Program Verification by Coinduction

Brandon Moore[1], Lucas Peña[2](✉), and Grigore Rosu[1,2]

[1] Runtime Verification, Inc., Urbana, IL, USA
[2] University of Illinois at Urbana-Champaign, Urbana, IL, USA
lpena7@illinois.edu

Abstract. We present a novel program verification approach based on coinduction, which takes as input an operational semantics. No intermediates like program logics or verification condition generators are needed. Specifications can be written using any state predicates. We implement our approach in Coq, giving a certifying language-independent verification framework. Our proof system is implemented as a single module imported unchanged into language-specific proofs. Automation is reached by instantiating a generic heuristic with language-specific tactics. Manual assistance is also smoothly allowed at points the automation cannot handle. We demonstrate the power and versatility of our approach by verifying algorithms as complicated as Schorr-Waite graph marking and instantiating our framework for object languages in several styles of semantics. Finally, we show that our coinductive approach subsumes reachability logic, a recent language-independent sound and (relatively) complete logic for program verification that has been instantiated with operational semantics of languages as complex as C, Java and JavaScript.

1 Introduction

Formal verification is a powerful technique for ensuring program correctness, but it requires a suitable verification framework for the target language. Standard approaches such as Hoare logic [1] (or verification condition generators) require significant effort to adapt and prove sound and relatively complete for a given language, with few or no theorems or tools that can be reused between languages. To use a software engineering metaphor, Hoare logic is a design pattern rather than a library. This becomes literal when we formalize it in a proof assistant.

We present instead a single language-independent program verification framework, to be used with an executable semantics of the target programming language given as input. The core of our approach is a simple theorem which gives a coinduction principle for proving partial correctness.

To trust a non-executable semantics of a desired language, an equivalence to an executable semantics is typically proved. Executable semantics of programming languages abound in the literature. Recently, executable semantics of several real languages have been proposed, e.g. of C [2], Java [3], JavaScript [4,5], Python [6], PHP [7], CAML [8], thanks to the development of executable semantics engineering frameworks like K [9], PLT-Redex [10], Ott [11], etc., which make defining a formal semantics for a programming language almost as easy as implementing an interpreter, if not easier. Our coinductive program verification approach can be used with any of these executable semantics or frameworks, and is correct-by-construction: no additional "axiomatic semantics", "program logic", or "semantics suitable for verification" with soundness proofs needed.

As detailed in Sect. 6, we are not the first to propose a language-independent verification infrastructure that takes an operational semantics as input, nor the first to propose coinduction for proving isolated properties about some programs. However, we believe that coinduction can offer a fresh, promising and general approach as a language-independent verification infrastructure, with a high potential for automation that has not been fully explored yet. In this paper we make two steps in this direction, by addressing the following research questions:

RQ1 *Is it feasible to have a sound and (relatively) complete verification infrastructure based on coinduction, which is language-independent and versatile, i.e., takes an arbitrary language as input, given by its operational semantics?*

RQ2 *Is it possible to match, or even exceed, the capabilities of existing language-independent verification approaches based on operational semantics?*

To address RQ1, we make use of a key mathematical result, Theorem 1, which has been introduced in more general forms in the literature, e.g., in [12,13] and in [14]. We mechanized it in Coq in a way that allows us to instantiate it with a transition relation corresponding to any target language semantics, hereby producing certifying program verification for that language. Using the resulting coinduction principle to show that a program meets a specification produces a proof which depends only on the operational semantics. We demonstrate our proofs can be effectively automated, on examples including heap data structures and recursive functions, and describe the implemented proof strategy and how it can be reused across languages defined using a variety of operational styles.

To address RQ2, we show that our coinductive approach not only subsumes reachability logic [15], whose practicality has been demonstrated with languages like C, Java, and JavaScript, but also offers several specific advantages. Reachability logic consists of a sound and (relatively) complete proof system that takes a given language operational semantics as a *theory* and derives reachability properties about programs in that language. A mechanical procedure can translate any proof using reachability logic into a proof using our coinductive approach.

We first introduce our approach with a simple intuitive example, then prove its correctness. We then discuss mechanical verification experiments across different languages, show how reachability logic proofs can be translated into coinductive proofs, and conclude with related and future work. Our entire Coq formalization, proofs and experiments are available at [16].

2 Overview and Basic Notions

Section 4 will show the strengths of our approach by means of verifying rather complex programs. Here our objective is different, namely to illustrate it by verifying a trivial IMP (C-style) program: `s=0; while (--n) {s=s+n;}`. Let `sum` stand for the program and `loop` for its `while` loop. When run with a positive initial value n of `n`, it sets `s` to the sum of $1, \ldots, n-1$. To illustrate non-termination, we assume unbounded integers, so `loop` runs forever for non-positive n. An IMP language syntax sufficient for this example and a possible execution trace are given in Fig. 1. The exact step granularity is not critical for our approach, as long as diverging executions produce infinite traces.

```
Pgm  ::= Stmt

Exp  ::= Id
       | Int
       | -- Id
       | Exp op Exp

Stmt ::= skip
       | Stmt Stmt
       | Id = Exp ;
       | if Exp { Stmt }
         else { Stmt }
       | while Exp { Stmt }
```

```
           ⟨s=0; while (--n) {s=s+n;} | n ↦ 4⟩
                ⟨while (--n) {s=s+n;} | n ↦ 4, s ↦ 0⟩
⟨if (--n) {s=s+n; loop} else {skip} | n ↦ 4, s ↦ 0⟩
  ⟨if (3) {s=s+n; loop} else {skip} | n ↦ 3, s ↦ 0⟩
                     ⟨s=s+n; loop | n ↦ 3, s ↦ 0⟩
                     ⟨s=0+n; loop | n ↦ 3, s ↦ 0⟩
                     ⟨s=0+3; loop | n ↦ 3, s ↦ 0⟩
                       ⟨s=3; loop | n ↦ 3, s ↦ 0⟩
                      ⟨skip; loop | n ↦ 3, s ↦ 3⟩
                ⟨while (--n) {s=s+n;} | n ↦ 3, s ↦ 3⟩
                                  ··· | ···
                ⟨while (--n) {s=s+n;} | n ↦ 1, s ↦ 6⟩
⟨if (--n) {s=s+n; loop} else {skip} | n ↦ 1, s ↦ 6⟩
  ⟨if (0) {s=s+n; loop} else {skip} | n ↦ 0, s ↦ 6⟩
                                ⟨skip | n ↦ 0, s ↦ 6⟩
```

Fig. 1. Syntax of **IMP** (left) and sample execution of `sum` (right)

While our coinductive program verification approach is self-contained and thus can be presented without reliance on other verification approaches, we prefer to start by discussing the traditional Hoare logic approach, for two reasons. First, it will put our coinductive approach in context, showing also how it avoids some of the limitations of Hoare logic. Second, we highlight some of the subtleties of Hoare logic when related to operational semantics, which will help understand the reasons and motivations underlying our definitions and notations.

