as its validity, as we have mentioned in Sect. 4.1.

4.5 Other Logic Based Verification Methods

So far, we have discussed four Hoare logic embeddings and foundational verification frameworks that use these embeddings. Most of these frameworks use Hoare triples as program specifications and prove them mainly by applying proof rules to Hoare triples, which we refer to as Hoare-logic-based verification. Nevertheless, there exist other logic-based verification methods that do not use Hoare triples and Hoare logics as their primary tools. In this section, we will briefly review some of these non-Hoare-logic-based verification techniques: Hoare monad and Dijkstra monad in Sect. 4.5.1, Iris (weakest-precondition-based verification) in Sect. 4.5.2, and Characteristic Formulae in Sect. 4.5.3. It is not clear whether these approaches are better than pure Hoare-logic-based ones, and the comparison among them is not the focus of this paper.

4.5.1 Hoare Monad and Dijkstra Monad

Instead of Hoare logic, we can also use Hoare monad and Dijkstra monad to assert and verify the correctness of a program. Based on their Hoare type theory [30], Nanevski et al. [29] type a program using the Hoare monad $\mathtt{ST}\ P\ A\ Q$, where the type of the program asserts the its execution from state satisfying the pre-condition P will return with a value of type A, and the return value and the ending state will satisfy the post-condition Q over both the return value and the program state. The type of a large program is derived from the types of statements that assemble it.

$$\textsc{bind- hst}\ \frac{\vdash e_1 : (v : B) \to \mathtt{ST}\,(R\,v)\ A\ Q \qquad \vdash e_2 : \mathtt{ST}\ P\ B\ R}{\vdash (e_1\, e_2) : \mathtt{ST}\ P\ A\ Q}$$

For example, the rule above defines how to compose two Hoare monads through the bind operation (lambda function application), which is similar to the HOARE- SEQ rule. The type of $e_1\, e_2$ is derived by the type of e_1 and e_2.

The typing proof of Hoare monad cannot be easily automated due to some existential quantifiers over some intermediate program state. A series of works [1, 19, 27, 41] develops Dijkstra monad to improve the automation of such technique and allow reasoning about program with more features like exception, non-determinism, and IO. Different from Hoare monad, Dijkstra monad types a program as $\texttt{WP\_ST}\ A\ \mathsf{wp}$, a predicate transformer mapping a postcondition of the computation to its precondition, where A is the return type and wp is the weakest precondition. The weakest precondition wp is of the type

$$(A \times \text{state} \rightarrow \text{Prop}) \rightarrow \text{state} \rightarrow \text{Prop}$$

which is exactly a predicate transformer from the postcondition to the precondition. Below is the typing rule for the bind in the Dijkstra monad.

$$\textsc{bind- dst}\ \frac{\vdash e_2 : \texttt{WP\_ST}\ B\ \mathsf{wp}_2 \qquad \vdash e_1 : (v : B) \rightarrow \texttt{WP\_ST}\ A\ (\mathsf{wp}_1\ v)}{\vdash (e_1\ e_2) : \texttt{WP\_ST}\ A\ (\lambda Q\ s.\ \mathsf{wp}_2\ (\lambda x\ s_1.\ \mathsf{wp}_1\ x\ Q\ s_1)\ s)}$$

This rule composes predicate transformers $\mathsf{wp}_1\ v$ (parameterized over the return value v) and wp_2 into a new one to type $e_1\ e_2$.

The typing of programs such as Hoare monads and Dijkstra monads resembles the definition of the program's big-step semantics, as all of them are definitions by structural induction over the program's syntax. Still using the bind operation as an example, BIND- BIGS defines the big-step semantics of $e_1\ e_2$.

$$\textsc{bind- bigs}\ \frac{(e_2, \sigma_1) \Downarrow (v_2, \sigma_3) \qquad (e_1, \sigma_3) \Downarrow (\lambda x.e_1', \sigma_4) \qquad (e_1'[x \mapsto v_2], \sigma_4) \Downarrow (v_1, \sigma_2)}{(e_1\ e_2, \sigma_1) \Downarrow (v_1, \sigma_2)}$$

It is similar to BIND- HST except that the relationship between the beginning state and ending state defined in BIND- BIGS becomes the relationship between the precondition and postcondition in BIND- HST. This similarity is even more obvious in BIND- DST, where programs' effects on the pre/post-conditions are composed instead of composing their effects on the program state.

4.5.2 Weakest-Precondition-Based Verification

In Sect. 4.2, we have embedded a Hoare triple using the weakest precondition: $\vDash_w \{P\}c\{Q\}$, iff. for any state σ satisfying precondition P, we have $\sigma \vDash \mathsf{WP}(c, \epsilon)\{Q, [\vec{R}]\}$. Although we use the Hoare logic to reason about triples using this embedding, it is also possible to directly reason about the weakest precondition, which is the approach taken by Iris [20].

We use the IF- SEQ rule to demonstrate the weakest-precondition-based verification method and its advantage. Suppose we want to prove $\{P\}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ ;\ c_3\{Q\}$ in Iris for some c_1, c_2, c_3, e. It is equivalent to prove that $P \mathrel{-\!\!*} \mathrm{WP}((\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ ;\ c_3)\{Q\}$, which has the following weakest precondition derivation (13). Since Iris's meta-logic is a separation logic, we may use $\mathrel{-\!\!*}$ in place of implications.

$$\frac{\dfrac{\begin{array}{c}(P * [\![e]\!] = \texttt{true}) \mathrel{-\!\!*} \mathrm{WP}c_1\{\lambda v.\mathrm{WP}(v\ ;\ c_3)\{Q\}\} \quad (\dagger) \\ (P * [\![e]\!] = \texttt{false}) \mathrel{-\!\!*} \mathrm{WP}c_2\{\lambda v.\mathrm{WP}(v\ ;\ c_3)\{Q\}\} \quad (\ddagger)\end{array}}{P \mathrel{-\!\!*} \mathrm{WP}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\{\lambda v.\mathrm{WP}(v\ ;\ c_3)\{Q\}\}}\ \textsc{wp- if}}{P \mathrel{-\!\!*} \mathrm{WP}((\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ ;\ c_3)\{Q\}}\ \textsc{wp- bind} \tag{13}$$

We can first apply the WP- BIND rule to hide the second part c_3 of the sequential composition into the post-condition as another weakest precondition. The rule WP- BIND extract the evaluation context (K) of the first command to execute (c) into a separate weakest precondition into the post-condition. This enables provers to focus on the execution of command c before verifying the evaluation of K that will be affected by the outcome v of c.

$$\text{WP- BIND}\ \frac{\mathrm{WP}\, c\, \{\lambda v.\, \mathrm{WP}\, K[v]\, \{Q\}\}}{\mathrm{WP}\, K[c]\, \{Q\}}$$

Then, by WP- IF rule, we need to prove the weakest preconditions of two branches, (†) and (‡), separately. Then, take (†) for example, we can further symbolically execute c_1 in Iris's weakest precondition calculus until it reaches a terminal value v_1 with the memory satisfying P'. Then, by WP- VAL and WP- SEQ, we can further reduce the proof goal into a weakest precondition only about the remaining c_3 and eventually reach a tautology after symbolic executions of c_3's weakest precondition. We can do a similar proof for the other branch (‡) as well.

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{P' \mathrel{-\!\!*} \mathrm{WP} c_3\{Q\}}}{P' \mathrel{-\!\!*} \mathrm{WP}(v_1\,;\,c_3)\{Q\}}\ \text{WP- SEQ}}{P' \mathrel{-\!\!*} \mathrm{WP} v_1\{\lambda v.\mathrm{WP}(v\,;\,c_3)\{Q\}\}}\ \text{WP- VAL}}{\cfrac{\vdots}{(P * [\![e]\!] = \texttt{true}) \mathrel{-\!\!*} \mathrm{WP} c_1\{\lambda v.\mathrm{WP}(v\,;\,c_3)\{Q\}\}}} \tag{14}$$

As we can see, the proof of two branches (†) and (‡) is exactly the proof of two branches of $\texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3$. The ability to put future computations (like c_3 in this example) into the post-condition and postpone their proofs essentially makes extended rules like IF- SEQ for free.

Users of Iris mainly use this kind of symbolic execution for weakest precondition, instead of the symbolic execution for Hoare triples. Therefore, most of our discussions about extended rules do not apply to the Iris framework.

4.5.3 Characteristic Formulae

Charguéraud [10, 11] and Guénea et al. [17] use characteristic formulae to verify programs and builds CFML above its logic for the verification of imperative Caml programs. The main idea of characteristic formulae is to define a function cf of type command $\rightarrow$ (assertion $\rightarrow$ assertion $\rightarrow$ Prop). The result $\mathsf{cf}(c)$ defines the relationship between the precondition and the postcondition of command c's Hoare triple. For example, the result for the sequential composition is defined below.

$$\mathsf{cf}(c_1\,;\,c_2) \triangleq \lambda P.\,\lambda Q.\,\exists R.\,\mathsf{cf}(c_1)\;P\;R \wedge \mathsf{cf}(c_2)\;R\;Q \tag{15}$$

This formalization is highly similar to the deeply embedded Hoare logic. The proof tree for a program in a deeply embedded Hoare logic is built recursively according to the program's syntax tree. While characteristic formulae are also generated recursively according to the program's syntax tree in a similar fashion. For example, the definition (15) resembles the deeply embedded sequence rule below.

$$\text{SEQ}\ \frac{\vdash \{P\}c_1\{R\} \qquad \vdash \{R\}c_2\{Q\}}{\vdash \{P\}c_1\,;\,c_2\{Q\}}$$

After a characteristic formula is generated, users can then verify in the meta-logic whether it satisfies the specification they want to prove. In this way, they can directly apply features in the meta-logic (usually the one directly supported by the interactive theorem prover like Coq) to do the proof.

On the other hand, the characteristic formulae generator cf is proved sound w.r.t. a big-step semantics,

$$\forall c, P, Q.\, \mathsf{cf}(c)\ P\ Q \Rightarrow \vDash_b \{P\}c\{Q\} \tag{16}$$

which makes the framework foundational. Here, the soundness definition (16) is a simplified version using the big-step-based embedding (but without control flows) introduced in Sect. 4.1, while the original definition in Charguéraud's work (Vol.6 of Software Foundation [10]) also considers features like separation logics.

5 Proving Extended Rules in Different Embeddings

So far, we introduced three categories of extended proof rules in Sect. 2 and reviewed four common embedding methods for Hoare logic in Sect. 4. This section investigates proofs or disproofs of these rules under different embedding methods. We start by arguing the necessity to prove extended rules in different embeddings.

Different embeddings define different sets of triples. Given a Hoare logic embedding $\vDash$, we use it to define a set of all triples that are satisfiable under the embedding as $\{S \mid\ \vDash S\}$. For two different embeddings $\vDash_1$ and $\vDash_2$, if there exists some triple S such that $\vDash_1 S \not\Leftrightarrow \vDash_2 S$, then we know sets $\{S \mid\ \vDash_1 S\}$ and $\{S \mid\ \vDash_2 S\}$ are not equal. Typically, a property that holds for elements in one set does not necessarily hold for those in a different set. Extra elements in a different set may not satisfy the property. As a result, ***a rule admissible in one embedding is not necessarily admissible in another embedding***. Thus, it is necessary to check whether each extended rule holds under each of these embeddings.

- The shallow embeddings we have considered so far define different sets of triples. Although it is possible to bridge the big-step-based embedding and the weakest-precondition-based embedding by proving the equivalence of two semantics, the equivalence between these embeddings and the continuation-based one is not trivial. The continuation-based embedding quantifies over an arbitrary continuation following the original command, which is absent in other embeddings. It also employs a reversed implication pattern, a property about post-conditions implies a property about the pre-condition, while other embeddings are not. The difficulty in proofs of extended rules also makes us to believe that different embeddings define different sets of triples.
- The deep embedding in this paper defines different sets of triples from those under shallow embeddings. This is true unless the deep one is proven both sound and complete w.r.t. the shallow one. However, completeness is usually not necessary for a program verification tool because users mainly require the correctness (soundness) of the tool.
In practice, tool developers do not provide completeness proof because the proofs involved are challenging for a complex program logic. Therefore, we assume the deeply embedded logic is not complete.

We find all extended rules in Fig. 3 can be proved for all four embeddings, but some proofs are complicated and some proof obligations may be unreasonably heavy in reality for a more complex language. We list our techniques to overcome these difficulties in Table 2 and make some brief comments here.

Table 2 Extended proof rules in different embedding methods

Proof Rule	Deep embed[c] (Sect. 5.1)	Big-step shallow embed (Sect. 5.2)	Weakest pre. shallow embed (Sect. 5.3)	Continuation shallow embed (Sect. 5.4)
SEQ- INV	Simple	Simple	Medium: By Simulation	Difficult: By Simulation and Construction[b]
NOCONTINUE	Simple	Simple	Medium: By Simulation	Difficult: By Simulation and Construction[b]
IF- SEQ	Simple	Simple	Medium: By Simulation	Medium: By Simulation
LOOP- NOCONTINUE	Simple	Simple	Medium: By Simulation	Medium: By Simulation
HOARE- EX	Difficult[a]	Simple	Simple	Simple

(a) HOARE- EX holds in deeply embedded logic, but the proof is complicated and requires impredicative assertions
(b) Many proofs in the continuation-based embedding are intricate. We can prove NOCONTINUE and SEQ- INV with complex constructions of continuation κ. But the feasibility of such construction and the complexity of relevant proofs remains unknown for different and more complicated programming languages
(c) The deep embedding and its proofs are implemented in our deeply embedded VST, which is integrated into the VST repository (https://github.com/PrincetonUniversity/VST)

Extended Rules — Transformation Rules

$$\text{IF-SEQ}\ \frac{\vdash \{P\}\texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3\{Q,[\vec{R}]\}}{\vdash \{P\}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;c_3\{Q,[\vec{R}]\}}$$

$$\text{LOOP-NOCONTINUE}\ \frac{c_1, c_2\ \text{contain no}\ \texttt{continue} \qquad \vdash \{P\}\texttt{for}(;;\,\texttt{skip})\ c_1\,;c_2\{Q,[\vec{R}]\}}{\vdash \{P\}\texttt{for}(;;\,c_2)\ c_1\{Q,[\vec{R}]\}}$$

Extended Rules — Structural Rules

$$\text{HOARE-EX}\ \frac{\text{forall}\ x.\ \vdash \{P(x)\}c\{Q,[\vec{R}]\}}{\vdash \{\exists x.\,P(x)\}c\{Q,[\vec{R}]\}} \qquad \text{NOCONTINUE}\ \frac{c\ \text{contains no}\ \texttt{continue} \quad \vdash \{P\}c\{Q,[R_{\text{brk}}, R_{\text{con}}]\}}{\vdash \{P\}c\{Q,[R_{\text{brk}}, R'_{\text{con}}]\}}$$

Extended Rules — Inversion Rules

$$\text{SEQ-INV}\ \frac{\vdash \{P\}c_1\,;c_2\{Q,[\vec{R}]\}}{\text{exists}\ Q'.\ \vdash \{P\}c_1\{Q',[\vec{R}]\}\ \text{and}\ \vdash \{Q'\}c_2\{Q,[\vec{R}]\}}$$

Fig. 10 Representative extended proof rules to be proved

- Most proofs for the *deep* embedding, except the proof of HOARE- EX, are simple inductions over proof trees. Rules are proved correct mainly by definitions for the *big-step* based embedding. These proofs are relatively trivial.
- Most proofs in the *weakest precondition* based embedding involve defining a simulation relation between pairs of expressions and program states. The IF- SEQ and LOOP-NOCONTINUE proof in the *continuation* based embedding uses the same idea. The principle behind these proofs is not difficult but do require significantly more work than those for *deep* embedding and *big-step* based embedding. Thus, we would say they are of medium difficulty.
- Proofs for SEQ- INV and NOCONTINUE in the *continuation* based embedding not only require this simulation technique, but also demand complex constructions of continuations. These proofs are very complicated.
- HOARE- EX's proofs under shallow embeddings are trivial but are relatively complicated under deep embedding.