2.1 Intuitive Hoare Logic Proof

A Hoare logic specification/triple has the form $\{\!|\varphi_{pre}|\!\}$ `code` $\{\!|\varphi_{post}|\!\}$. The convenience of this notation depends on specializing to a particular target language, such as allowing variable names to be used directly in predicates to stand for their values, or writing only the current statement. This hides details of the environment/state representation, and some framing conventions or compositionality assumptions over the unmentioned parts. A Hoare triple specifies a set of (partial correctness) reachability claims about a program's behavior, and it is

(IMP statement rules)

$$\frac{\cdot}{\{\!|\varphi[\mathtt{e}/\mathtt{x}]|\!\}\ \mathtt{x=e;}\ \{\!|\varphi|\!\}} \quad \text{(HL-ASGN)}$$

$$\frac{\{\!|\varphi_1|\!\}\ \mathtt{s}_1\ \{\!|\varphi_2|\!\},\ \ \{\!|\varphi_2|\!\}\ \mathtt{s}_2\ \{\!|\varphi_3|\!\}}{\{\!|\varphi_1|\!\}\ \mathtt{s}_1\ \mathtt{s}_2\ \{\!|\varphi_3|\!\}} \quad \text{(HL-SEQ)}$$

$$\frac{\{\!|\varphi \wedge \mathtt{e} \neq 0|\!\}\ \mathtt{s}_1\ \{\!|\varphi'|\!\},\ \ \{\!|\varphi \wedge \mathtt{e} = 0|\!\}\ \mathtt{s}_2\ \{\!|\varphi'|\!\}}{\{\!|\varphi|\!\}\ \mathtt{if\ (e)\ then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}\ \{\!|\varphi'|\!\}} \quad \text{(HL-IF)}$$

$$\frac{\{\!|\varphi \wedge \mathtt{e} \neq 0|\!\}\ \mathtt{s}\ \{\!|\varphi|\!\}}{\{\!|\varphi|\!\}\ \mathtt{while\ (e)}\ \{\mathtt{s}\}\ \{\!|\varphi \wedge \mathtt{e} = 0|\!\}} \quad \text{(HL-WHILE)}$$

(Generic rule)

$$\frac{\models \psi \rightarrow \varphi,\ \ \{\!|\varphi|\!\}\ \mathtt{s}\ \{\!|\varphi'|\!\},\ \ \models \varphi' \rightarrow \psi'}{\{\!|\psi|\!\}\ \mathtt{s}\ \{\!|\psi'|\!\}} \quad \text{(HL-CONSEQ)}$$

Fig. 2. IMP program logic.

typically an over-approximation (i.e., it specifies more reachability claims than desired or feasible). Specifically, assume some formal language semantics of IMP defining an execution step relation $R \subseteq C \times C$ on a set C of configurations of the form $\langle \mathtt{code} \mid \sigma \rangle$, like those in Fig. 1. We write $a \rightarrow_R b$ for $(a, b) \in R$. Section 2.3 (Fig. 3) discusses several operational semantics approaches we experimented with (Sect. 4), that yield such step relations R. A (partial correctness) *reachability claim* (c, P), relating an initial state $c \in C$ and a target set of states $P \subseteq C$, is *valid* (or *holds*) iff the initial state c can either reach a state in P or can take an infinite number of steps (with $\rightarrow_R$); we write $c \Rightarrow_R P$ to indicate that claim (c, P) is valid, and $a \rightarrow b$ or $c \Rightarrow P$ instead of $a \rightarrow_R b$ or $c \Rightarrow_R P$, resp., when R is understood. Then $\{\!|\varphi_{pre}|\!\}\mathtt{code}\{\!|\varphi_{post}|\!\}$ specifies the set of reachability claims

$$\{(\langle \mathtt{code} \mid \sigma_{pre} \rangle, \{\langle \mathtt{skip} \mid \sigma_{post} \rangle \mid \sigma_{post} \models \varphi_{post}\}) \mid \sigma_{pre} \models \varphi_{pre}\}$$

and it is *valid* iff all of its reachability claims are valid. It is necessary for P in reachability claims (c, P) specified by Hoare triples to be a set of configurations (and thus an over-approximation): it is generally impossible for φ_{post} to determine exactly the possible final configuration or configurations.

While one can prove Hoare triples valid directly using the step relation $\rightarrow_R$ and induction, or coinduction like we propose in this paper, the traditional approach is to define a language-specific proof system for deriving Hoare triples from other triples, also known as *a* Hoare logic, or program logic, for the target programming language. Figure 2 shows such a program logic for IMP. Hoare logics are generally not executable, so testing cannot show whether they match the *intended* semantics of the language. Even for a simple language like IMP, if one mistakenly writes $\mathtt{e} = 1$ instead of $\mathtt{e} \neq 0$ in rule (HL-WHILE), then one gets an incorrect program logic. When trusted verification is desired, the program logic needs to be proved sound w.r.t. a reference executable semantics of the language, i.e, that each derivable Hoare triple is valid. This is a highly non-trivial task for complex languages (C, Java, JavaScript), in addition to defining a Hoare logic itself. Our coinductive approach completely avoids this difficulty by requiring no additional semantics of the programming language for verification purposes.

The property to prove is that `sum` (or more specifically `loop`) exits only when `n` is 0, with `s` as the sum $\sum_{i=1}^{n-1} i$ (or $\frac{n(n-1)}{2}$). In more detail, any configuration whose statement begins with `sum` and whose store defines `n` as n can run indefinitely or reach a state where it has just left the loop with $\mathtt{n} \mapsto 0$, $\mathtt{s} \mapsto \sum_{i=1}^{n-1} i$, and the store otherwise unchanged. As a Hoare logic triple, that specification is

$$\{\!|\mathtt{n} = n|\!\} \ \mathtt{s=0;\ while(--n)\{s=s+n;\}} \ \{\!|\mathtt{s} = \sum_{i=1}^{n-1} i \wedge \mathtt{n}=0|\!\}$$

As seen, this Hoare triple asserts the validity of the set of reachability claims

$$S \equiv \{(c_{n,\sigma}, P_{n,\sigma}) \mid \forall n, \forall \sigma \text{ undefined in } \mathtt{n}\} \qquad (1)$$

where

$$c_{n,\sigma} \equiv \langle \mathtt{s=0;\ while(--n)\{s=s+n;\}} \,|\, \mathtt{n} \mapsto n,\ \sigma\rangle$$
$$P_{n,\sigma} \equiv \{\langle \mathtt{skip} \,|\, \mathtt{n} \mapsto 0, \mathtt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma'\rangle \mid \forall \sigma' \text{ undefined in } \mathtt{n}, \mathtt{s}\}$$

We added the σ and σ' state frames above for the sake of complete details about what Hoare triples actually specify, and to illustrate why P in claims (c, P) needs to be a set. Since the addition/removal of σ and σ' does not change the subsequent proofs, for the remainder of this section, for simplicity, we drop them.

Now let us assume, without proof, that the proof system in Fig. 2 is sound (for the executable step relation $\rightarrow_R$ of IMP discussed above), and let us use it to derive a proof of the `sum` example. Note that the proof system in Fig. 2 assumes that expressions have no side effects and thus can be used unchanged in state formulae, which is customary in Hoare logics, so the program needs to be first translated out into an equivalent one without the problematic `--n` where expressions have no side effects. We could have had more Hoare logic rules instead of needing to translate the code segment, but this would quickly make our program logics significantly more complicated. Either way, with even a simple imperative programming language like we have here, it is necessary to either add Hoare logic rules to Fig. 2 or to modify our code segment. These inconveniences are taken for granted in Hoare logic based verifiers, and they require non-negligible additional effort if trusted verification is sought. For comparison, our coinductive verification approach proposed in this paper requires no transformation of the original program. After modifying the above problematic expression, our code segment gets translated to the (hopefully) equivalent code:

```
s=0; n=n-1; while (n) {s=s+n; n=n-1;}
```

Let `loop'` be the new loop and let φ_{inv}, its invariant, be

$$\mathtt{s} = \frac{((n-1) - \mathtt{n})\,(n + \mathtt{n})}{2}$$

The program variable `n` stands for its current value, while the mathematical variable n stands for the initial (sometimes called "old") value of `n`. Next, using the assign and sequence Hoare logic rules in Fig. 2, as well as basic arithmetic via the (HL-CONSEQ) rule, we derive

$$\{\!|\texttt{n} = n|\!\}\ \texttt{s=0; n=n-1;}\ \{\!|\varphi_{inv}|\!\} \tag{2}$$

Similarly, we can derive $\{\!|\varphi_{inv} \wedge \texttt{n} \neq 0|\!\}$ `s=s+n; n=n-1;` $\{\!|\varphi_{inv}|\!\}$. Then, applying the while rule, we derive $\{\!|\varphi_{inv}|\!\}$ `loop'` $\{\!|\varphi_{inv} \wedge \texttt{n} = 0|\!\}$. The rest follows by the sequence rule with the above, (2), and basic arithmetic.

This example is not complicated, in fact it is very intuitive. However, it abstracts out a lot of details in order to make it easy for a human to understand. It is easy to see the potential difficulties that can arise in larger examples from needing to factor out the side effect, and from mixing both program variables and mathematical variables in Hoare logic specifications and proofs. With our coinduction verification framework, all of these issues are mitigated.

2.2 Intuitive Coinduction Proof

Since our coinductive approach is language-independent, we do not commit to any particular, language-specific formalism for specifying reachability claims, such as Hoare triples. Consequently, we will work directly with raw reachability claims/specifications $S \subseteq C \times \mathcal{P}(C)$ consisting of sets of pairs (c, P) with $c \in C$ and $P \subseteq C$ as seen above. We show how to coinductively prove the claim for the example `sum` program in the form given in (1), relying on nothing but a general language-independent coinductive machinery and the trusted execution step relation $\to_R$ of IMP. Recall that we drop the state frames (σ) in (1).

Intuitively, our approach consists of symbolic execution with the language step relation, plus coinductive reasoning for circular behaviors. Specifically, suppose that $S_{circ} \subseteq C \times \mathcal{P}(C)$ is a specification corresponding to some code with circular behavior, say some loop. Pairs $(c, P) \in S_{circ}$ with $c \in P$ are already valid, that is, $c \Rightarrow_R P$ for those. "Execute" the other pairs $(c, P) \in S_{circ}$ with the step relation $\to_R$, obtaining a new specification S' containing pairs of the form (d, P), where $c \to_R d$; since we usually have a mathematical description of the pairs in S_{circ} and S', this step has the feel of symbolic execution. Note that S_{circ} is valid if S' is valid. Do the same for S' obtaining a new specification S'', and so on and so forth. If at any moment during this (symbolic) execution process we reach a specification S that is included in our original S_{circ}, then simply assume that S is valid. While this kind of cyclic reasoning may not seem sound, it is in fact valid, and justified by *coinduction*, which captures the essence of partial correctness, *language-independently*. Reaching something from the original specification shows we have reached some fixpoint, and coinduction is directly related to greatest fixpoints. This is explained in detail in Sect. 3.

In many examples it is useful to chain together individual proofs, similar to (HL-SEQ). Thus, we introduce the following sequential composition construct:

Definition 1. *For $S_1, S_2 \subseteq C \times \mathcal{P}(C)$, let $S_1 \mathbin{⨾} S_2 \equiv \{(c, P) \mid \exists Q \,.\, (c, Q) \in S_1 \wedge \forall d \in Q, (d, P) \in S_2\}$. Also, we define* $\mathrm{trans}(S)$ *as $S \mathbin{⨾} S$ (*trans *can be thought of as a transitivity proof rule).*

If S_1 and S_2 are valid then $S_1 \mathbin{⨾} S_2$ is also valid (Lemma 2).