In following sections, we will demonstrate our proofs under different embeddings using our WhileCF toy language. These proofs are one of the major contributions of the paper and are formalized in Coq [15].

To review, we paste the extended rules we are going to prove in Fig. 10.

5.1 Deep Embedding

5.1.1 The Choice of the Deep Embedding

Before proving extended rules in *some* deeply embedded language, we should first fix *the* deep embedding we want to discuss. Theoretically, difference choices of the primary rule set determine different deep embeddings of program logic. We find that, if we only pick compositional rules, singleton command rules and the consequence rule, we can prove all inversion rules quite easily by proof-tree induction, and on top of that, we can then establish most transformation rules and structural rules in a straightforward way. We choose the deep embedding with rules in Fig. 2 as primary rules. We first stick to this specific deep embedding in this subsection to demonstrate extended rules' proofs and discuss other possible choices afterwards in Sect. 7.

Proving Inversion Rules.
The proofs of inversion rules in the deep embedding are mainly by induction over the original proof trees. We prove SEQ- INV and LOOP- INV in Theorem 3 and Theorem 4 for demonstration. Other inversion rules like IF- INV have similar proofs.

Theorem 3 *For any P, c_1, c_2, Q and $\vec{R}$, if $\vdash \{P\}c_1\,;c_2\{Q, [\vec{R}]\}$, then there exists another assertion S such that $\vdash \{P\}c_1\{S, [\vec{R}]\}$ and $\vdash \{S\}c_2\{Q, [\vec{R}]\}$.*

Proof We prove it by induction over the proof tree of shape $\vdash \{\cdot\}c_1\,;c_2\{\cdot, [\cdot]\}$. In fact, the last step of proving $\{P\}c_1\,;c_2\{Q, [\vec{R}]\}$ is either HOARE- SEQ or HOARE- CONSEQUENCE. For the former situation, the conclusion obviously holds. For the latter situation, we can find P', Q' and $\vec{R}'$ such that

$$\vdash \{P'\}c_1\,;c_2\{Q', [\vec{R}']\},\ P \vdash P',\ Q' \vdash Q,\ R'_{\text{brk}} \vdash R_{\text{brk}},\ \text{and}\ R'_{\text{con}} \vdash R_{\text{con}}.$$

By induction hypothesis, we can find S such that $\vdash \{P'\}c_1\{S, [\vec{R}']\}$ and $\vdash \{Q\}c_2\{Q', [\vec{R}']\}$. Then, we get $\vdash \{P\}c_1\{S, [\vec{R}]\}$ and $\vdash \{S\}c_2\{Q, [\vec{R}]\}$ by HOARE- CONSEQUENCE.

□

Theorem 4 *For any P, c_1, c_2, Q and $\vec{R}$, if $\vdash \{P\}\texttt{for}(;;c_2)\ c_1\{Q, [R_{brk}, R_{con}]\}$, then there exists two assertions I_1, I_2 such that*

$$\vdash \{I_1\}c_1\{I_2, [Q, I_2]\}\ \textit{and}\ \vdash \{I_2\}c_2\{I_1, [Q, \bot]\},\ \textit{and}\ P \vdash I_1.$$

Proof We prove it by induction over the proof tree of shape $\vdash \{\cdot\}\texttt{for}(;;c_2)\ c_1\{\cdot\}$. Similar to the previous proof, the last step in the proof tree is either HOARE- LOOP or HOARE- CONSEQUENCE. For the first case, the conclusion trivially holds. For the second case, we can find P', Q' and $\vec{R}'$ such that

$$\vdash \{P'\}\texttt{for}(;;c_2)\ c_1\{Q', [\vec{R}']\},\ P \vdash P',\ Q' \vdash Q,\ R'_{\text{brk}} \vdash R_{\text{brk}},\ \text{and}\ R'_{\text{con}} \vdash R_{\text{con}}.$$

By induction hypothesis, we can find I_1, I_2 such that $\vdash \{I_1\}c_1\{I_2, [Q', I_2]\}$ and $\vdash \{I_2\}c_2\{I_1, [Q', \bot]\}$ and $P' \vdash I_1$. This also implies $P \vdash I_1$ and by HOARE- CONSEQUENCE we have I_1, I_2 such that $\vdash \{I_1\}c_1\{I_2, [Q, I_2]\}$ and $\vdash \{I_2\}c_2\{I_1, [Q, \bot]\}$. □

Proving HOARE- EX ***rule***.
HOARE- EX (Theorem 7) is hard to prove in a deeply embedded logic since the proof tree of $\vdash \{P(x)\}c\{Q, [\vec{R}]\}$ may be different for a different x. We prove it by induction on the syntax tree of c instead. Here, we only show one induction step in the proof—the case for sequential composition. Other induction steps are proved similarly. A complete proof can be found in our Coq development.

This proof is based on one important assumption about the assertion language: we can quantify over assertions inside an assertion and inject from the meta-logic's propositions into the assertion language, and moreover:

Hypothesis 5 For any assertion P, Q, $\vec{R}$ and program c, if $\vdash \{P\}c\{Q, [\vec{R}]\}$ holds, then we have $P \ \vdash \ \exists P_0.\ \Big(\vdash \{P_0\}c\{Q, [\vec{R}]\}\Big) \wedge P_0$.

This hypothesis above is an entailment in the assertion logic and the right-hand side is an assertion. In this assertion, P_0 is an existentially quantified assertion inside another assertion, and $\left(\vdash \{P_0\}c\{Q, [\vec{R}]\}\right)$ is a meta-logic proposition, stating that a triple is provable, used as a conjunct in a compound assertion. This hypothesis is obviously true since the existentially quantified P_0 on the right-hand side can be instantiated by P.

Remark This kind of assertions, where universe quantifiers and existential quantifiers can quantify over assertions and Hoare triples can be admitted as assertions, are known as impredicative assertions or impredicative polymorphism [31]. Our proof here assumes impredicative assertions are available in the assertion language, which we discuss later in Sect. 8.1.2.

Back to our proof, Hypothesis 5 and SEQ- INV immediately validates Lemma 6, which is used in the proof of HOARE- EX (Theorem 7).

Lemma 6 *For any P, c_1, c_2, R and $\vec{S}$, if $\vdash \{P\}c_1\,;c_2\{R, [\vec{S}]\}$, then*

$$\vdash \{P\}\,c_1\left\{\exists Q.\left(\vdash \{Q\}c_2\{R, [\vec{S}]\}\right) \wedge Q, [\vec{S}]\right\}.$$

Theorem 7 *For any T, P, c, Q and $\vec{R}$, if for any x of type T, we have $\vdash \{P(x)\}c\{Q, [\vec{R}]\}$, then $\vdash \{\exists x.\ P(x)\}c\{Q, [\vec{R}]\}$.*

Proof By induction on the syntax tree of c, the induction step for the sequential composition is to prove $\vdash \{\exists x.\ P(x)\}c_1\,;c_2\{R, [\vec{S}]\}$ under the following assumptions:

(IH1) For any type T and P, Q, R, if $\vdash \{P(x)\}c_1\{Q, [\vec{R}]\}$ holds for any $x : T$, then $\vdash \{\exists x.\ P(x)\}c_1\{Q, [\vec{R}]\}$.
(IH2) For any type T and P, Q, R, if $\vdash \{P(x)\}c_2\{Q, [\vec{R}]\}$ holds for any $x : T$, then $\vdash \{\exists x.\ P(x)\}c_2\{Q, [\vec{R}]\}$.
(Assu) $\vdash \{P(x)\}c_1\,;c_2\{R, [\vec{S}]\}$ holds for any $x : T$.

First, by (Assu) and Lemma 6, $\vdash \{P(x)\}\,c_1\left\{\exists Q.\left(\vdash \{Q\}c_2\{R, [\vec{S}]\}\right) \wedge Q, [\vec{S}]\right\}$ holds for any x of type T. Then, using (IH1), we can get:

$$\vdash \{\exists x.\ P(x)\}\,c_1\left\{\exists Q.\left(\vdash \{Q\}c_2\{R, [\vec{S}]\}\right) \wedge Q, [\vec{S}]\right\}.$$

Now, if we can prove $\vdash \{\exists Q.\left(\vdash \{Q\}c_2\{R, [\vec{S}]\}\right) \wedge Q\}c_2\{R, [\vec{S}]\}$, then our conclusion will immediately follow according to HOARE- SEQ. In fact, it is straightforward. By (IH2), we only need to prove that for any assertion Q,

$$\vdash \{\left(\vdash \{Q\}c_2\{R, [\vec{S}]\}\right) \wedge Q\}c_2\{R, [\vec{S}]\}.$$

Suppose T_0 is the set of proofs of $\vdash \{Q\}c_2\{R, [\vec{S}]\}$, then

$$\left(\vdash \{Q\}c_2\{R, [\vec{S}]\}\right) \wedge Q \dashv\vdash \exists x : T_0.\ Q$$

Thus, by HOARE- CONSEQUENCE, we only need to prove $\vdash \{\exists x : T_0.\ Q\}c_2\{R, [\vec{S}]\}$. By (IH2) again, we only need to prove: if $\vdash \{Q\}c_2\{R, [\vec{S}]\}$ has a proof, then $\vdash \{Q\}c_2\{R, [\vec{S}]\}$; this is tautology. Now we complete the induction step for sequential composition. $\square$

The above proof of the sequential composition branch in proving Theorem 7 is based on SEQ- INV through Lemma 6. In fact, the induction over the syntax tree of the program c has to consider all possible constructors of a command. Therefore, it is necessary to develop inversion rules and inversion lemmas like Lemma 6 for each constructor. Moreover, *only careful design of primary rules for the deeply embedded Hoare logic can establish these inversion rules: for each program constructor, there should exist exactly one primary rule corresponding to it*. Otherwise, the indeterminacy of the Hoare triple's proof tree will obstruct the proof of inversion rules.

Proving Other Rules.
The proof of other extended rules in the deep embedding follows a common pattern. *We first extract information from Hoare triples in premises using inversion rules and then use primary rules to combine these information to establish new triples.* This is exactly the proof scheme presented in Fig. 4 in Sect. 2.3 when we prove NOCONTINUE using inversion rules. The proof for other rules is similar to this one and is omitted here.

This proof technique using inversion rules can also be applied to shallowly embedded logics, if these inversion rules are true. However, we would like to explore proofs without assuming the correctness of inversion rules in the following sections. Even though they are true for our toy language and shallow embeddings for their logics, they may not be true in reality when some other features are added to the logic. In these cases, a proof to circumvent inversion rules would be helpful.

Application in VST-A.
VST-A [49] is an annotation verifier built above VST. Its soundness proof not only utilizes inversion rules but also the proof pattern used in this section.

VST-A accepts programs with assertion annotations provided by the programmer. The annotations specify what the programmer believes the program state will satisfy at a given point during program execution and are used to guide the verification of the whole program. VST-A splits the program into paths connecting these assertions and the pre-/post-conditions and generates proof goals (Hoare triples) of these paths. These paths are much simpler than the whole program, so that the verifier can automate most of their proofs, and the prover can prove the remaining one with reasonable effort. Eventually, VST-A's soundness will guarantee the correctness of the whole program given the Hoare triples of all these paths. VST-A's soundness proof is formalized as the following proposition.

Proposition 1 (VST-A Soundness) *For any P, Q, C, where C is the annotated program, if* $\text{AllHyps}(P, Q, \text{paths}(C))$ *is true, then* $\{P\}C \Downarrow \{Q\}$ *is true.*

Here $\text{AllHyps}(P, Q, \text{paths}(C))$ means triples of all paths in C generated by VST-A is provable, and $\{P\}C \Downarrow \{Q\}$ means the Hoare triple holds with $C \Downarrow$ removing annotations from C. VST-A uses induction over C's syntax tree to prove the soundness. For example, in the sequential composition case, the authors of VST-A need to find an intermediate assertion R and prove:

$$\begin{array}{ll} \text{if} & \text{AllHyps}(P, R, \text{paths}(C_1)) \Rightarrow \{P\}C_1 \Downarrow \{R\} \\ \text{and} & \text{AllHyps}(R, Q, \text{paths}(C_2)) \Rightarrow \{R\}C_2 \Downarrow \{Q\} \\ \text{then} & \text{AllHyps}(P, Q, \text{paths}(C_1\,;C_2)) \Rightarrow \{P\}C_1\,; C_2 \Downarrow \{Q\} \end{array}$$

To prove this, it suffices to show that $\text{AllHyps}(P, Q, \text{paths}(C_1\,; C_2))$ ensures AllHyps $(P, R, \text{paths}(C_1))$ and $\text{AllHyps}(R, Q, \text{paths}(C_2))$, i.e., they need to split Hoare triples for paths in $C_1\,; C_2$ to paths in C_1 and C_2. A non-trivial case is where a path includes commands from both C_1 and C_2, and they use the inversion lemma for HOARE- SEQ to extract the intermediate assertion R and prove this condition using lemmas similar to Hypothesis 5 and Lemma 6.

5.2 Big-Step-Based Shallow Embedding

Proving Transformation Rules.

There is a general proof scheme for transformation rules under shallow embeddings. *We can first prove some notion of "semantic similarity" of two programs in the goal and premise, then lift this "semantic similarity" to the logic level and derive a relation between Hoare triples, i.e., the extended rule.* For example, the "semantic similarity" in the big-step-based embedding is exactly the refinement of the big-step reduction.

Definition 1 We say program c_1 refines c_2 ($c_1 \sqsubseteq c_2$) w.r.t. a big-step semantics iff.

- for any s_1, s_2, ek, if $(c_1, s_1) \Downarrow (\text{ek}, s_2)$, then $(c_2, s_1) \Downarrow (\text{ek}, s_2)$;
- and for any s, if $(c_1, s) \Uparrow$, then $(c_2, s) \Uparrow$.

Lemma 8 lifts the big-step refinement to the implication between Hoare triples using the big-step embedding.

Lemma 8 *For any c_1, c_2, if $c_2 \sqsubseteq c_1$, then for any $P, Q, \vec{R}$, $\vDash_b \{P\}c_1\{Q, [\vec{R}]\}$ implies $\vDash_b \{P\}c_2\{Q, [\vec{R}]\}$.*

Proof The proof of the theorem is straightforward. Assume the pre-state is s_1, which satisfies P.

- We use contradiction to prove c_2 has no error. If it can cause error, $(c_2, s_1) \Uparrow$, then according to $c_2 \sqsubseteq c_1$, we can also construct an error in c_1's execution, $(c_1, s_1) \Uparrow$, which contradicts with $\vDash_b \{P\}c_1\{Q, [\vec{R}]\}$.
- For any ending configuration (ek, s_2) which c_2 can reach, we construct $(c_1, s_1) \Downarrow (\text{ek}, s_2)$ from $(c_2, s_1) \Downarrow (\text{ek}, s_2)$ according to $c_2 \sqsubseteq c_1$. By $\vDash_b \{P\}c_1\{Q, [\vec{R}]\}$, we can show (ek, s_2) satisfies the post-condition and thus prove $\vDash_b \{P\}c_2\{Q, [\vec{R}]\}$

□

With Lemma 8, proofs for transformation rules IF- SEQ and LOOP- NOCONTINUE become apparent with the following two refinements.

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;c_3 \sqsubseteq \texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3$$
$$\texttt{for}(;;c_2)\ c_1 \sqsubseteq \texttt{for}(;;\texttt{skip})\ c_1\,;c_2$$

Their proofs are straightforward. For example, we can split the big-step reduction

$$((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;c_3, s_1) \Downarrow (\epsilon, s_2)$$

into the following proposition, which essentially encodes the reduction segments of two execution paths.

$$\begin{aligned}(\text{eval}(b, s_1) = \texttt{true} \wedge (c_1, s_1) \Downarrow (\epsilon, s_3) \wedge (c_3, s_3) \Downarrow (\epsilon, s_2))\\ \vee (\text{eval}(b, s_1) = \texttt{false} \wedge (c_2, s_1) \Downarrow (\epsilon, s_3) \wedge (c_3, s_3) \Downarrow (\epsilon, s_2))\end{aligned}$$

We can easily reconstruct a reduction of $\texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3$ from s_1 to (ϵ, s_2) using this proposition. Other branches of IF- SEQ's refinement proof are similar, which can be easily enumerated by discussing four different reductions (three different exit control flows and one error case) of the program.

The proof for the loop-nocontinue is similar but involves induction over the loop iteration. The reduction of each iteration of `for(;;`c_2`)` c_1 will be split into two reduction segments of c_1 and c_2, which can be easily combined into a reduction of $c_1 \,;\, c_2$ since both contain no `continue`. Since the reduction of $c_1 \,;\, c_2$ is exactly one iteration of `for(;;skip)` $c_1 \,;\, c_2$, combined with the induction hypothesis, it reproduces the reduction of `for(;;skip)` $c_1 \,;\, c_2$ and proves the refinement.

The above proof strategy is also taken by Software Foundations Vol. 6 [36] to prove a similar transformation rule in the big-step-based embedding. We will see later in the following sections that this approach can also be applied to the weakest-precondition-based embedding and the continuation-based embedding but can have quite different definitions of "semantic similarity".

Proving Structural Rules.
We then show the proof for HOARE- EX here and prove other rules in Appendix B. The conclusion for HOARE- EX holds for another two shallow embeddings with similar proofs, which we omit in the following sections.

Theorem 9 *For all c, P, and Q, if for all x, $\vDash_b \{P(x)\}c\{Q\}$, then $\vDash_b \{\exists x.\, P(x)\}c\{Q\}$.*

Proof For $\sigma \vDash \exists x.\, P(x)$, we know there exists some x such that $\sigma \vDash P(x)$. By the definition of $\vDash_b$, we can instantiate the premise with initial state σ and the conclusion immediately follows by definition. □

Proving Inversion Rules.
Proofs for the inversion rules of the big-step-based embedding is similar to those of the deep embedding. The big-step semantics is defined inductively w.r.t. the structure of a program, and we can reorganize the big-step relation underlying the triple just like what we do for the deeply embedded logic. As a result, we can easily prove inversion rules. We only demonstrate the proof for SEQ- INV here.

Theorem 10 *For all c_1, c_2, P, Q, and R, if $\vDash_b \{P\}c_1 \,;\, c_2\{Q, [\vec{R}]\}$, then there exists Q' such that (1) $\vDash_b \{P\}c_1\{Q', [\vec{R}]\}$ and (2) $\vDash_b \{Q'\}c_2\{Q, [\vec{R}]\}$.*

Proof We use the weakest pre-condition of c_2 as Q': $\sigma \vDash Q'$ iff. it never cause error, i.e., not $(c_2, \sigma) \Uparrow$, and $\forall \sigma_1.(c_2, \sigma) \Downarrow (\epsilon, \sigma_1)$ implies $\sigma_1 \vDash Q$.

(1) The safety of c_1 is guaranteed by the safety of $c_1 \,;\, c_2$ and we only need to show the state after c_1 terminates satisfies post-conditions. If c_1 exits normally, we combine the premise in the weakest pre. and have $(c_1 \,;\, c_2, \sigma) \Downarrow (\epsilon, \sigma_1)$. By premise, we have $\sigma_1 \vDash Q$ and the intermediate state satisfies Q' the conclusion holds by the definition of the strongest postcondition. If c_1 exits by control flow, say $(c_1, \sigma) \Downarrow (\text{ek}, \sigma')$, we have $(c_1 \,;\, c_2, \sigma) \Downarrow (\text{ek}, \sigma')$ and by premise, we prove the conclusion.

(2) The conclusion is obvious by the definition of weakest precondition Q'. □

5.3 Weakest-Precondition-Based Shallow Embedding

Proving Transformation Rules: Refinement.
The principle for proving transformation rules in Sect. 5.2 still applies to small-step-based embeddings (both weakest-precondition-based embedding and continuation-based embedding). But we need to find an appropriate definition of "semantic similarity". A first attempt is to use the refinement again, but under the small-step semantics.

Definition 2 We say program c_1 refines c_2 (denoted $c_1 \sqsubseteq_s c_2$) in small-step semantics iff. for any initial states σ_1 and terminal configuration (c, κ, σ), the following is true.

$$((c_1, \epsilon, \sigma_1) \rightarrow (c, \kappa, \sigma)) \Rightarrow ((c_2, \epsilon, \sigma_1) \rightarrow (c, \kappa, \sigma))$$

We also have Lemma 11 to lift the semantic refinement to the implication between weakest preconditions, which immediately implies the implication between Hoare triples.

Lemma 11 *For any c_1, c_2, if $c_2 \sqsubseteq_s c_1$, then for any σ, P, Q, $\vec{R}$, $\sigma \vDash \mathrm{WP}(c_1, \epsilon)\{Q, [\vec{R}]\}$ implies $\sigma \vDash \mathrm{WP}(c_2, \epsilon)\{Q, [\vec{R}]\}$.*

The proof for Lemma 11 is a simple and straightforward co-induction over $\sigma \vDash \mathrm{WP}(c_2, \epsilon)\{Q, [\vec{R}]\}$.

To use the lemma, we need to again prove the following two refinements.

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;\,c_3 \sqsubseteq_s \texttt{if}\ b\ \texttt{then}\ c_1\,;\,c_3\ \texttt{else}\ c_2\,;\,c_3$$
$$\texttt{for(;;}c_2\texttt{)}\ c_1 \sqsubseteq_s \texttt{for(;;skip)}\ c_1\,;\,c_2$$

A possible proof idea is similar to the one in Sect. 5.2. We need to split the reduction of the left-hand-side program into several reduction segments along all execution paths. However, the notion of "reduction segments" is not for free in the small-step semantics as it is in the big-step semantics. For example, a direct inversion of the following reduction does not yield the reduction segments of c_1, c_2, and c_3.

$$((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;\,c_3, \epsilon, \sigma_1) \rightarrow^* (c, \kappa, \sigma)$$

We need to manually prove that the sequential composition can be split into two reductions and the if-statement can be split into two reductions of two branches, i.e., there exists an ending configuration[10] (c', κ', σ'), such that

$$\begin{aligned}&\big(((b, \sigma_1) \rightarrow^* (\texttt{true}, \sigma_1) \wedge (c_1, , \sigma_1) \rightarrow^* (c', \kappa', \sigma'))\\ &\quad \vee ((b, \sigma_1) \rightarrow^* (\texttt{false}, \sigma_1) \wedge (c_2, , \sigma_1) \rightarrow^* (c', \kappa', \sigma'))\big)\\ &\quad \wedge (\kappa' = \epsilon \Rightarrow (c', \kappa' \cdot \mathrm{KSeq}(c_3), \sigma') \rightarrow^* (c, \kappa, \sigma))\end{aligned}$$

The proof needs to be split into two separate lemmas for the sequential composition and the if-statement. Both are proved by induction over the reduction length. We can then reorganize these reduction segments to form a reduction of $\texttt{if}\ b\ \texttt{then}\ c_1\,;\,c_3\ \texttt{else}\ c_2\,;\,c_3$.

While it is still a reasonable proof for the IF- SEQ, the proof for the loop is overwhelming. In Sect. 5.2, we can directly induct over the number of loop iterations, since it is how the big-step semantic for loops is defined, we cannot do the same thing for loops in small-step semantics. A direct induction can only be performed to induct over the small-step reduction length! We need to have another inductive definition for reduction segments of a loop, which we present informally in definition 3.

Definition 3 For any ending configuration (c, κ, s), the reduction from $(\texttt{for(;;}c_2\texttt{)}\ c_1, \epsilon, s_1)$ to (c, κ, s) can be decomposed into an $n-$iteration segment iff.

- $n = 0$, then (c, κ, s) is reached by some $\texttt{break}$ from $(c_1\,;\,c_2, \epsilon, s_1)$, or it is some stuck configuration reachable from $(c_1\,;\,c_2, \epsilon, s_1)$, or c_2 eventually reduces to $\texttt{continue}$ and $(c, \kappa, s) = (\texttt{continue}, \mathrm{KLoop}_2(c_1, c_2), s)$ gets stuck.

[10] While the ending configuration of the big-step semantics can only be (ek, s) for any s, in the small-step semantics, it can be $(\texttt{skip}, \epsilon, s)$, $(\texttt{break}, \epsilon, s)$, $(\texttt{continue}, \epsilon, s)$, or any (c, ϵ, s) that cannot be reduced any more due to some error, e.g., $(\texttt{continue}, \mathrm{KLoop}_2(c_1, c_2), s)$ since it has no reduction defined.

- $n = n' + 1$, then $(c_1\,;\,c_2, \epsilon, s_1) \to^* (\texttt{skip}, \epsilon, s')$, and $(\texttt{for}(;;c_2)\, c_1, \epsilon, s') \to^* (c, \kappa, s)$ can be decomposed into an $n'-$iteration segment.

Numerous details of handling control flows have been neglected in this definition. To prove such decomposition is possible, we use induction over the reduction length, and it involves many auxiliary definitions, which we do not demonstrate here. Instead, we present another more organized proof technique: simulation.

Proving Structural Rules: Simulation.
The idea is to directly induct over the length of reductions for the left-hand-side program in a refinement, and for each head reduction, we find zero or finitely many head reductions of the right-hand-side program that corresponds to it. This is exactly the idea of simulation, and we use the simulation relation (definition 4) to record this kind of correspondence, which is used for both the weakest-precondition-based embedding and the continuation-based embedding.

Definition 4 **(Simulation)** A relation $\sim$ between two programs is a simulation relation iff. for any $c_1, \kappa_1, c_2, \kappa_2$, if the *source* program (c_1, κ_1) can be simulated by the *target* program (c_2, κ_2), i.e. $(c_1, \kappa_1) \sim (c_2, \kappa_2)$, then

- *Termination:* if (c_1, κ_1) has safely terminated, i.e., κ_1 is empty ϵ and c_1 is $\texttt{skip}$, $\texttt{break}$, or $\texttt{continue}$, then (c_2, κ_2) can reduce to (c_1, κ_1) in zero or finite steps without modifying the program state, (or it can stuck in an infinite loop without modifying the state)[11];
- *Preservation:* if for some program state σ, (c_1, κ_1, σ) reduces to $(c_1', \kappa_1', \sigma')$, then there exists (c_2', κ_2') simulates (c_1', κ_1') by $(c_1', \kappa_1') \sim (c_2', \kappa_2')$, and (c_2, κ_2) can reduce to (c_2', κ_2') in zero or finite steps with same state modifications, i.e., $(c_2, \kappa_2, \sigma) \to_c^* (c_2', \kappa_2', \sigma')$ [12]
- *Error:* if for some state σ, (c_1, κ_1, σ) does not belong to either of the above cases, i.e., it is stuck by an error, then (c_2, κ_2, σ) will also reduce into some error in zero or finite steps.

Intuitively, in the *termination* case, when the *source* terminates, the *target* that simulates the source will also terminate in finite number of steps and these steps have no virtual effect on the program state. In the *preservation* case, any reduction step of the *source* can be simulated by zero or finite steps of the *target*, which mimic the source's modifications to the program state, and program pairs they reduce to preserve the simulation relation. In the *error* case, when the *source* causes an error, the *target* will also raise an error along a certain execution trace.

One may expect that the simulation directly implies the refinement, but the simulation definition here is more general than the refinement because it does not require two programs to reduce to the same error. However, we can directly use the simulation as the notion of "semantic similarity" for embeddings using small-step semantics. We can use a simulation lemma (Lemma 12 and Lemma 16) to lift simulations to the logic level. The simulation lemma here (Lemma 12) lifts the simulation between two programs to the implication between their weakest preconditions.

[11] We add the disjunction in parentheses to aid the proof for rules only under the continuation-based embedding, and we remove it from the definition in this section.

[12] Notice that *stuttering*; might occur when (c_2, κ_2) always takes zero step, and (c_1, κ_1) may not terminate in this case. This is a common problem researchers want to avoid when proving simulations in compiler verifications. However, we are considering partial correctness in this paper and only want certain properties will hold when the program terminates. Stuttering is not a problem here.

Lemma 12 (Weakest Pre. Simulation) *If there is a simulation relation $\sim$ and (c_1, κ_1) simulates (c_2, κ_2), i.e., $(c_2, \kappa_2) \sim (c_1, \kappa_1)$, then for all $\sigma, c_1, c_2, \kappa_1, \kappa_2, Q$, and $\vec{R}$, $\sigma \models \mathit{WP}\,(c_1, \kappa_1)\{Q, [\vec{R}]\}$ implies $\sigma \models \mathit{WP}\,(c_2, \kappa_2)\{Q, [\vec{R}]\}$.*

Proof We prove the lemma by induction over the reduction of (c_2, κ_2, σ).

- If (c_2, κ_2) has terminated, the simulation implies (c_1, κ_1) will terminate with the same exit kind and we prove the conclusion by reflexivity.
- If (c_2, κ_2) has an error, the simulation implies (c_1, κ_1) will also have an error, which contradicts the safety guarantee (the program never causes an error) in its weakest precondition.
- If (c_2, κ_2) further reduces, the simulation relation guides us on how to reduce (c_1, κ_1) while preserving the simulation. We then unfold the weakest pre.'s definition and forward the execution accordingly and prove the lemma by the induction hypothesis.