Given n, let Q_n and T_n be the following sets of configurations, where Q_n and T_n represent the *invariant set* and *terminal set*, respectively:

$$Q_n \equiv \{\langle \texttt{loop} \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i \rangle \mid \forall n'\}$$
$$T_n \equiv \{\langle \texttt{skip} \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i \rangle\}$$

and let us define the following specifications:

$$S_1 \equiv \{(\langle \texttt{s=0; loop} \,|\, \texttt{n} \mapsto n \rangle, Q_n) \mid \forall n\}$$
$$S_2 \equiv \{(\langle \texttt{loop} \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i \rangle, T_n) \mid \forall n, n'\}$$

Our target S in (1) is included in $S_1 \mathbin{⨾} S_2$, so it suffices to show that S_1 and S_2 are valid. S_1 clearly is: $\langle \texttt{s=0;loop} \,|\, \texttt{n} \mapsto n \rangle \to_R^+ \langle \texttt{loop} \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0 \rangle$ represents the (symbolic) execution step or steps taken to assign program variable `s`, and the set of specifications $\{(\langle \texttt{loop} \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0 \rangle, Q_n) \mid \forall n\}$ is vacuously valid (note $\sum_{i=n}^{n-1} i = 0$). For the validity of S_2, we partition it in two subsets, one where $n' = 1$ and another with $n' \neq 1$ (case analysis). The former holds same as S_1, noting that

$$\langle \texttt{loop} \,|\, \texttt{n} \mapsto 1, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i \rangle \to_R^+ \langle \texttt{skip} \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i \rangle$$

The latter holds by coinduction (for S_2), because first

$$\langle \texttt{loop} \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i \rangle \to_R^+ \langle \texttt{loop} \,|\, \texttt{n} \mapsto n' - 1, \texttt{s} \mapsto \textstyle\sum_{i=n'-1}^{n-1} i \rangle$$

and second the following inclusion holds:

$$\{(\langle \texttt{loop} \,|\, \texttt{n} \mapsto n' - 1, \texttt{s} \mapsto \textstyle\sum_{i=n'-1}^{n-1} i \rangle, T_n) \mid \forall n, n'\} \subseteq S_2$$

The key part of the proof above was to show that the reachability claim about the loop (S_2) was stable under the language semantics. Everything else was symbolic execution using the (trusted) operational semantics of the language. By allowing desirable program properties to be uniformly specified as reachability claims about the (executable) language semantics itself, our approach requires no auxiliary formalization of the language for verification purposes, and thus no soundness or equivalence proofs and no transformations of the original program to make it fit the restrictions of the auxiliary semantics. Unlike for the Hoare logic proof, the main "proof rules" used were just performing execution steps using the operational semantics rules, as well as the generic coinductive principle. Section 3 provides all the technical details.

$$\boxed{\text{Structural Operational Semantics}}$$

$$\langle x \mid \sigma \rangle \rightarrow \langle \sigma(x) \mid \sigma \rangle$$

$$\langle \texttt{--}x \mid \sigma \rangle \rightarrow \langle i \mid \sigma[i/x] \rangle \quad \text{if } i = \sigma(x) -_{Int} 1$$

$$\frac{\langle e_1 \mid \sigma \rangle \rightarrow \langle e_1' \mid \sigma' \rangle}{\langle e_1 \texttt{ op } e_2 \mid \sigma \rangle \rightarrow \langle e_1' \texttt{ op } e_2 \mid \sigma' \rangle}$$

$$\frac{\langle e_2 \mid \sigma \rangle \rightarrow \langle e_2' \mid \sigma' \rangle}{\langle i_1 \texttt{ op } e_2 \mid \sigma \rangle \rightarrow \langle i_1 \texttt{ op } e_2' \mid \sigma' \rangle}$$

$$\langle i_1 \texttt{ op } i_2 \mid \sigma \rangle \rightarrow \langle i_1 \ op_{Int}\ i_2 \mid \sigma \rangle$$

$$\frac{\langle s_1 \mid \sigma \rangle \rightarrow \langle s_1' \mid \sigma' \rangle}{\langle s_1 \ \ s_2 \mid \sigma \rangle \rightarrow \langle s_1' \ \ s_2 \mid \sigma' \rangle}$$

$$\langle \texttt{skip} \ \ s \mid \sigma \rangle \rightarrow \langle s \mid \sigma \rangle$$

$$\frac{\langle e \mid \sigma \rangle \rightarrow \langle e' \mid \sigma' \rangle}{\langle x \texttt{ := } e \mid \sigma \rangle \rightarrow \langle x \texttt{ := } e' \mid \sigma' \rangle}$$

$$\langle x \texttt{ := } i \mid \sigma \rangle \rightarrow \langle \texttt{skip} \mid \sigma[i/x] \rangle$$

$$\frac{\langle e \mid \sigma \rangle \rightarrow \langle e' \mid \sigma' \rangle}{\langle \texttt{if } e \texttt{ then } \{s_1\} \texttt{ else } \{s_2\} \mid \sigma \rangle \rightarrow \langle \texttt{if } e' \texttt{ then } \{s_1\} \texttt{ else } \{s_2\} \mid \sigma' \rangle}$$

$$\langle \texttt{if } \ i \ \texttt{ then } \{s_1\} \texttt{ else } \{s_2\} \mid \sigma \rangle \rightarrow \langle s_1 \mid \sigma \rangle \quad \text{if } i \neq 0$$

$$\langle \texttt{if 0 then } \{s_1\} \texttt{ else } \{s_2\} \mid \sigma \rangle \rightarrow \langle s_2 \mid \sigma \rangle$$

$$\langle \texttt{while } e \ \{s\} \mid \sigma \rangle \rightarrow \langle \texttt{if } e \texttt{ then } \{s \texttt{ while } e \ \{s\}\} \texttt{ else } \{\texttt{skip}\} \mid \sigma \rangle$$

$$\boxed{\text{Reduction Semantics}}$$

(evaluation contexts syntax omitted— [17])

$$\frac{r \rightarrow r'}{E[r] \rightarrow E[r']}$$

$$\langle E \mid \sigma \rangle[x] \rightarrow \langle E \mid \sigma \rangle[\sigma(x)]$$

$$\langle E \mid \sigma \rangle[\texttt{--}x] \rightarrow \langle E \mid \sigma[i/x] \rangle[i] \quad \text{if } i = \sigma(x) -_{Int} 1$$

$$\langle E \mid \sigma \rangle[x \texttt{:=}\ i] \rightarrow \langle E \mid \sigma[i/x] \rangle[\texttt{skip}]$$