□

To prove IF- SEQ and LOOP- NOCONTINUE, we only need to construct two simulations $\sim_1$ and $\sim_2$ with the following properties respectively.

$$((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\,;c_3, \epsilon) \sim_1 (\texttt{if}\ b\ \texttt{then}\ c_1\,;c_3\ \texttt{else}\ c_2\,;c_3, \epsilon)$$
$$(\texttt{for(;;}c_2\texttt{)}\ c_1, \epsilon) \sim_2 (\texttt{for(;;skip)}\ c_1\,;c_2, \epsilon)$$

For example, we can define the simulation for LOOP- NOCONTINUE as the smallest relation with the following properties for any c_1, c_2 containing no `continue`.

$$
\begin{array}{l}
(\forall c, \kappa.(c, \kappa) \sim_2 (c, \kappa)) \qquad\qquad\qquad \wedge \\
\left(\forall \kappa. \left(\begin{array}{l}
(\texttt{for(;;}c_2\texttt{)}\ c_1, \kappa) \sim_2 (\texttt{for(;;skip)}\ c_1\,; c_2, \kappa) \\
\wedge\big((c_1\,; \texttt{continue}, \mathrm{KLoop}_1(c_1, c_2)\cdot\kappa) \sim_2 (c_1, c_2\cdot\texttt{continue}\cdot\mathrm{KLoop}_1(c_1\,; c_2, \texttt{skip})\cdot\kappa)\big) \\
\wedge \left(\begin{array}{l}
\forall c_0, \kappa_0.(c_0, \kappa_0) \text{ has no continue} \Rightarrow \\
(c_0, \kappa_0\cdot\texttt{continue}\cdot\mathrm{KLoop}_1(c_1, c_2)\cdot\kappa) \sim_2 \\
\quad (c_0, \kappa_0\cdot c_2\cdot\texttt{continue}\cdot\mathrm{KLoop}_1(c_1\,; c_2, \texttt{skip})\cdot\kappa)
\end{array}\right) \\
\wedge\big((\texttt{continue}, \mathrm{KLoop}_1(c_1, c_2)\cdot\kappa) \sim_2 (c_2, \texttt{continue}\cdot\mathrm{KLoop}_1(c_1\,; c_2, \texttt{skip})\cdot\kappa)\big) \\
\wedge \left(\begin{array}{l}
\forall c_0, \kappa_0.(c_0, \kappa_0) \text{ has no continue} \Rightarrow \\
(c_0, \kappa_0\cdot\mathrm{KLoop}_2(c_1, c_2)\cdot\kappa) \sim_2 (c_0, \kappa_0\cdot\texttt{continue}\cdot\mathrm{KLoop}_1(c_1\,; c_2, \texttt{skip})\cdot\kappa)
\end{array}\right) \\
\wedge\big((\texttt{skip}, \mathrm{KLoop}_2(c_1, c_2)\cdot\kappa) \sim_2 (\texttt{skip}, \mathrm{KLoop}_2(c_1\,; c_2, \texttt{skip})\cdot\kappa)\big) \\
\wedge\big((\texttt{break}, \mathrm{KLoop}_2(c_1, c_2)\cdot\kappa) \sim_2 (\texttt{break}, \mathrm{KLoop}_2(c_1\,; c_2, \texttt{skip})\cdot\kappa)\big)
\end{array}\right)\right)
\end{array}
$$

This seemingly daunting definition is simply a conjunction of all possible intermediate configurations of two programs that need to be related together. Similarly, one can also prove the relation $\sim_2$ to be a simulation. We simply prove that in each conjunction branch, after one small-step reduction of the left-hand-side program, we can make zero or finite steps in the right-hand-side program and arrive at another conjunction branch in this relation. For example, in the first branch,

$$(\texttt{for(;;}c_2\texttt{)}\ c_1, \kappa) \sim_2 (\texttt{for(;;skip)}\ c_1\,; c_2, \kappa)$$

after one step of $(\texttt{for(;;}c_2\texttt{)}\ c_1, \kappa)$, we take three steps of $(\texttt{for(;;skip)}\ c_1\,; c_2, \kappa)$ and arrive at the second branch

$$(c_1\,; \texttt{continue}, \mathrm{KLoop}_1(c_1, c_2)\cdot\kappa) \sim_2 (c_1, c_2\cdot\texttt{continue}\cdot\mathrm{KLoop}_1(c_1\,; c_2, \texttt{skip})\cdot\kappa).$$

This is just a simple proof that can be automated through small-step symbolic execution, while the complicated induction proofs are all hidden in the simulation lemma. Then, by Lemma 12, we prove LOOP- NOCONTINUE.

In fact, the construction of these simulation relations $\sim_2$ can guide the proof using pure refinement. All conjunction branches will correspond to some intermediate proofs using definition 3. Strictly speaking, there is no obvious advantages using either approach, but we believe the simulation approach is more suitable for proving small-step properties by thinking in small-step semantics. It also helps proofs of other extended rules as we will see soon.

Proving Structural Rules.
We prove NOCONTINUE by showing the correctness of Lemma 13.

Lemma 13 *For all $\sigma, c, \kappa, Q, R_{brk}, R_{con}$, and R'_{con}, if $\sigma \vDash \mathit{WP}\,(c, \kappa)\{Q, [R_{brk}, R_{con}]\}$ and c and κ contains no* `continue`*, then $\sigma \vDash \mathit{WP}\,(c, \kappa)\{Q, [R_{brk}, R'_{con}]\}$.*

Proof We first prove in Lemma 14 that command reduction preserves the no-continue property, which is obvious by discussing all constructors of the small-step semantics.

Lemma 14 *For all $c, c', \kappa, \kappa', \sigma$, and σ', if c and κ contains no* `continue` *and $(c, \kappa, \sigma) \rightarrow_c (c', \kappa', \sigma')$, then c and κ contains no* `continue`.

We then induct over the number of steps (c, κ) takes to terminate.

- For the base case where (c, κ) is a terminal, the conclusion is obvious.
- For the case where (c, κ, σ) can step into (c', κ', σ'), we forward the execution of (c, κ, σ) in both weakest preconditions and only need to prove

$$\sigma' \vDash \mathrm{WP}\,(c', \kappa')\{Q, [R_{\mathrm{brk}}, R_{\mathrm{con}}]\} \text{ implies } \sigma' \vDash \mathrm{WP}\,(c', \kappa')\{Q, [R_{\mathrm{brk}}, R'_{\mathrm{con}}]\}$$

  which follows from the induction hypothesis and Lemma 14.

□

Proving Inversion Rules
We prove SEQ- INV, a representative inversion rule, by Lemma 15, where the simulation lemma helps again.

Lemma 15 *For all c_1, c_2, Q, and $\vec{R}$, there exists Q' such that for all σ, (1) if $\sigma \vDash \mathit{WP}\,(c_1\,;c_2, \epsilon)\{Q, [\vec{R}]\}$ then $\sigma \vDash \mathit{WP}\,(c_1, \epsilon)\{Q', [\vec{R}]\}$, and (2) if $\sigma \vDash Q'$ then $\sigma \vDash \mathit{WP}\,(c_2, \epsilon)\{Q, [\vec{R}]\}$.*

Proof To prove SEQ- INV, we need to find an intermediate assertion Q', for which we use $\mathrm{WP}\,(c_2, \epsilon)\{Q, [\vec{R}]\}$. This immediately gives us (2) and we can prove (1) following the scheme proposed in Fig. 11, where BIND- INV is the inversion of bind rule. This is true because we know when c_1 exits by a break or a continue, we will skip $c_2 \cdot \epsilon$ and reach post-conditions in $\vec{R}$; when c_1 exits normally, we will step into $c_2 \cdot \epsilon$ and the weakest precondition for it should be satisfied. □

5.4 Continuation-Based Shallow Embedding

Proving Transformation Rules.
Similar to weakest-precondition-based embedding, we may use the simulation as the "semantic similarity" with the simulation lemma (Lemma 16) that lifts a simulation relation to relations between guard predicates.

$$\begin{tikzcd}
\mathsf{WP}\,(c_1;;c_2,\epsilon)\{Q,[\vec{R}]\} \arrow[r, dashed] \arrow[d, squiggly, "\text{simulation}"'] & \mathsf{WP}\,(c_1,\epsilon)\{\mathsf{WP}\,(c_2,\epsilon)\{Q,[\vec{R}]\},[\vec{R}]\} \\
\mathsf{WP}\,(c_1,c_2\cdot\epsilon)\{Q,[\vec{R}]\} \arrow[r, Rightarrow, "\text{BIND-INV}"] & \mathsf{WP}\,(c_1,\epsilon)\{\mathsf{WP}\,(\texttt{skip},c_2\cdot\epsilon)\{Q,[\vec{R}]\},[\vec{R}]\} \arrow[u, squiggly, "\text{simulation}"']
\end{tikzcd}$$

Fig. 11 Proof scheme of SEQ- INV with weakest precondition

Here, we use another definition of the simulation relation. We add the disjunction in parentheses in definition 4, which asserts that when the *source* terminates, the *target* will no longer change the program state and will either terminate in zero or finite steps, or stuck in an infinite loop. Because under the continuation-based embedding, we only require the simulation lemma to provide the derivation between the safety of two programs, i.e., they can proceed without error. The simulation does not need to simulate their terminations, which they may never reach. This is an important property which we use in proofs.

Lemma 16 (Guard Simulation) *For all $c_1, c_2, \kappa_1, \kappa_2$, and P, if $\{P\}(c_1, \kappa_1)$ and we have the simulation $(c_2, \kappa_2) \sim (c_1, \kappa_1)$, then $\{P\}(c_2, \kappa_2)$.*

Proof The proof is similar to the one for Lemma 12 by induction over the reduction of (c_2, κ_2). The only difference is the *termination* case, where we no longer require any information from (c_1, κ_1) since the conclusion holds by definition.

With Lemma 16, we can easily prove IF- SEQ and LOOP- NOCONTINUE. The proof idea is similar to the one in Sect. 5.3 and we omit it here.

Proving Structural Rules

Lemma 17 is the key step for proving NOCONTINUE in continuation-based embedding, where complex constructions of continuations are involved.

Lemma 17 *For all c, κ_0, P, Q, R_{brk}, and R_{con}, we have $\{Q\}(\texttt{skip}, \kappa_0)$ and $\{R_{brk}\}(\texttt{break}, \kappa_0)$ imply $\{P\}(c, \kappa_0)$, if (1) c has no* `continue`*; and (2) for all κ, $\{Q\}(\texttt{skip}, \kappa)$ and $\{R_{brk}\}(\texttt{break}, \kappa)$ and $\{R_{con}\}(\texttt{continue}, \kappa)$ imply $\{P\}(c, \kappa)$.*

Proof Here, we set R'_{con} in NOCONTINUE to $\bot$, which can derive any other R'_{con} by HOARE-CONSEQUENCE rule. As a result, we only know the execution of κ_0 is guarded by Q and R_{brk} but not R_{con}. However, we can construct a new κ from κ_0 such that the behaviors of two continuations are "similar" when entering by normal exit and break exit, but κ "terminates" immediately when entering by continue exit to guarantee $\{R_{\text{con}}\}(\texttt{continue}, \kappa)$. Also, since c has no `continue`, we know the behaviors of (c, κ_0) and (c, κ) are "similar" so that they can simulate.

For such construction to be viable, we first need Lemma 18 to enable conversion from continuations to commands, whose proof is trivial and omitted.

Lemma 18 *For any continuation κ, we can construct a command c_κ, such that $(\texttt{skip}, \kappa) \sim (c_\kappa, \epsilon)$, by chaining expressions corresponds to each level in κ using sequencing commands and virtual loops*[13].

We then show the existence of such construction from κ_0 to κ.

[13] Virtual loops take the form of $c_3{::}\texttt{for}_i(;\,;c_2)c_1$, where c_3 is the command remaining in the previous iteration and $i = 1, 2$. Its semantics is defined by the reduction $(c_3{::}\texttt{for}_i(;\,;c_2)c_1, \kappa, \sigma) \rightarrow_c (c_3, \text{KLoop}_i(c_1, c_2)\cdot\kappa, \sigma)$. Assume c_κ is constructed from κ, we transform $\kappa \cdot \text{KLoop}_i(c_1, c_2)$ into $c_\kappa{::}\texttt{for}_i(;\,;c_2)c_1$.

We define program $\texttt{dead} \triangleq \texttt{for(;;skip) skip}$. It will always cause $c\,;\texttt{dead}$ stuck in an infinite loop after the execution of c and it is always safe to execute $\texttt{dead}$ from any program state.

We assume κ_0 takes the form of

$$\underbrace{K_A}_{A} \cdot \underbrace{\text{KLoop}_i(c_1, c_2)}_{B} \cdot \underbrace{K_C}_{C}$$

where B is the first (innermost) loop continuation in κ_0. The case where κ_0 contains no loop, i.e., B and C is empty ϵ, is trivial and is not discussed here. The normal entry into it executes all ABC; the break entry into it executes C only; the continue entry into it executes BC. We use c_{AB} to denote the command transformed from continuation AB by Lemma 18. Now, we construct[14]

$$\kappa \triangleq \underbrace{c_{AB} \cdot \texttt{break}}_{A'} \cdot \underbrace{\text{KLoop}_1(\texttt{skip}, \texttt{dead})}_{B'} \cdot \underbrace{K_C}_{C'}$$

The *normal* entry will execute c_{AB} and then skip the loop in B' to execute K_C, i.e., it will execute all continuations ABC in κ_0; the *break* entry will skip A'[15] and is scoped by the loop in B' and enter K_C through normal entry, which is equivalent to first skipping loop in B then executing K_C; the *continue* entry will skip A' and stuck in a dead loop in B'. As a result, $\{Q\}(\texttt{skip}, \kappa_0)$ will imply $\{Q\}(\texttt{skip}, \kappa)$, $\{Q_{\text{brk}}\}(\texttt{break}, \kappa_0)$ will imply $\{Q_{\text{brk}}\}(\texttt{continue}, \kappa)$, and $\{Q_{\text{con}}\}(\texttt{continue}, \kappa)$ unconditionally holds because a dead loop always make progress.

By premise, we have $\{P\}(c, \kappa)$. Notice that (c, κ) simulates (c, κ_0) because κ simulates κ_0 when c exit normally or by break, but c can only exit in these ways instead of continue exit. By Lemma 16, we prove $\{P\}(c, \kappa_0)$. □

Proving Inversion Rules

Similar to the proof of NOCONTINUE, the proof of SEQ- INV (Lemma 19) is also based on constructions of continuations.

Lemma 19 *For all c_1, c_2 and P, if for all κ, $\{Q\}(\texttt{skip}, \kappa)$ and $\{R_{brk}\}(\texttt{break}, \kappa)$ and $\{R_{con}\}(\texttt{continue}, \kappa)$ imply $\{P\}(c_1\,; c_2, \kappa)$, then there exists Q' such that for all κ_0,*

(1) $\{Q'\}(\texttt{skip}, \kappa_0)$ and $\{R_{brk}\}(\texttt{break}, \kappa_0)$ and $\{R_{con}\}(\texttt{continue}, \kappa_0)$ imply $\{P\}(c_1, \kappa_0)$,
(2) $\{Q\}(\texttt{skip}, \kappa_0)$ and $\{R_{brk}\}(\texttt{break}, \kappa_0)$ and $\{R_{con}\}(\texttt{continue}, \kappa_0)$ imply $\{P\}(c_2, \kappa_0)$.

Proof We construct Q' as the strongest post. of c_1: $\sigma \vDash Q'$ iff. $\exists \sigma_0. (c_1, \epsilon, \sigma_0) \rightarrow_c (\texttt{skip}, \epsilon, \sigma)$ and $\sigma_o \vDash P$.

(1) Similar to the proof of NOCONTINUE, we need to construct a κ from κ_0 to utilize the premise by satisfying $\{Q\}(\texttt{skip}, \kappa)$. We take $\kappa \triangleq \texttt{dead} \cdot \kappa_0$. κ simulates κ_0 through break and continue entry and we have $\{R_{\text{brk}}\}(\texttt{break}, \kappa)$ and $\{R_{\text{con}}\}(\texttt{continue}, \kappa)$ by Lemma 16. κ always loops through the normal entry and we have $\{Q\}(\texttt{skip}, \kappa)$. By premise, we have $\{P\}(c_1\,; c_2, \kappa)$. This guarantees c_1 can execute safely from P, and we only need to show that κ_0 can execute safely after c_1.

[14] We abbreviate $\text{KSeq}(c) \cdot \kappa$ as $c \cdot \kappa$.

[15] Although there is a loop in c_{AB}, it is hidden in a sequential continuation and the $\texttt{break}$ will ignore the loop in c_{AB}. Such ill-formed continuation with loop commands in a sequential continuation is only possible in fabricated ones and will not occur in normal small-step reductions.

When c_1 exits through break and continue, $(c_1\,;c_2)\cdot\kappa$ simulates $c_1\cdot\kappa_0$ and we have $\{P\}(c_1,\kappa_0)$; $\{Q'\}(\texttt{skip},\kappa_0)$ tells us when c_1 exits normally, we can keep executing κ_0 safely. Together, we have $\{P\}(c_1,\kappa_0)$ in all cases.

(2) By specializing the premise with κ_0, we can reduce the proof goal to $\{P\}(c_1\,;c_2,\kappa_0)$ implies $\{Q'\}(c_2,\kappa_0)$. This is obvious because $\{P\}(c_1\,;c_2,\kappa_0)$ shows the safety $c_2\cdot\kappa_0$ after we exit c_1 normally, and Q', program states after c_1's normal exit, which apparently guards $c_2\cdot\kappa_0$. □

Summary

In summary, we have proved representatives (Fig. 10) of extended rules in three categories under four different embeddings.

- The deep embedding allows most of the proofs to be simple, but it has a relatively complicated proof for the HOARE- EX rule.
- The big-step-based embedding has relatively simple proofs for all extended rules. There is also a general proof principle for transformation rules under all shallow embeddings: first prove a relationship between two programs' semantics, then lift it to a relationship between two programs' Hoare triple. However, as mentioned in Sect. 3.1 and Sect. 4.1, the difficulty to extend it with more complex semantic features like concurrency limits its applications.
- In the weakest-precondition-based embedding, proofs are of medium difficulty. We incorporate the general proof principle and the simulation relation to produce more organized proofs. The simulation relation is straightforward to construct and is used throughout all extended rule's proof in this embedding, but it could be tedious to define when the program is complicated.
- In the continuation-based embedding, complexities for transformation rules' proofs are still acceptable with the general proof principle, but is very difficult for other extended rules. We need complex constructions for new continuations to utilize information in the premise.

Using deep embedding appears to have more benefits with minimum limits on the logic's application. Its extended rules' proofs are mainly at the logic level, instead of the semantic level. Therefore, we do not need to worry about simulations and defining continuations in this embedding. The only problem is the HOARE- EX rule, which we have proved in this section and the proof can be maintained when extending the language and logic, as long as there exists exactly one primary rule corresponds to it for each program constructor as mentioned in Sect. 5.1.

6 From Shallow Embedding to Deep Embedding

In Sect. 5, we have seen proofs for extended proof rules under different program logic embeddings. The proofs under shallow embeddings (except the big-step-based one) are relatively complicated for certain extended rules, e.g., IF- SEQ and NOCONTINUE. Proofs under both the weakest-precondition-based one and the continuation-based one involve the notion of simulation. Proofs under the continuation-based one even require complex constructions of continuations. Although it may be possible to formalize these complicated proofs in real verification tools, we propose an easier way to work around them by building a deeply embedded logic above the existing shallowly embedded logic and proving these extended rules under the deeply embedded one.

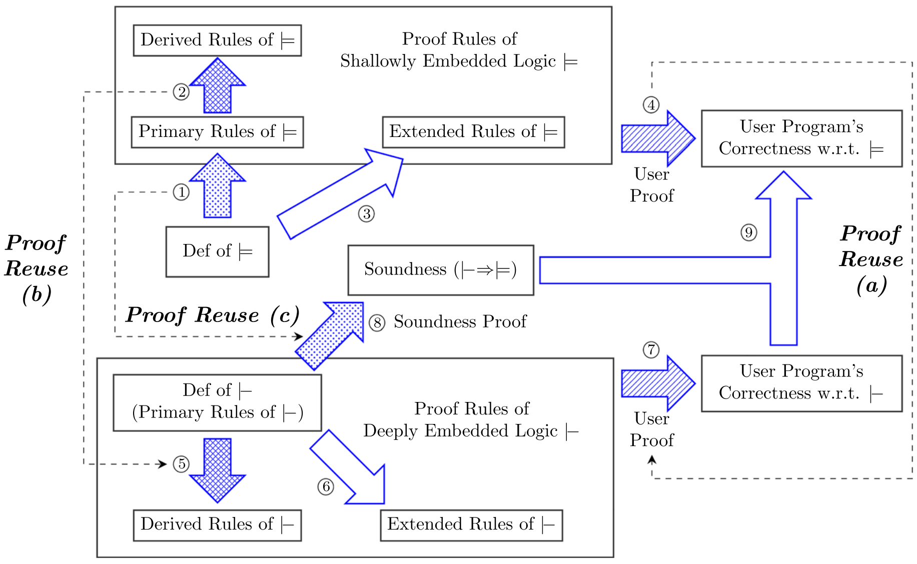

Fig. 12 The Framework for Building Deep Embedding from Shallow Embedding. (i) Proofs are represented by arrows labeled with numbers. (ii) Proofs of arrows with matching textures can be reused in place of the other. (iii) Symbol ⊢ represents the deeply embedded logic, while ⊨ represents the shallowly embedded logic. (iv) The proof ⑥ is easier than proof ③

Figure 12 presents our framework for building a deeply embedded logic by reusing most of the proofs from an existing shallowly embedded one. The existing shallow embedding contains the following:

- The definition of the shallow embedding $\models \{P\}c\{Q, \vec{R}\}$.
- A set of primary rules and their proofs ① directly based on the definition.
- A set of derived rules and their proofs ② built on primary rules.
- A set of extended rules and their proofs ③ directly based on the definition.
- User's proof ④ of their programs using only[16] rules of these three types.

The deeply embedded verification tool we want needs to provide the followings:

- The definition of the embedding, i.e., the set of primary rules for proving $\vdash \{P\}c\{Q, \vec{R}\}$.
- A set of derived rules and a set of extended rules along with their proofs ⑤⑥ directly built on primary rules.
- User's proof ⑦ of their own programs using rules of these three types.
- To guarantee the correctness of the deep embedding, a soundness theorem and its proof ⑧ is required.

We use the same sets of proof rules in the deep embedding as the shallow one. The primary rules, derived rules, and extended rules in the shallow embedding and the deep embedding have the same forms. In this way, many proofs in the shallow embedding side can be reused to assist the construction of the deep embedding as indicated in Fig. 12.

Proof reuse (a). Since all proof rules in two embedding have same forms, users can upgrade their proofs in the shallow embedding ④ to those in the deep one ⑦ with only

[16] The embedding's definition is not exposed to users as part of its interface. The user's proof only consists of applications of these proof rules and proofs irrelevant to the program logic.

one change in proofs. They only need to replace occurrences of proof rules with their counterparts in the deep one. The new proof establishes the program correctness as deeply embedded Hoare triples.

Proof reuse (b). As mentioned in Sect. 1, derived rules are directly derived from primary rules with trivial proofs. Since primary rules in two embeddings have same forms, derived rules' proofs ② and ⑤ are in the same situation as case ***(a)*** and we can reuse them.

Proof reuse (c). The most important part in a deeply embedded logic is its soundness proof ⑧, which guarantees the logic's correctness. However, we can construct the soundness proof for free by reusing existing primary rules' proofs ① in the shallow embedding. To prove the soundness, we start by induction over the proof tree and obtain subgoals that are identical to shallowly embedded primary rules, which are already proved.

Moreover, given the soundness and user program correctness in the deep embedding, we can also derive program correctness in the shallow embedding ⑨, if it is of user's desire. It is a trivial instantiation of the soundness theorem on deeply embedded Hoare triples of user programs, where both are generated by reusing proofs in the shallow embedding.

If the original shallowly embedded logic does prove some extended rules, we may also reuse them. However, as we have shown in Sect. 5, their proofs ③ in the shallow embedding are often complex constructions, while in the deep embedding, their proofs ⑥ are mostly simple inductions over the proof tree. In many cases, for extended rules not proved on the shallow embedding side, we can easily prove them on the lifted deep embedding side.

In conclusion, building deeply embedded verification tools from an existing shallowly embedded one mainly involves simple reuses of existing proofs and simplified proofs of extended rules. Existing user's proofs can also be easily upgraded to the deeply embedded version.

7 Choices of Primary Rules in Deep Embedding

At the beginning of Sect. 5.1, we have fixed our deeply embedded logic with only primary rules from Fig. 2. In fact, we may choose deep embeddings with other admitted primary rules. For example, we may choose FRAME rule as a primary rule as discussed in Sect. 8.1.2. In this section, we want to discuss the choices of making extended rules as primary rules.

Choosing HOARE- EX ***as a Primary Rule.***

In Sect. 5, we notice that the proof of HOARE- EX is simpler in shallow embeddings than its proof in the deep embedding. It seems a good choice to make it a primary rule and use the technique in Sect. 6 to lift its simple shallow embedding proof to the deep embedding's soundness proof. However, we still prefer not to choose it as a primary proof rule in the deeply embedded logic and prove it as an extended rule. This is also the design choice in our deeply embedded VST. Adding HOARE- EX as a primary rule causes problems.

Adding more primary rules to the proof system would invalidate the original proofs of other extended rules. In deep embedding, we mainly use induction over proof trees when proving extended rules, and now we need to discuss one more case in such induction where the Hoare triple is constructed by HOARE- EX. Taking SEQ- INV as an example, the extra case is that $\vdash \{P\}c_1; c_2\{Q, [\vec{R}]\}$ is constructed by HOARE- EX, i.e., P must takes the form of $\exists x : T. P'(x)$. The induction hypothesis is that for any x of type T we can find an S such that $\vdash \{P'(x)\}c_1\{S, [\vec{R}]\}$ and $\vdash \{S\}c_2\{Q, [\vec{R}]\}$. We then need to use the axiom of choice (AC) to complete the proof: we need to find a function S' from T to assertions, such that for any x of type T, $\vdash \{P'(x)\}c_1\{S'(x), [\vec{R}]\}$ and $\vdash \{S'(x)\}c_2\{Q, [\vec{R}]\}$. It is controversial whether we should accept AC when we prove meta-theorems. The original VST does not need AC. But to complete this proof in Coq, we actually need the computational version of AC (`FunctionalChoice_on` in Coq[17]). It allows us to construct the S' in the above proof.

```
Definition RelationalChoice_on :=
 forall R:A→ B→ Prop,
  (forall x : A, exists y : B, R x y) →
  (exists R' : A→ B→ Prop, subrelation R' R ∧ forall x, exists! y, R' x y).

Definition FunctionalChoice_on :=
 forall R:A→ B→ Prop,
  (forall x : A, exists y : B, R x y) →
  (exists f : A→ B, forall x : A, R x (f x)).
```

To prevent such problems when we want more extended rule, we treat HOARE- EX as an extended rule and prove it as a meta-theorem.

Choosing Transformation Rules as Primary Rules.
Transformation rules cannot treated as primary rules either. Transformation rules and primary rules for the same primitive bring non-determinism to the proof tree of a program, which invalidates our previous proof of inversion rules.

For example, if we admit LOOP- NOCONTINUE as a primary rule, then in the proof of LOOP-INV, we need to consider one more induction case: given $\vdash \{P\}$`for(;;skip)` c_1 ; $c_2\{Q, [\vec{R}]\}$, we need to show that there exists I_1 and I_2 such that $\vdash$ $\{I_1\}c_1\{I_2, [Q, I_2]\}$ and $\vdash$ $\{I_2\}c_2\{I_1, [Q, \bot]\}$ and $P \vdash I_1$. Different from the case in Sect. 5.2, the induction hypothesis cannot extract any information from the premise. The loop must take the form of `for(;;`c_2`)` c_1 to apply the induction hypothesis. As a result, we fail to prove LOOP- INV.

Moreover, admitting transformation rules as primary rules acquires developers to give their soundness proof in some shallow embedding. As we have seen in Sect. 5, their proofs under weakest-precondition-based embedding and continuation-based embedding is not as simple as other primary rules' proofs.

Choosing other Extended Rules as Primary Rules.
Inversion rules cannot be chosen as primary rules. Conclusions of primary rules must take the form of Hoare triples, while those of inversion rules are often propositions led by meta-language's existential quantifier. We cannot define the provable relation for these propositions.

Other structural rules like LOOP- NOCONTINUE are preferred not chosen as primary rules for the same reason as transformation rules. In Sect. 8.1 and Sect. 8.1.2, we introduce two more structural rules, the FRAME rule and the HYPO- FRAME rule, which are preferred not chosen as primary rules as well.

As a result, our current choice of primary rules (compositional rules, the consequence rule, and singleton command rules) is the most suitable one for instantiating a deeply embedded verification tool.

8 Discussion: Extensions of Hoare Logic

In this section, we discus various extensions of the Hoare logic discussed in previous sections and their potential influence on the logic embedding. Specifically, we will first consider program logics with load and store to discuss some basics of separation logic embeddings.

[17] https://coq.inria.fr/doc/v8.9/stdlib/Coq.Logic.ChoiceFacts.html.

While-CF with Memory Operations

$$c \in \text{command} := \cdots \mid x = \text{new}(v) \mid x = [y] \mid [x] = y$$

Separation Logic Proof Rules

HOARE-ALLOC

$$\vdash \{\texttt{emp}\} x = \text{new}(v) \{\exists l.\, [\![x]\!] = l \wedge l \mapsto v\}$$

HOARE-LOAD

$$\vdash \{l \mapsto v \wedge [\![y]\!] = l\} \;\; x = [y] \;\; \{l \mapsto v \wedge [\![x]\!] = v \wedge [\![y]\!] = l, [\bot, \bot]\}$$

HOARE-STORE

$$\vdash \{l \mapsto v \wedge [\![x]\!] = l \wedge [\![y]\!] = u\} \;\; [x] = y \;\; \{l \mapsto u \wedge [\![x]\!] = l \wedge [\![y]\!] = u, [\bot, \bot]\}$$

FRAME

$$\frac{\begin{array}{c} c \text{ does not modify program variables freely occurring in F} \\ \vdash \{P\} c \{Q, [Q_{\text{brk}}, Q_{\text{con}}]\} \end{array}}{\vdash \{P * F\} c \{Q * F, [Q_{\text{brk}} * F, Q_{\text{con}} * F]\}}$$

Fig. 13 Additional Proof Rules for Separation Logics

Then, we introduce rules related to procedure calls above it and further discuss embeddings for the logic with procedure calls. We will also discuss various features of the program logic and assertion language like total correctness, non-determinism, and impredicative assertions, along with the soundness proof technique mentioned in Sect. 3.3.