$$i_1 \texttt{ op } i_2 \rightarrow i_1 \ op_{Int}\ i_2$$
$$\texttt{skip} \ \ s \rightarrow s$$
$$\texttt{if } \ i \ \texttt{ then } \{s_1\} \texttt{ else } \{s_2\} \rightarrow s_1 \quad \text{if } i \neq 0$$
$$\texttt{if 0 then } \{s_1\} \texttt{ else } \{s_2\} \rightarrow s_2$$
$$\texttt{while } e \ \{s\} \rightarrow \texttt{if } e \texttt{ then } \{s \texttt{ while } e \ \{s\}\} \texttt{ else } \{\texttt{skip}\}$$

$$\boxed{\text{K Semantics}}$$

(configuration and strictness omitted— [9])

$$\langle \underset{i}{\underline{x}} \ ...\rangle_k \ \langle ... \ x \mapsto i \ ... \rangle_{state}$$

$$\langle \underset{i -_{Int} 1}{\underline{\texttt{--} \ x}} \ ...\rangle_k \ \langle ... \ x \mapsto \underset{i -_{Int} 1}{\underline{i}} \ ... \rangle_{state}$$

$$\langle \underset{\texttt{skip}}{\underline{x \ \texttt{:=} \ i}} \ ...\rangle_k \ \langle ... \ x \mapsto \underset{i}{\underline{\_}} \ ... \rangle_{state}$$

(plus the last five simple rules under reduction semantics)

Fig. 3. Three different operational semantics of **IMP**, generating the same execution step relation R (or $\rightarrow_R$).

2.3 Defining Execution Step Relations

Since our coinductive verification framework is parametric in a step relation, which also becomes the only trust base when certified verification is sought, it is imperative for its practicality to support a variety of approaches to define step relations. Ideally, it should not be confined to any particular semantic style that ultimately defines a step relation, and it should simply take existing semantics "off-the-shelf" and turn them into sound and relatively complete program verifiers for the defined languages. We briefly recall three of the semantic approaches that we experimented with in our Coq formalization [16].

Small-step structural operational semantics [18] (Fig. 3 top) is one of the most popular semantic approaches. It defines the transition relation inductively. This semantic style is easy to use, though often inconvenient to define some features such as abrupt changes of control and true concurrency. Additionally, finding the next successor of a configuration may take longer than in other approaches. Reduction semantics with evaluation contexts [17], depicted in the middle of Fig. 3, is another popular approach. It allows us to elegantly and compactly define complex evaluation strategies and semantics of control intensive constructs (e.g., call/cc), and it avoids a recursive definition of the transition relation. On the other hand, it requires an auxiliary definition of contexts along with splitting and plugging functions.

As discussed in Sect. 1, several large languages have been given formal semantics using K [9] (Fig. 3 bottom). K is more involved and less conventional than the other approaches, so it is a good opportunity to evaluate our hypothesis that we can just "plug-and-play" operational semantics in our coinductive framework. A K-style semantics extends the code in the configuration to a list of terms, and evaluates within subterms by having a transition that extracts the term to the front of the list, where it can be examined directly. This allows a non-recursive definition of transition, whose cases can be applied by unification.

In practice, in our automation, we only need to modify how a successor for a configuration is found. Besides that, the proofs remain exactly the same.

3 Coinduction as Partial Correctness

The intuitive coinductive proof of the correctness of `sum` in Sect. 2.2 likely raised a lot of questions. We give formal details of that proof in this section as well go through some definitions and results of the underlying theory. All proofs, including our Coq formalization, are in [16].

3.1 Definitions and Main Theorem

First, we introduce a definition that we used intuitively in the previous section:

Definition 2. *If* $R \subseteq C \times C$*, let* $\mathrm{valid}_R \subseteq C \times \mathcal{P}(C)$ *be defined as* $\mathrm{valid}_R = \{(c, P) \mid c \Rightarrow_R P \text{ holds}\}$.

Recall from Sect. 2.1 that $c \Rightarrow_R P$ holds iff the initial state c can either reach a state in P or can take an infinite number of steps (with $\rightarrow_R$). Pairs $(c, P) \in C \times \mathcal{P}(C)$ are called *claims* or *specifications*, and our objective is to prove they hold, i.e., $c \Rightarrow_R P$. Sets of claims $S \subseteq C \times \mathcal{P}(C)$ are valid if $S \subseteq \text{valid}_R$. To show such inclusions by coinduction, we notice that valid_R is a greatest fixpoint, specifically of the following operator:

Definition 3. *Given $R \subseteq C \times C$, let $\text{step}_R : \mathcal{P}(C \times \mathcal{P}(C)) \rightarrow \mathcal{P}(C \times \mathcal{P}(C))$ be*

$$\text{step}_R(S) = \{(c, P) \mid c \in P \ \vee \ \exists d\,.\, c \rightarrow_R d \wedge (d, P) \in S\}$$

Therefore, to prove $(c, P) \in \text{step}_R(S)$, one must show either that $c \in P$ or that $(succ(c), P) \in S$, where $succ(c)$ is a resulting configuration after taking a step from c by the operational semantics.

Definition 4. *Given a monotone function $F : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$, let its F-*closure *$F^* : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ be defined as $F^*(X) = \mu Y.\, F(Y) \cup X$, where μ is the least fixpoint operator. This is well-defined as $Y \mapsto F(Y) \cup X$ is monotone for any X.*

The following lemma suffices for reachability verification:

Lemma 1. *For any $R \subseteq C \times C$ and $S \subseteq C \times \mathcal{P}(C)$, we have $S \subseteq \text{step}_R(\text{step}^*_R(S))$ implies $S \subseteq \text{valid}_R$.*

The intuition behind this lemma is captured in Sect. 2.2: we continue taking steps and once we reach a set of states already seen, we know our claim is valid. This would not be valid if $\text{step}_R(\text{step}^*_R(S))$ was replaced simply with $\text{step}^*_R(S)$, as $X \subseteq F^*(X)$ hold trivially for any F and X. Lemma 1 (along with elementary set properties) replaces the entire program logic shown in Fig. 2. The only formal definition specific to the target language is the operational semantics. Lemma 1 does not need to be modified or re-proven to use it with other languages or semantics. It generalizes into a more powerful result, that can be used to derive a variety of coinductive proof principles:

Theorem 1. *If $F, G : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ are monotone and $G(F(A)) \subseteq F(G^*(A))$ for any $A \subseteq D$, then $X \subseteq F(G^*(X))$ implies $X \subseteq \nu F$ for any $X \subseteq D$, where νF is the greatest fixpoint of F.*

Proofs, including a verified proof in our Coq formulation are in [16]. The proof can also be derived from [12–14], though techniques from these papers had previously not been applied to program verification. Lemma 1 is an easy corollary, with both F and G instantiated as step_R, along with a proof that $\nu\, \text{step}_R = \text{valid}_R$ (see [16]). However, instantiating F and G to be the same function is not always best. An interesting and useful G is the transitivity function trans in Definition 1, which satisfies the hypothesis in Theorem 1 when F is step_R. [16] shows other sound instantiations of G.