8.1 Embeddings of Separation Logic

When reasoning about memory operations, it is useful to extend the Hoare logic to a separation logic [37]. A separation logic adds a logic connective, "$*$", separating conjunction, to the assertion language. The assertion $P * Q$ asserts two disjoint pieces of memory satisfying P and Q respectively. Typically, a separation logic adds proof rules for memory-related assignments (rules for load and store) and the FRAME rule to Hoare logic. Also, when a function call is involved in a programming language, separation logic allows verifiers to specify only the local effects of callee functions, while proving how the caller uses this function to modify the global memory.

Figure 13 extends our previous toy WhileCF language with the following memory operations.

- Allocation, $x = \text{new}(v)$, where x is a program variable that stores the location storing the value v.
- Simple load, $x = [y]$, where x and y are program variables, and this command loads the value from the location with address y into variable x.
- Simple store, $[x] = y$, where x and y are program variables, and this command stores the value of y into the location with address y.

In addition, we assume that there is no memory load in an expression. Every memory operation is performed as a load or store command. This is similar to the Clight program in CompCert [23] and the C program in VST-Floyd [9].

To verify these memory operations, Fig. 13 adds three standard primary rules, HOARE-ALLOC, HOARE- LOAD and HOARE- STORE, to our previous logic in Sect. 2. The FRAME rule is the main focus of this section.

8.1.1 Frame Rule and More Potential Shallow Embeddings

The FRAME rule allows prover to remove unused separating conjuncts (assertions in separation logics) of memory from both pre-/post-conditions and prove the specification with remaining memories. The derivability of the FRAME rule depends on whether the programming language satisfies a property named the "frame property". Gotsman et al. [16] proved that if the operational semantics satisfies the frame property, then the FRAME rule is sound. The following hypotheses state the frame property via a big-step semantics and via a small-step semantics. We use $\sigma' \oplus \sigma$ to represent the disjoint union of two pieces of memory, σ' and σ.

Hypothesis 20 **(Frame property for a big-step semantics)** For any c, ek, σ_1, σ_2', σ,

- if $(c, \sigma_1 \oplus \sigma) \Uparrow$, then $(c, \sigma_1) \Uparrow$;
- if $(c, \sigma_1 \oplus \sigma) \Downarrow (\mathrm{ek}, \sigma_2')$, then either $(c, \sigma_1) \Uparrow$ or $\sigma_2' = \sigma_2 \oplus \sigma$ for some σ_2 and $(c, \sigma_1) \Downarrow (\mathrm{ek}, \sigma_2)$.

Hypothesis 21 **(Frame property for a small-step semantics)** For any $c_1, c_2, \kappa_1, \kappa_2, \sigma_1, \sigma_2'$, σ,

- if $(c_1, \kappa_1, \sigma_1 \oplus \sigma) \nrightarrow_c$, then $(c_1, \kappa_1, \sigma_1) \nrightarrow_c$;
- if $(c_1, \kappa_1, \sigma_1 \oplus \sigma) \rightarrow_c (c_2, \kappa_2, \sigma_2')$, then either $(c_1, \kappa_1, \sigma_1) \nrightarrow_c$ or $\sigma_2' = \sigma_2 \oplus \sigma$ for some σ_2 and $(c_1, \kappa_1, \sigma_1) \rightarrow_c (c_2, \kappa_2, \sigma_2)$.

These two properties are true for the WhileCF language extended with memory operations as well as for most of reasonable languages (including realistic languages like C). The above conclusion by Gotsman et al. [16] holds for $\vDash_b$, $\vDash_w$, and $\vDash_c$ (respectively stand for the validity definition under big-step-based embedding, weakest-precondition-based embedding, and continuation-based embedding). For example, Theorem 22 proves the FRAME rule in the big-step-based embedding.

Theorem 22 *If the Hypothesis 20 is true, the* FRAME *rule holds for the big-step-based embedding defined in Sect. 4.1.*

Proof To prove $\vDash_\mathrm{b} \{P * F\}c\{Q * F, [R_\mathrm{brk} * F, R_\mathrm{con} * F]\}$, we need to show that for any $\sigma \vDash P * F$ (1) $\neg\,(c, \sigma) \Uparrow$, and (2) for all ek, σ_2, if $(c, \sigma) \Downarrow (\mathrm{ek}, \sigma_2)$, σ_2 satisfies corresponding post-conditions with the frame F according to ek.

According to the premise $\vDash_\mathrm{b} \{P\}c\{Q, [R_\mathrm{brk}, R_\mathrm{con}]\}$, we know for any $\sigma_1 \vDash P$ (i) $\neg\,(c, \sigma_1) \Uparrow$, and (ii) for all ek, σ_2, if $(c, \sigma_1) \Downarrow (\mathrm{ek}, \sigma_2)$, σ_2 satisfies corresponding post-conditions without the frame F according to ek.

We can split σ, according to $\sigma \vDash P * F$, into σ_1 and σ' satisfying

$$\sigma = \sigma_1 \oplus \sigma' \quad \text{and} \quad \sigma_1 \vDash P \quad \text{and} \quad \sigma' \vDash F.$$

By (i), we know $\neg\,(c, \sigma_1) \Uparrow$. Then, by Hypothesis 20, we have $\neg\,(c, \sigma_1 \oplus \sigma') \Uparrow$ because otherwise we have a contradiction and we prove (1).

To prove (2), by Hypothesis 20, for any ek, σ_2 with $(c, \sigma_1 \oplus \sigma') \Downarrow (\mathrm{ek}, \sigma_2)$, we know that

- either $(c, \sigma_1) \Uparrow$, which contradicts with the premise (i) and finishes the proof;

- or there exists σ_2' with $\sigma_2 = \sigma_2' \oplus \sigma'$ and $(c, \sigma_1) \Downarrow (\text{ek}, \sigma_2')$. By (ii), we know σ_2' satisfies corresponding post-conditions without the frame F according to ek (i.e., $\sigma_2' \vDash Q$, $\sigma_2' \vDash R_{\text{brk}}$, or $\sigma_2' \vDash R_{\text{con}}$). Since $\sigma' \Vdash F$, we know $\sigma_2 = \sigma_2' \oplus \sigma'$ satisfies corresponding post-conditions with the frame F according to ek (i.e., $\sigma_2 \vDash Q * F$, $\sigma_2 \vDash R_{\text{brk}} * F$, or $\sigma_2 \vDash R_{\text{con}} * F$), which finishes the proof.

□

Interestingly, a different style of shallow embeddings for separation logic [5] are widely used and the FRAME rule will hold intrinsically. The FRAME rule in this formalization is called the "baked-in frame rule" in some literatures. For any shallow embedding $\vDash_x$ among $\vDash_b, \vDash_w$, and $\vDash_c$, it defines the Hoare triple as

$$\vDash_{\text{sep}-x} \{P\}c\{Q, [R_{\text{brk}}, R_{\text{con}}]\} \text{ iff. for any } F, \vDash_x \{P * F\}c\{Q * F, [R_{\text{brk}} * F, R_{\text{con}} * F]\}.$$

Taking continuation-based shallow embedding for example, we can use another definition: $\vDash_{\text{sep}-c} \{P\}c\{Q, [R_{\text{brk}}, R_{\text{con}}]\}$ iff. for arbitrary continuation κ and *arbitrary frame* F (also a separation logic assertion),

$$\text{if} \quad \left\{ \begin{array}{l} \{Q * F\}(\texttt{skip}, \kappa) \\ \{Q_{\text{brk}} * F\}(\texttt{break}, \kappa) \\ \{Q_{\text{con}} * F\}(\texttt{continue}, \kappa) \end{array} \right. \quad \text{then} \quad \{P * F\}(c, \kappa).$$

Above is also shallowly embedded VST's definition of Hoare triples. Under such embedding, we can easily verify the FRAME rule without using the frame property of small-step operational semantics. The premise $\vDash_{\text{sep}-c}$ $\{P\}c\{Q, [R_{\text{brk}}, R_{\text{con}}]\}$ means that for arbitrary frame F'',

$$\vDash_c \{P * F''\}c\{Q * F'', [R_{\text{brk}} * F'', R_{\text{con}} * F'']\}.$$

Let $F'' = F * F'$. Then, we know $\vDash_c \{(P * F) * F'\}c\{(Q * F) * F', [(R_{\text{brk}} * F) * F', (R_{\text{con}} * F) * F']\}$ for any F and F', which exactly states the conclusion of frame:

$$\vDash_{\text{sep}-c} \{P * F\}c\{Q * F\}.$$

Although such sep$-x$ embedding does not invalidate any primary rules or extended rules we have mentioned so far, it does make other proofs more complicated since we need to deals with this extra frame F. In comparison, both embedding styles are feasible. The original one makes proofs of other primary rules and extended rules relatively concise, while the sep$-x$ version makes the proof of the frame rule simple.

To summarize, in a shallow embedding, we can either directly prove the FRAME rule if Hypothesis 20 and Hypothesis 21 are true, or use the baked-in frame rule and reprove other rules using the new shallow embedding $\vDash_{\text{sep-x}}$. It does not matter whether the FRAME rule is a primary rule or an extended rule in a shallow embedding because both types of proof rules are just lemmas.

8.1.2 Frame Rule and Deep Embeddings

In deeply embedded program logics, we can either choose to make FRAME a primary rule or treat it as an extended rule.

The former option means that the separation logic's soundness is based on the fact that FRAME preserves Hoare triple's validity. This proof is very similar to the verification of

FRAME in shallowly embedded logics. When developing a deeply embedded logic based on a shallowly embedded logic (see Sect. 6), one could directly reuse the proof of the FRAME rule in the shallow embedding. However, as discussed in previous sections, adding one more primary rule to the proof system may invalidate the other proofs because every case analysis and induction on proof trees will have one more branch to prove. In the case of FRAME rule, although extended rules we have discussed may not be invalidated by this extra case, such a choice would cause more uncertainty when we want to add more extended rules where this extra case may be unprovable. Keeping the proof system concise with only singleton rules, compositional rules, and the consequence rule will make the logic more extensible and thus we give up this design choice and prove the frame rule as an extended rule.

Nevertheless, we are not yet able to prove the FRAME rule with our previous deeply embedded program logic by simple induction over the proof tree of $\{P\}c\{Q\}$. Two base cases corresponding to HOARE- LOAD and HOARE- STORE are not provable. We should allow these rules to be compatible with extra frames. For example, we use HOARE- LOAD- FRAME to replace HOARE- LOAD.

HOARE- LOAD- FRAME
$$\vdash \{F * l \mapsto v \wedge [\![y]\!] = l\}\ x = [y]\ \{F * l \mapsto v \wedge [\![x]\!] = v \wedge [\![y]\!] = l, [\bot, \bot]\}$$

Then, we can easily prove the FRAME rule by induction over the proof tree. Similarly, for a language with more commands, we only need to guarantee proof rules for memory-related operations are compatible with extra frames.

To summarize, there are also two ways to formalize the FRAME rule in a deep embedding: either directly admit it as a primary rule, or bake the FRAME rule into singleton rules for memory operations. Both require admitting the FRAME rule as part of the primary rules but the second one makes maintaining existing extended rules' proofs easier.

Procedure Calls and Hypothetical Frame Rule

In this section, we further extend the language and logic in Fig. 13 with procedure calls. Figure 14 adds the command for procedure calls, call $f()$, where procedure calls have no arguments or return value. Function arguments and return values are written to specific locations to be transferred between the caller and the callee.

Different from the logic in previous sections, the judgement of the logic in Fig. 14 takes the form of $\Delta \vdash \{P\}c\{Q, [\vec{R}]\}$. Here, Δ is a list of function hypotheses (a.k.a. the type-context in VST), containing a list of Hoare triples $\forall X_i.\{P_i\}k_i\{Q_i\}$ specifying the pre/post-condition of the procedure with identifier $k_i \in$ FuncID and logical variables X_i the triple depends on. With the function hypothesis, HOARE- CALL can determine the effect of procedure call by looking up the callee's specification in Δ.

Shallow embeddings of judgement involving procedure calls [34] are different from those in Sect. 4. Taking the denotational semantics based shallow embedding $\vDash_b$ for example, we introduce η in (17), a mapping from procedures k_i to their denotations, to avoid circularity.

$$\eta \in \text{FuncID} \to \text{state} \to \text{state} \to \text{exit\_kind} \to \text{Prop} \tag{17}$$

We use $\eta(k_i)$ to lookup the procedure's denotation and $c[\![\eta]\!]$ to represent the program after substituting call to procedures by their denotations. The shallow embedding $\Delta \vDash_b \{P\}c\{Q, [\vec{R}]\}$ is defined as: for any $(\forall X_i.\{P_i[X_i]\}k_i\{Q_i[X_i]\}) \in \Delta$, X, and η, $\vDash_b \{P_i[X]\}\eta(k_i)\{Q_i[X], [\bot, \bot]\}$ is true, then $\vDash_b \{P\}c[\![\eta]\!]\{Q, [\vec{R}]\}$ is true.

In a logic combining separation logic and procedure call, we can generalize the FRAME rule to the HYPO- FRAME rule in Fig. 14, which was first proposed by O'Hearn *et. al* [34]. This rule allows extending specifications of both the main program c and procedures k_i it invokes

While-CF with Memory Operations and Procedure Calls

$$c \in \text{command} := \cdots \mid x = [y] \mid [x] = y \mid \text{call } f()$$

Procedure Call Proof Rules

$$\text{HOARE-CALL}\ \frac{(\forall X_i.\ \{P_i[X_i]\}k_i\{Q_i[X_i]\}) \in \Delta}{\Delta \vdash \{P_i[X]\}\ \ \text{call } k_i()\ \ \{Q_i[X], [\bot, \bot]\}}$$

$$\text{HYPO-FRAME}\ \frac{\begin{array}{c} c \text{ modifies program variables freely occuring in } R \text{ only through } k_i \\ (\forall X_i.\ \{P_i[X_i]\}k_i\{Q_i[X_i]\}) \in \Delta \iff (\forall X_i.\ \{P_i[X_i] * R\}k_i\{Q_i[X_i] * R\}) \in \Delta' \\ \Delta \vdash \{P\}c\{Q, [Q_{\text{brk}}, Q_{\text{con}}]\} \end{array}}{\Delta' \vdash \{P * R\}c\{Q * R, [Q_{\text{brk}} * R, Q_{\text{con}} * R]\}}$$

Fig. 14 Additional Proof Rules for Separation Logics and Procedure Calls

by composing an extra frame R to their preconditions ($P_i[X_i]$ and P) and postconditions ($Q_i[X_i]$ and Q). It is easily derivable in a deeply embedded logic by induction over the proof tree with modified atomic rules in Sect. 8.1.1, and it is proved for the shallow embedding we introduced in this section by O'Hearn *et. al* [34].