We can also use Theorem 1 with other definitions of validity expressible as a greatest fixpoint, e.g., all-path validity. For nondeterministic languages we might prefer to say $c \Rightarrow^\forall P$ holds if no path from c reaches a stuck configuration without passing through P. This is the greatest fixpoint of

$$\text{step}^\forall_R(S) = \{(c, P) \mid c \in P \ \vee \exists d\,.\, c \rightarrow_R d \wedge \forall d\,.\, (c \rightarrow_R d \text{ implies } (d, P) \in S)\}$$

The universe of validity notions that can be expressed coinductively, and thus the universe of instances of Theorem 1 is virtually limitless. Below is another notion of validity that we experimented with in our Coq formalization [16]. When proving global program invariants or safety properties of non-deterministic programs, we want to state not only reachability claims $c \Rightarrow P$, but also that all the transitions from c to configurations in P respect some additional property, say T. For example, a global state invariant I can be captured by a T such that $(a, b) \in T$ iff $I(a)$ and $I(b)$, while an arbitrary safety property can be captured by a T that encodes a monitor for it. This notion of validity, which we call (all-path) "until" validity, is the greatest fixpoint of:

$$\begin{aligned}\text{until}^{\forall}_{R}(S) = &\{(c, T, P) \mid c \in P \lor \\ &\qquad \exists d \,.\, c \rightarrow_R d \land \forall d \,.\, (c \rightarrow_R d \text{ implies } (c, d) \in T \land (d, T, P) \in S)\}\end{aligned}$$

This allows verification of properties that are not expressible using Hoare logic.

3.2 Example Proof: Sum

Now we demonstrate the results above by providing all the details that were skipped in our informal proof in Sect. 2.2. The property that we want to prove, expressed as a set of claims (c, P), is

$$\begin{aligned}S \equiv \{(&\langle \texttt{s=0;while(--n)\{s=s+n;\}}\, T \,|\, \texttt{n} \mapsto n, \sigma[\bot/\texttt{s}]\rangle, \\ &\{\langle T \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma\rangle\}) \mid \forall n, T, \sigma\}\end{aligned}$$

We have to prove $S \subseteq \text{valid}_R$. Note that this specification is more general than the specifications in Sect. 2.2. Here, T represents the remainder of the code to be executed, while σ represents the remainder of the store, with $\sigma[\bot/\texttt{s}]$ as σ restricted to $Dom(\sigma)/\{\texttt{s}\}$. Thus, we write out the entire configuration here, which gives us freedom in expressing more complex specifications if needed.

Instead of proving this directly, we will prove two subclaims valid and connect them via sequential composition (Definition 1). First, we need the following:

Lemma 2. *$S_1 \mathbin{⨟} S_2 \subseteq \text{valid}_R$ if $S_1 \subseteq \text{valid}_R$ and $S_2 \subseteq \text{valid}_R$.*

As before, let

$$\begin{aligned}Q_n &\equiv \{\langle \texttt{loop;}\ T \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle \mid \forall n'\} \\ T_n &\equiv \{\langle T \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i\rangle\}\end{aligned}$$

and define

$$\begin{aligned}S_1 &\equiv \{(\langle \texttt{s=0; loop;}\ T \,|\, \texttt{n} \mapsto n, \sigma[\bot/\texttt{s}]\rangle, Q_n) \mid \forall n, T, \sigma\} \\ S_2 &\equiv \{(\langle \texttt{loop;}\ T \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle, T_n) \mid \forall n, n', T, \sigma\}\end{aligned}$$

Since $S \subseteq S_1 \fatsemi S_2$ (by Q_n), it suffices to show $S_1 \cup S_2 \subseteq \text{valid}_R$. To prove $S_1 \subseteq \text{valid}_R$, by Lemma 1 we show $S_1 \subseteq \text{step}_R(\text{step}^*_R(S_1))$. Regardless of the employed executable semantics, this should hold:

$$\forall n, T, \sigma.\, \langle \texttt{s=0; loop; } T \,|\, \texttt{n} \mapsto n, \sigma[\bot/\texttt{s}]\rangle \to_R \langle \texttt{loop; } T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma\rangle$$

Choosing the second case of the disjunction in step_R with d matching this step, it suffices to show

$$\{(\langle \texttt{loop; } T \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma\rangle, Q_n) \mid \forall n, T, \sigma\} \subseteq \text{step}^*_R(S_1)$$

Note that we can unfold any fixpoint $F^*(S)$ to get the following two equations:

$$F(F^*(S)) \subseteq F(F^*(S)) \cup S = F^*(S) \qquad S \subseteq F(F^*(S)) \cup S = F^*(S) \qquad (3)$$

We use the first equation to expose an application of step_R on the right hand side, so it suffices to show the above is a subset of $\text{step}_R(\text{step}^*_R(S))$. We then use the first case of the disjunction (showing $c \in P$) in step_R, and instantiating n' to n proves this goal, since $\sum_{i=n}^{n-1} i = 0$. Thus $S_1 \subseteq \text{valid}_R$.

Now we prove $S_2 \subseteq \text{valid}_R$, or $S_2 \subseteq \text{step}_R(\text{step}^*_R(S_2))$. First, note the operational semantics of IMP rewrites while loops to if statements. Then, by the definition of step_R, it suffices to show that

$$\{(\langle \texttt{if(--n)\{s=s+n;loop\}; } T \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle, T_n) \mid \forall n, n', T, \sigma\} \subseteq \text{step}^*_R(S_2)$$

Using the first unfolding from (3), it suffices to show the above is a subset of $\text{step}_R(\text{step}^*_R(S_2))$, i.e. we expose an application of step_R on the right hand side. The definition of step_R thus allows the left hand side to continue taking execution steps, as long as we keep unfolding the fixpoint. Continuing this, the if condition becomes a single, but symbolic, boolean value. Specifically, it suffices to show:

$$\{(\langle \texttt{if(}n'\text{-}1 \neq 0\texttt{)\{s=s+n;loop\};} T \,|\, \texttt{n} \mapsto n'\text{-}1, \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle, T_n) \,|\, \forall n, n', T, \sigma\} \subseteq \text{step}^*_R(S_2)$$

Further progress requires making a case distinction on whether $n' - 1 = 0$. A case distinction corresponds to observing that $A \cup B \subseteq X$ if both $A \subseteq X$ and $B \subseteq X$. Here we split the current set of claims into those with $n' - 1 = 0$ and $n' - 1 \neq 0$, and separately establish the following inclusions:

$$\{(\langle \texttt{if(false)\{s=s+n;loop\};} T \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma\rangle, T_n) \mid \forall n, T, \sigma\} \subseteq \text{step}^*_R(S_2)$$
$$\{(\langle \texttt{if(true)\{s=s+n;loop\};} T \,|\, \texttt{n} \mapsto n'\text{-}1, \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle, T_n) | \forall n, n' \neq 1, T, \sigma\} \subseteq \text{step}^*_R(S_2)$$

Continuing symbolic execution and using $\sum_{i=n'}^{n-1} i + (n' - 1) = \sum_{i=n'-1}^{n-1} i$, we get

$$\{(\langle T \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma\rangle, T_n) \mid \forall n, T, \sigma\} \subseteq \text{step}^*_R(S_2)$$
$$\{(\langle \texttt{loop; } T \,|\, \texttt{n} \mapsto n' - 1, \texttt{s} \mapsto \textstyle\sum_{i=n'-1}^{n-1} i, \sigma\rangle, T_n) \mid \forall n, n', T, \sigma, n' - 1 \neq 0\} \subseteq \text{step}^*_R(S_2)$$

In the $n' - 1 = 0$ case, the current configuration is already in the corresponding target set. To conclude, we expose another application of step_R as before, but use the clause $c \in P$ of the disjunction in step_R to leave the trivial goal $\forall n, T, \sigma.\, \langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \frac{n(n-1)}{2}, \sigma\rangle \in \{\langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \frac{n(n-1)}{2}, \sigma\rangle\}$. For the $n' - 1 \neq 0$ case, we have a set of claims that are contained in the initial specification S_2. We conclude by showing $S_2 \subseteq \text{step}_R^*(S_2)$ from the second equation in (3) by noting that $S \subseteq F^*(S)$ for any F. So this set of claims is contained in S_2 by instantiating the universally quantified variable n' in the definition of S_2 with $n' - 1$. Thus it is contained in $\text{step}_R^*(S_2)$ and thus it is a subset of valid_R.

3.3 Example Proof: Reverse

Consider now the following program to reverse a linked list, written in the HIMP language (Fig. 5a). We will discuss HIMP in more detail Sect. 4.

```
decl p; decl y; p := 0;
while (x<>0) { y := (x+1); *(x+1) := p; p := x; x := y; }
```

Call the above code `rev` and the loop `rev-loop`. We prove this program is correct following intuitions from separation logic [19,20] but using the exact same coinductive technical machinery as before. Assuming we have a predicate that matches a heap containing only a linked list starting at address x and representing the list l (which we will see in Sect. 4.2), our specification becomes:

$$S \equiv \{(\langle \texttt{rev;}\, T \mid \text{list}(l, x)\rangle, \{\langle T \mid \lambda r.\text{list}(rev(l), r)\rangle\}) \mid \forall l, x, T\}$$

where rev is the mathematical list reverse. We proceed as in the previous example, first using lemma then stepping with the semantics, but with Q_n as

$$\{\langle \texttt{rev-loop;}\, T \mid \text{list}(A, x) * \text{list}(B, p) * \texttt{x} \mapsto x * \texttt{p} \mapsto p * \texttt{y} \mapsto y * \lambda r.\text{list}(B\texttt{++}A, r)\rangle \\ \mid \forall A, B, p, y\}$$

where `++` is list append. We continue as before to prove our original specification. S_1 and S_2 follow from our choice for Q_n, our "loop invariant." Specifically,

$$S_1 \equiv \{(\langle \texttt{rev;}\, T \mid \text{list}(l, x)\rangle, \{\langle \texttt{rev-loop;}\, T \mid \text{list}(A, x) * \text{list}(B, p) * \texttt{x} \mapsto x * \texttt{p} \mapsto p * \texttt{y} \mapsto y \\ * \lambda r.\text{list}(B\texttt{++}A, r)\rangle \mid \forall A, B, p, y\}) \mid \forall l, x, T\}$$

$$S_2 \equiv \{(\langle \texttt{rev-loop;}\, T \mid \text{list}(A, x) * \text{list}(B, p) * \texttt{x} \mapsto x * \texttt{p} \mapsto p * \texttt{y} \mapsto y * \lambda r.\text{list}(B\texttt{++}A, r)\rangle, \\ \{\langle T \mid \lambda r.\text{list}(rev(l), r)\rangle\}) \mid \forall A, B, p, y, l, x, T\}$$

Then, the individual proofs for these specifications closely follow the same flavor as in the previous example: use step_R to execute the program via the operational semantics, use unions to case split as needed, and finish when we reach something in the target set or that was previously in our specification. The inherent similarity between these two examples hints that automation should not be too difficult. We go into detail regarding such automation in Sect. 4.

Reasoning with fixpoints and functions like step_R can be thought of as reasoning with proof rules, but ones which interact with the target programming language only through its operational semantics. The step_R operation corresponds, conceptually, to two such proof rules: taking an execution step and

HIMP

```
append(x, y)
  decl p;
  if (!x) return y;
  p := x;
  while(*(p+1)<>0) p := *(p+1);
  *(p+1) := y;
  return x;
```

Stack

```
: append over if over begin
1+ dup @ dup while nip repeat
drop ! else nip then ;
```

Lambda

```
(λ (λ IfNil 1 0
 ((λ (λ 0 0)
     (λ 1 (λ 1 1 0))) (λ
  (λ (λ (λ 0 1)) (Deref 0)
    (λ IfNil (Cdr 0)
      ((λ 5) (Assign 0
        (Cons (Car 0) 3)))
      (2 (Cdr 0))))))
  1)))
```

Fig. 4. Destructive list append in three languages.

showing that the current configuration is in the target set. Sequential composition and the trans rule corresponds to a transitivity rule used to chain together separate proofs. Unions correspond to case analysis. The fixpoint in the closure definition corresponds to iterative uses of these proof rules or to referring back to claims in the original specification.