Total Correctness

Throughout the paper, we have been discussing the Hoare logic for partial correctness, where we only require the program to produce no error and its output and ending state to satisfy certain assertions. Another correctness property that researchers care about a program is its total correctness. The total correctness adds one more condition on top of the partial correctness: the execution of the program terminates.

In a simple While language with only assignment, sequential composition, if-statement, and while-loop, the only way to cause non-termination is by executing an infinite loop. A famous approach to prove the termination of a loop is through loop variant [28]. For example, the WHILE- TOTAL rule below uses an expression e as the loop-variant, which evaluates to some nature number. The value of e is guaranteed to decrease and when it reaches 0, the loop terminates. With the WHILE- TOTAL rule and other standard proof rules for other language constructs (e.g., those in Fig. 2), a program logic can reason about the total correctness of the While language.

$$\text{WHILE- TOTAL}\ \frac{\begin{array}{c} I \wedge [\![b]\!] = \texttt{true} \vdash [\![e]\!] > 0 \\ \forall n.\ \vdash_{\text{total}} \{I \wedge [\![b]\!] = \texttt{true} \wedge [\![e]\!] = n\}c\{I \wedge [\![e]\!] < n\} \end{array}}{\vdash_{\text{total}} \{I\}\texttt{while}(b)\ c\{I \wedge [\![b]\!] = \texttt{false}\}}$$

In a deeply embedded program logic, the inversion corresponding to the loop trivially holds. With only the loop proof rule being substituted, other extended rules remain unaffected.

In practice, users may care about how long a program terminates rather than just whether a program terminates. Charguéraud *et. al* [13] formalize a separation logic framework with time credits and use it to verify the amortized complexity of a union-find implementation. They provide a proof rule to unroll a recursive function application, and then use the meta-level induction to prove that the program terminates within a certain number of steps bounded by the time credit. However, this paper does not consider the meta-level induction nor the time credit, and what extended rules they will contribute to the logic is worth future investigation.

Non-determinism

Non-determinism is also a feature researchers want to support in Hoare logic, especially when verifying non-deterministic programs like concurrent programs. This paper's result does not rely on a programming language being deterministic. One may freely introduce language primitives with demonic non-determinism and equip the deeply embedded program logic with corresponding primary rules. For these primary rules, one may prove their corresponding inversion rules, which then ensures the correctness of structural rules as we show in Sect. 5.1. Since only primary rules are added, transformation rules and other inversion rules are unaffected. Therefore, users can still embrace various extended rules with non-determinism involved.

Impredicative Assertions

Impredicative assertions are assertions whose universe quantifiers and existential quantifiers can quantify over assertions variables. These assertion languages usually also admits Hoare triples as assertions. This feature is required for our HOARE- EX proof in Sect. 5.1 to work.

Whether impredicative assertions are available depends on the choice of meta logic. For a shallowly embedded assertion language in Coq, which we use for our proofs in Sect. 5, the formalization of these impredicative assertions is inherent because we can inject meta-logic propositions into assertions. Definitions (18) and (19) show how to define the interpretation (shallow embeddings) of these assertions.

$$\sigma \vDash \exists P_0 . S(P_0) \text{ iff. there exists an assertion } P_0 \text{ s.t. } \sigma \vDash S(P_0) \tag{18}$$

$$\sigma \vDash M \text{ iff. } M \text{ where } M \text{ is a meta-logic proposition (e.g., } \vdash \{P\}c\{Q, [\vec{R}]\}) \tag{19}$$

In general, impredicative quantification and does no interact well with complex inductive definitions. For example, impredicative quantification is absent in Agda. Coq poses constraints on the impredicatively quantified object, while Isabelle/HOL does not allow complex inductive definitions, making impredicativity available in both.

If we would use a deeply embedded assertion language, we need to construct the counterpart of the provable judgement and higher-order quantifiers in the assertion language. This is possible, and defining such a syntax system is not hard. The following definition could be a reasonable candidate:

$$\begin{aligned} x &\in \text{individual-variables-name} \\ A &\in \text{assertion-variable-name} \\ t &::= \ldots \mid x \\ P &::= t = t \mid A \mid \neg P \mid P \wedge P \mid \exists x.\, P \mid \exists A.\, P \mid \vdash \{P\}c\{P, [P, P]\} \end{aligned}$$

Technical problems may appear when defining assertions' interpretation $\sigma \vDash P$. A simple syntactical substitution of the quantified variable for its value will result in a non-decreasing recursive interpretation function. For example, in the definition below, the assertion P' that substitutes A in P may be larger than P, which makes the interpretation function non-terminating.

$$\sigma \vDash \exists A . P \iff \text{there exists assertion } P' \text{ such that } \sigma \vDash P[A/P']$$

Instead of a direct syntactical substitution, we can use an interpretation assignment J to store interpretations of free variables.

$$J \in \text{assertion-variable-name} \to (\text{state} \to \text{Prop})$$

We use the judgement $\sigma, J \vDash P$ to help define the interpretation of an assertion. For higher-ordered existential quantifiers, we choose in the meta-level the interpretation d of the assertion variable A and assign d to A in the assignment J.

$$\sigma, J \vDash \exists A.P \iff \text{there exists } d \in \text{state} \to \text{Prop such that } \sigma, J[A/d] \vDash P$$

The above interpretation makes the assertion P to be potentially an open term with the free variable A occurring in P. When we need to interpret a free assertion variable A on a state σ, we simply query J for A's interpretation function and apply it to σ.

$$\sigma, J \vDash A \iff J[A](\sigma)$$

For these Hoare triple assertions, we can directly interpret them as their validities (shallow embeddings). Taking the big-step-based embedding for example, we simply copy and paste the definition in Sect. 4.1 and add assignment J when interpreting pre-/post-conditions.

$$\sigma, J \vDash (\vdash \{P\}c\{Q, [\vec{R}]\}) \iff \left(\begin{array}{l} \text{for all } \sigma_1, J \vDash P, \neg\,(c, \sigma_1) \Uparrow \\ \text{and for all ek}, \sigma_2, \text{ if } (c, \sigma_1) \Downarrow (\text{ek}, \sigma_2) \\ \quad \text{then ek} = \epsilon \text{ implies } \sigma_2, J \vDash Q \\ \quad \text{and ek} = \text{brk implies } \sigma_2, J \vDash R_{\text{brk}} \\ \quad \text{and ek} = \text{con implies } \sigma_2, J \vDash R_{\text{con}} \end{array}\right)$$

For a closed assertion P, we will interpret it on any assignment because it should not depend on it.

$$\sigma \vDash P \iff P \text{ is closed and } \sigma, J \vDash P \text{ is true for any } J$$

Another approach is to inductively define the interpretation of assertions as a deep embedding, which is proposed by Ni and Shao [31]. For example, the interpretation rule for the existential quantifier is the following.

$$\frac{P' \in \text{assertion} \qquad \sigma \vdash P[A/P']}{\sigma \vdash \exists A.P}$$

This definition only defines the interpretations for assertions that have finite derivation trees. Assertions with unbounded size (brought by higher-ordered quantifiers) will not have a finite derivation tree as well as an interpretation. This deeply embedded interpretation is then proved sound w.r.t. assertions true meaning. Interested readers may refer to their work [31] for more details.

The Soundness Proof Technique

Section 3.3 mentioned a soundness proof technique, where logic developers first prove the logic is sound w.r.t. some auxiliary validity definition $\Vert\!\vDash S$, i.e.,

$$\text{forall } S.\ \vdash S \text{ implies } \Vert\!\vDash S, \tag{20}$$

and then prove this auxiliary validity implies the real validity $\vDash S$, i.e.,

$$\text{forall } S.\ \Vert\!\vDash S \text{ implies } \vDash S. \tag{21}$$

The auxiliary validity $\Vert\!\vDash S$ differs from the real validity $\vdash S$ in that $\Vert\!\vDash S$ usually has extra information about programs execution which makes proving (20) easier, while this information is not available when proving the real validity $\vDash S$ and therefore a lemma (21) that erases this information is required. For example, in Brookes's soundness proof for the concurrent separation logic [7], it first uses thread local enabling $(s, h, A) \xrightarrow[\Gamma]{\lambda} (s', h', A')$ to define the auxiliary validity $\Gamma \vdash \{P\}c\{Q\}$ for their Hoare triples, and then use this to derive the real validity $\vdash \{P\}c\{Q\}$ defined using the true semantics $(s, h, A) \overset{\lambda}{\Rightarrow} (s', h', A')$ of the program running in a machine.

9 Deeply Embedded VST

Besides the program logic for our WhileCF toy language discussed so far, we also implement proposed theories in a real verification projects. We use the scheme in Sect. 6 to lift originally shallowly embedded VST [2] into a deeply embedded VST, where extended rules are easily proved.

We build deeply embedded VST[18] based on the shallowly embedded VST, a verification framework for C programs. It is originally designed to use continuation-based shallow embedding to formalize the Hoare logic. Different from our toy language and toy logic, VST is designed to support named function invocations as well and we demonstrate the details VST adds to the triple.

VST uses the continuation-based shallow embedding with small-step semantics defined as $ge \vdash (c, \kappa, m) \rightarrow (c', \kappa', m')$. Different from our small-step semantics in Sect. 4, it contains a global environment ge that stores external variables and functions. When an external call occurs, it will look up the function body from ge and add it to the continuation κ.

VST's triples take the form of $\Delta \vDash \{P\}c\{Q, [\vec{R}]\}$, where Δ is the type-context containing function specifications. Its formalization in Coq is similar to the continuation-based shallow embedding ($\vDash_{\text{sep}-c}$) discussed in this paper. Specifically, the triple means: if every function obey specifications in Δ, then for all continuations κ and all separation logic frames F,

$$\text{if} \begin{cases} \{Q * F\}(\texttt{skip}, \kappa) \\ \text{and } \{R_{\text{brk}} * F\}(\texttt{break}, \kappa) \\ \text{and } \{R_{\text{con}} * F\}(\texttt{continue}, \kappa) \\ \text{and } \{R_{\text{ret}} * F\}(\texttt{return}, \kappa) \end{cases}, \text{ then } \{P * F\}(c, \kappa).$$

A function f obeys specification $\{P\}f\{Q\}$ in Δ iff. its function body c has $\Delta \vDash \{P\}c\{Q\}$. VST uses step-indexing to resolve the circularity in the definition.

The shallowly embedded VST does not provide extended rules like SEQ- INV and NOCONTINUE, which we prove in Sect. 5.4 and find their proof construction challenging. We then seek to formalize a deeply embedded Hoare logic. As proposed in Sect. 6, we first define the provable predicate for Hoare triple by primary proof rules (atomic rules, compositional rules, and the consequence rule) and then directly use the validity definition and soundness proofs in the original shallowly embedded VST to establish the soundness of our deeply embedded VST. We then prove extended rules in the deep embedding by induction over the proof tree. As discussed in Sects. 6 and 8.1.2, we choose to treat HOARE- EX and FRAME rule as extended rules and prove them along with other extended rules, although they were proved sound in the original shallowly embedded VST. Also, VST did support proof rules like HOARE- LOAD-

[18] Our deeply embedded VST has been integrated into the VST repository (https://github.com/PrincetonUniversity/VST) and we include the version when it was first integrated. It consists of two files, `SeparationLogicAsLogic.v` and `SeparationLogicAsLogicSoundness.v` in the `floyd` folder.

Table 3 Evaluation of deeply embedded VST

	Lines of code
Shallow Embedding Proofs of Primary Rules	12,068
Deep Embedding Definition	297
Deep Embedding Soundness Proof	261
Deep Embedding Extended Rule Proofs	2114
Modification to User Proofs	**0**

FRAME and HOARE- STORE- FRAME in its original development. Thus, those modifications discussed at the end of Sect. 8.1.2 are not even needed.

We demonstrate the evaluation of our deeply embedded VST in Table 3. It only takes very few lines of code to formalize primary rules and logic soundness by reusing original proofs from the shallow embedding. We are able to formalize extended rules in the deep embedding with reasonable lines of code, which was not available in the shallow embedding. Moreover, there is no need to modify user proofs and users can upgrade their projects depending on the shallowly embedded VST to our deeply embedded one for free.

10 Related Work

In Sect. 4, we have already introduced various verification projects using different Hoare logic embeddings. In this section, we review other work on program logic embeddings.

Cook's [14] proof of relative completeness for Hoare logic gives another view of problems we have discussed in Sect. 5. We now assume the deeply embedded logic $\vdash S$ comes with its completeness proof w.r.t. the shallowly embedded logic $\vDash S$, instead of the incompleteness assumption in Sect. 5. With completeness, the proof of inversion rules under all shallow embeddings can be greatly simplified. Take SEQ- INV as an example, the completeness allows entailment from $\vDash \{P\}c_1 \,;\, c_2\{Q, [\vec{R}]\}$ to $\vdash \{P\}c_1 \,;\, c_2\{Q, [\vec{R}]\}$, and use the SEQ- INV under the deep embedding to get $\vdash \{P\}c_1\{S, [\vec{R}]\}$ and $\vdash \{S\}c_2\{Q, [\vec{R}]\}$. Lastly, by the soundness of the deeply embedded logic, we get $\vDash \{P\}c_1\{S, [\vec{R}]\}$ and $\vDash \{S\}c_2\{Q, [\vec{R}]\}$ and finish the proof. The completeness enables proofs of this style for inversion rules, which can be further utilized to simplify proofs of structural rules and transformation rules. Nevertheless, as we have mentioned in Sect. 5, the completeness proof is often missing for logics in real verification projects, and our complicated constructive proofs under the shallow embedding should be trivial when compared to the even more complex completeness proof.

Nipkow [43] compares shallowly and deeply embedded assertion languages in an early paper. He considers a combination of shallowly embedded programming languages, shallowly embedded assertion languages and shallowly embedded program logics. He also considers a combination of deeply embedded programming languages, deeply embedded assertion languages and executable symbolic execution functions. His focus is the comparison between these two settings, while we compare multiple logic embeddings with fixed language and assertion embeddings.

11 Conclusion

In this paper, we clarify the nomenclature of "deep embedding" and "shallow embedding" for formalizing programming languages, assertion languages, and program logics and explore one deep embedding and three different techniques to shallowly embed a program logic. We identify a set of extended rules that could benefit the verification work, and we point out critical proof steps to validate them under different embeddings. We find they are relatively easier to prove under the deeply embedded logic because we can use induction over the proof tree. To alleviate the proof burden of instantiating extended rules for existing verification tools using shallow embeddings, we propose a method to build deeply embedded program logics on existing shallowly embedded ones, where we only need to reuse proofs for primary rules under the shallow embedding to build the deep one.

We also extend our results from conventional Hoare logics to separation logics. We present the deeply embedded VST, where we reuse the original shallowly embedded VST's proofs for soundness proofs of primary rules and prove extended rules under the deep embedding, which is much easier than the proofs in the original VST (e.g. IF- SEQ, LOOP- NOCONTINUE). With deeply embedded VST, we demonstrate the feasibility of our theory in a real verification project.

In conclusion, this paper shows that using different embedding to formalize a program logic brings different benefits, where the shallow embedding allows a more straightforward formalization and the deep embedding makes it easier to add extended proof rules. Our work also indicates the possibility of formalizing a deeply embedded logic based on a shallowly embedded logic by reusing its soundness proof. Moreover, for any shallowly embedded logic (even with different embedding techniques), we can build its corresponding deeply embedded logic by the same set of primary rules. This could be an efficient way to extend existing verification tools to support more powerful proof rules like our extended proof rules.

Appendix A: Big-Step Semantics

Figure 15 and Fig. 16 list the complete big-step semantics for the toy language. We use $(c, \sigma) \Downarrow (\text{ek}, \sigma')$ to denote that the program c starts execution from state σ will safely terminate in state σ' with exit kind ek. We use $(c, \sigma) \Uparrow$ to denote that the program c starts execution from state σ will run into an error.

Appendix B: Proofs of Some Extended Rules in Big-Step Semantics

We give some proofs of extended rules that are not covered in Sect. 5.2. All extended rules are proved in our Coq formalization.

Theorem 23 (IF- SEQ) *Forall P, Q, $\vec{R}$, c_1, c_2, c_3, if*

$$\vDash_b \{P\}\texttt{if}\ b\ \texttt{then}\ c_1; c_3\ \texttt{else}\ c_2; c_3\{Q, [\vec{R}]\},$$

then

$$\vDash_b \{P\}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c_3\{Q, [\vec{R}]\}.$$

Proof Consider the initial state to be σ_1 with $\sigma_1 \vDash P$.

REF
$$\frac{\text{eval}(e,\sigma)=v \qquad h(l)=\bot}{(x=\texttt{ref}(e),(\sigma,h))\Downarrow(\epsilon,(\sigma[x\mapsto l],h[l\mapsto v]))}$$

ASSIGN
$$\frac{\text{eval}(e,\sigma)=v}{(x=e,(\sigma,h))\Downarrow(\epsilon,(\sigma[x\mapsto l],h))}$$

STORE
$$\frac{\text{eval}(e_1,\sigma)=l \qquad \text{eval}(e_2,\sigma)=v \qquad h(l)\neq\bot}{([e_1]=e_2,(\sigma,h))\Downarrow(\epsilon,(\sigma,h[l\mapsto v]))}$$

STORE_FAIL
$$\frac{\text{eval}(e_1,\sigma)=l \qquad h(l)=\bot}{([e_1]=e_2,(\sigma,h))\Uparrow}$$

LOAD
$$\frac{\text{eval}(e,\sigma)=l \qquad h(l)=v}{(x=[e],(\sigma,h))\Downarrow(\epsilon,(\sigma[x\mapsto v],h))}$$

LOAD_FAIL
$$\frac{\text{eval}(e,\sigma)=l \qquad h(l)=\bot}{(x=[e],(\sigma,h))\Uparrow}$$

SEQ1
$$\frac{(c_1,\sigma_1)\Downarrow(\epsilon,\sigma_3) \qquad (c_2,\sigma_3)\Downarrow(\text{ek},\sigma_2)}{(c_1\,;c_2,\sigma_1)\Downarrow(\text{ek},\sigma_2)}$$

SEQ2
$$\frac{(c_1,\sigma_1)\Downarrow(\text{ek},\sigma_2)}{(c_1\,;c_2,\sigma_1)\Downarrow(\text{ek},\sigma_2)}$$

SEQ_FAIL1
$$\frac{(c_1,\sigma)\Uparrow}{(c_1\,;c_2,\sigma)\Uparrow}$$

SEQ_FAIL2
$$\frac{(c_1,\sigma_1)\Downarrow(\epsilon,\sigma_2) \qquad (c_1,\sigma_2)\Uparrow}{(c_1\,;c_2,\sigma_1)\Uparrow}$$

Fig. 15 Big-step semantics of the Toy Language (1)

IF_TRUE
$$\frac{\text{eval}(e,\sigma_1)=\text{true} \qquad (c_1,\sigma_1)\Downarrow(\text{ek},\sigma_2)}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\sigma_1)\Downarrow(\text{ek},\sigma_2)}$$

IF_TRUE_FAIL
$$\frac{\text{eval}(e,\sigma_1)=\text{true} \qquad (c_1,\sigma_1)\Uparrow}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\sigma_1)\Uparrow}$$

IF_FALSE
$$\frac{\text{eval}(e,\sigma_1)=\text{false} \qquad (c_2,\sigma_1)\Downarrow(\text{ek},\sigma_2)}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\sigma_1)\Downarrow(\text{ek},\sigma_2)}$$

IF_FALSE_FAIL
$$\frac{\text{eval}(e,\sigma_1)=\text{false} \qquad (c_2,\sigma_1)\Uparrow}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\sigma_1)\Uparrow}$$

FOR
$$\frac{\begin{array}{c}\text{ek is not brk} \qquad (c_1,\sigma_1)\Downarrow(\text{ek},\sigma_3)\\ (c_2,\sigma_3)\Downarrow(\epsilon,\sigma_4) \qquad (\texttt{for}(;;c_2)\ c_1,\sigma_4)\Downarrow(\epsilon,\sigma_2)\end{array}}{(\texttt{for}(;;c_2)\ c_1,\sigma_1)\Downarrow(\epsilon,\sigma_2)}$$

FOR_FAIL1
$$\frac{(c_1,\sigma_1)\Uparrow}{(\texttt{for}(;;c_2)\ c_1,\sigma_1)\Uparrow}$$

FOR_FAIL2
$$\frac{\begin{array}{c}\text{ek is not brk}\\ (c_1,\sigma_1)\Downarrow(\text{ek},\sigma_2) \qquad (c_2,\sigma_2)\Uparrow\end{array}}{(\texttt{for}(;;c_2)\ c_1,\sigma_1)\Uparrow}$$

FOR_BREAK
$$\frac{(c_1,\sigma_1)\Downarrow(\text{brk},\sigma_2)}{(\texttt{for}(;;c_2)\ c_1,\sigma_1)\Downarrow(\epsilon,\sigma_2)}$$

BREAK
$$(\texttt{break},\sigma)\Downarrow(\text{brk},\sigma)$$

CONTINUE
$$(\texttt{continue},\sigma)\Downarrow(\text{brk},\sigma)$$

Fig. 16 Big-step semantics of the Toy Language (2)

If $\mathrm{eval}(e, \sigma_1) = \mathrm{true}$, then we know $c_1; c_3$ will not cause error, and therefore c_1 and c_3 will not cause error and $(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c_3$ will not cause error. The same happens when $\mathrm{eval}(e, \sigma_1) = \mathrm{false}$, and we know $\texttt{if}\ b\ \texttt{then}\ c_1; c_3\ \texttt{else}\ c_2; c_3$ will not cause error.

If $\mathrm{eval}(e, \sigma_1) = \mathrm{true}$ and c_1 terminates with normal exit, then we know forall σ_3 with

$$((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c_3, \sigma_1) \Downarrow (\mathrm{ek}, \sigma_3)$$

there exists σ_2 such that

$$(c_1, \sigma_1) \Downarrow (\epsilon, \sigma_2) \qquad (c_3, \sigma_2) \Downarrow (\mathrm{ek}, \sigma_3)$$

and we can construct by

$$\dfrac{\dfrac{(c_1, \sigma_1) \Downarrow (\epsilon, \sigma_2) \qquad (c_3, \sigma_2) \Downarrow (\mathrm{ek}, \sigma_3)}{(c_1; c_3, \sigma_1) \Downarrow (\mathrm{ek}, \sigma_3)} \qquad \mathrm{eval}(e, \sigma_1) = \mathrm{true}}{(\texttt{if}\ b\ \texttt{then}\ c_1; c_3\ \texttt{else}\ c_2; c_3, \sigma_1) \Downarrow (\mathrm{ek}, \sigma_3)}$$

and by the triple in the premise, we know $(\mathrm{ek}, \sigma) \vDash \{Q, [\vec{R}]\}$ and we prove the triple in the proof goal. If $\mathrm{eval}(e, \sigma_1) = \mathrm{true}$ and c_1 terminates with some control flow exit, then $\texttt{if}\ b\ \texttt{then}\ c_1; c_3\ \texttt{else}\ c_2; c_3$ terminates with the same exit and state, and therefore we know the terminal state satisfies post-conditions. The same happens when $\mathrm{eval}(e, \sigma_1) = \mathrm{false}$, and we prove the triple to be valid.

Theorem 24 (NOCONTINUE) *Forall P, Q, R_{brk}, R_{con}, R'_{con}, c, if c does not contain* `continue` *and* $\vDash_b \{P\}c\{Q, [R_{brk}, R_{con}]\}$, *then* $\vDash_b \{P\}c\{Q, [R_{brk}, R'_{con}]\}$.

Proof Consider the initial state to be σ_1 with $\sigma_1 \vDash P$.

The triple in the premise guarantees c will not cause error and we only need to show any (ek, σ_2) that (c, σ_1) reduces to satisfies $[R_{\mathrm{brk}}, R'_{\mathrm{con}}]$. In fact, we can prove by induction over $(c, \sigma_1) \Downarrow (\mathrm{ek}, \sigma_2)$ that ek is not con, i.e., it never exits with continue because c does not contain `continue`. Therefore, (ek, σ_2) satisfies $[R_{\mathrm{brk}}, \bot]$ and obviously satisfies $[R_{\mathrm{brk}}, R'_{\mathrm{con}}]$. □

The proof idea of NOCONTINUE in the big-step-based embedding is similar to that in the weakest-precondition-based embedding, but we do not need to do the simulation because there is no requirement after each step of reduction in the big-step-based one, and we only need to show certain properties of the final state it reduces to. They are very different from the proof in the continuation-based one because the interpretation of the post-condition's satisfiability is different.

Acknowledgements This material is based upon work supported by NSF China 61902240.

Funding This research is sponsored by National Natural Science Foundation of China (NSFC) Grant No. 61902240.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests other than those declared in the Funding section. The authors have no conflict of interest to declare that are relevant to the content of this article.

References

1. Ahman, D., Hriţcu, C., Maillard, K., Martínez, G., Plotkin, G., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 515–529 (2017)
2. Appel, A.W.: Verified software toolchain. In: Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software, pp. 1–17 (2011)
3. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. (TOPLAS) **23**(5), 657–683 (2001)
4. Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., de Rauglaudre, D., Filliâtre, J.-C., Giménez, E., Herbelin, H., et al.: The COQ proof assistant reference manual. INRIA, version 6(11) (1999)
5. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules. In: 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05), pp. 260–269. IEEE (2005)
6. Bourbaki, N.: Sur le théorème de zorn. Arch. Math. **2**(6), 434–437 (1949)
7. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1–3), 227–270 (2007)
8. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: International Conference on Theorem Proving in Higher Order Logics, pp. 134–149. Springer (2008)
9. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: a separation logic tool to verify correctness of programs. J. Autom. Reason. **61**, 367–422 (2018)
10. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, pp. 418–430 (2011)
11. Charguéraud, A.: Program verification through characteristic formulae. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, pp. 321–332 (2010)
12. Charguéraud, A.: Software foundations (6): Separation logic foundations. Electronic Textbook, Version 1.6 (Coq 8.17 or later), 2023-08 (2021)
13. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. **62**(3), 331–365 (2019)
14. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**(1), 70–90 (1978)
15. Coq Formalization of Extended Rules' Proofs. https://github.com/TaoYC0904/ExtendedProofRules
16. Gotsman, A., Berdine, J., Cook, B.: Precision and the conjunction rule in concurrent separation logic. Electron. Notes Theor. Comput. Sci. **276**, 171–190 (2011)
17. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. 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, 22–29 April 2017, Proceedings 26, pp. 584–610. Springer (2017)
18. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for cakeml. 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 26, pp. 584–610. Springer (2017)
19. Jacobs, B.: Dijkstra and Hoare monads in monadic computation. Theor. Comput. Sci. **604**, 30–45 (2015)
20. Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, 20 (2018)
21. Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., Jacobs, B.: The future is ours: prophecy variables in separation logic. Proc. ACM Program. Lang. **4**(POPL), 1–32 (2019)
22. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: Sel4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220 (2009)
23. Krebbers, R., Leroy, X., Wiedijk, F.: Formal C semantics: CompCert and the C standard. In: International Conference on Interactive Theorem Proving, pp. 543–548. Springer (2014)
24. Lammich, P., Meis, R.: A separation logic framework for imperative HOL. Archive of Formal Proofs, 161 (2012)
25. Lammich, P., Nipkow, T.: Purely functional, simple, and efficient implementation of prim and dijkstra. Archive of Formal Proofs (2019)
26. Liang, H.: Refinement verification of concurrent programs and its applications. PhD thesis, USTC, China (2014)

27. Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hriţcu, C., Rivas, E., Tanter, É.: Dijkstra monads for all. Proc. ACM Program. Lang. **3**(ICFP), 1–29 (2019)
28. Manna, Z., Pnueli, A.: Axiomatic approach to total correctness of programs. Acta Inform. **3**, 243–263 (1974)
29. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: YNot: dependent types for imperative programs. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 229–240 (2008)
30. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation1. J. Funct. Program. **18**(5–6), 865–911 (2008)
31. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 320–333 (2006)
32. Nipkow, T.: Hoare logics in Isabelle/HOL. In: Proof and System-Reliability, pp. 341–367. Springer (2002)
33. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Berlin (2002)
34. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 268–280 (2004)
35. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer, Berlin Heidelberg (1994)
36. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software foundations (2010). https://softwarefoundations.cis.upenn.edu
37. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS 2002: IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)
38. Sanán, D., Zhao, Y., Hou, Z., Zhang, F., Tiu, A., Liu, Y.: CSimpl: a rely-guarantee-based framework for verifying concurrent programs. In: Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017, Proceedings, Part I 23, pp. 481–498. Springer (2017)
39. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
40. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 77–87 (2015)
41. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. ACM SIGPLAN Not. **48**(6), 387–398 (2013)
42. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955)
43. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, 14–17 September 2004. Proceedings 17, pp. 305–320. Springer (2004)
44. Witt, E.: Beweisstudien zum Satz von M. Zorn. Herrn Erhard. Schmidt zum 75. Geburtstag gewidmet. Math. Nachr. **4**(1–6), 434–438 (1950)
45. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels. In: International Conference on Computer Aided Verification, pp. 59–79. Springer (2016)
46. Yu, D., Hamid, N.A., Shao, Z.: Building certified libraries for PCC: dynamic storage allocation. In: Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, 7–11 April 2003 Proceedings, pp. 363–379. Springer (2003)
47. Zhan, B.: Auto2, a saturation-based heuristic prover for higher-order logic. In: International Conference on Interactive Theorem Proving, pp. 441–456. Springer (2016)
48. Zhan, B.: Verifying imperative programs using auto2. Archive of Formal Proofs (2018)
49. Zhou, L., Qin, J., Wang, Q., Appel, A.W., Cao, Q.: VST-A: a foundationally sound annotation verifier. Proc. ACM Program. Lang. **8**(POPL), 2069–2098 (2024)