4 Experiments

Now that we have proved the correctness of our coinductive verification approach and have seen some simple examples, we must consider the following pragmatic question: "Can this simple approach really work?". We have implemented it in Coq, and specified and verified programs in a variety of languages, each language being defined as an operational semantics [16]. We show not only that coinductive program verification is feasible and versatile, but also that it is amenable to highly effective proof automation. The simplifications in the manual proof, such as taking many execution steps at once, translate easily into proof tactics.

We first discuss the example languages and programs, and the reusable elements in specifications, especially an effective style of representation predicates for heap-allocated data structures. Then we show how we wrote specifications for example programs. Next we describe our proof automation, which was based on an overall heuristic applied unchanged for each language, though parameterized over subroutines which required somewhat more customization. Finally, we conclude with discussion of our verification of the Schorr-Waite graph-marking example and a discussion of our support for verification of divergent programs.

4.1 Languages

We discuss three languages following different paradigms, each defined operationally. Many language semantics are available with the distributions of K [9], PLT-Redex [10], and Ott [11], e.g., but we believe these three languages are sufficient to illustrate the language-independence of our approach. Figure 4 shows a destructive linked list append function in each of the three languages.

HIMP (IMP with Heap) is an imperative language with (recursive) functions and a heap. The heap addresses are integers, to demonstrate reasoning about low-level representations, and memory allocation/deallocation are primitives. The configuration is a 5-tuple of current code, local variable environment mapping identifiers to values, call stack with frames as pairs of code and environment, heap, and a collection of functions as a map from function name to definition.

Stack is a Forth-like stack based language, though, unlike in Forth, we do make control structures part of the grammar. A shared data stack is used both for local state and to communicate between function invocations, eliminating the store, formal parameters on function declarations, and the environment of stack frames. Stack's configuration is also a 5-tuple, but instead of a current environment there is a stack of values, and stack frames do not store an environment.

Lambda is a call-by-value lambda calculus, extended with primitive integers, pair and nil values, and primitive operations for heap access. Fixpoint combinators enable recursive definitions without relying on primitive support for named functions. We use De Bruijn indices instead of named variables. The semantics is based on a CEK/CESK machine [21,22], extended with a heap. Lambda's configuration is a 4-tuple: current expression, environment, heap, continuation.

```
Pgm ::= FunDef*

FunDef ::=
      Id ( Id*, ) { Stmt }

Exp ::= Id ( Exp*, )
  | alloc | load Exp
  | Exp . Id
  | build Map
  | ...

Stmt ::= * Exp := Exp
  | dealloc Exp
  | Id ( Exp*, ) | decl Id
  | return Exp ;
  | return ;
  | ...
```

(a) **HIMP** syntax, extending the **IMP** syntax

```
Pgm ::= FunDef*

FunDef ::=
      name : Inst*

Inst ::= Dup n
  | Roll n
  | Pop | Push z
  | BinOp f
  | Load | Store
  | Call name | Ret
  | If Inst* Inst*
  | While
      Inst* Inst*
```

(b) **Stack** syntax

```
Pgm ::= Val

Val ::= Nat | Inc | Dec | Add
  | Add1 Nat | Eq | Eq Val
  | Nil | Cons | Cons1 Val
  | Car | Cdr
  | Closure (Exp, Env)
  | Pair (Val, Val)

Exp ::= Exp Exp | λ Exp
  | Var Nat
  | if Exp then Exp else Exp
  | Exp ; Exp | Deref Exp
  | & Exp | * Exp | Exp := Exp

Env ::= Val*
```

(c) **Lambda** syntax

Fig. 5. Syntax of **HIMP**, **Stack**, and **Lambda**

4.2 Specifying Data Structures

Our coinductive verification approach is agnostic to how claims in $C \times \mathcal{P}(C)$ are specified. In Coq, we can specify sets using any definable predicates. Within this design space, we chose matching logic [23] for our experiments, which introduces patterns that concisely generalize the formulae of first order logic (FOL) and separation logic, as well as term unification. Symbols apply on patterns to build other patterns, just like terms, and patterns can be combined using FOL connectives, just like formulae. E.g., pattern $P \wedge Q$ matches a value if P and Q both match it, $[t]$ matches only the value t, $\exists x.P$ matches if there is any assignment of x under which P matches, and $[\![\varphi]\!]$ where φ is a FOL formula matches any value if φ holds, and no values otherwise (in [23] neither $[t]$ nor $[\![\varphi]\!]$ require a visible marker, but in Coq patterns are a distinct type, requiring explicit injections).

To specify programs manipulating heap data structures we use patterns matching subheaps that contain a data structure representing an abstract value. Following [24], we define representation predicates for data structures as functions from abstract values to more primitive patterns. The basic ingredients are primitive map patterns: pattern **emp** for the empty map, $k \mapsto v$ for the singleton map binding key k to value v, and $P * Q$ for maps which are a disjoint union of submaps matching P and, resp., Q. We use abbreviation $\langle\varphi\rangle \equiv [\![\varphi]\!] \wedge \mathbf{emp}$ to facilitate inline assertions, and $p \mapsto \{v_0, \ldots, v_i\} \equiv p \mapsto v_0 * \ldots * (p+i) \mapsto v_i$ to describe values at contiguous addresses. A heap pattern for a linked list starting at address p and holding list l is defined recursively by

$$\begin{aligned} \mathrm{list}(\mathrm{nil}, p) &= \langle p = 0\rangle \\ \mathrm{list}(x : l, p) &= \langle p \neq 0\rangle * \exists p_l \,.\, p \mapsto \{x, p_l\} * \mathrm{list}(l, p_l) \end{aligned}$$

We also define list_seg(l, e, p) for list segments, useful in algorithms using pointers to the middle of a list, by generalizing the constant 0 (the pointer to the end of the list) to the trailing pointer parameter e. Also, simple binary trees:

$$\begin{aligned} \mathrm{tree}(\mathrm{leaf}, p) &= \langle p = 0\rangle \\ \mathrm{tree}(\mathrm{node}(x, l, r), p) &= \langle p \neq 0\rangle * \exists p_l, p_r . p \mapsto \{x, lp, rp\} * \mathrm{tree}(l, lp) * \mathrm{tree}(r, rp) \end{aligned}$$

Given such patterns, specifications and proofs can be done in terms of the abstract values represented in memory. Moreover, such primitive patterns are widely reusable across different languages, and so is our proof automation that deals with primitive patterns. Specifically, our proof scripting specific to such pattern definitions is concerned exclusively with unfolding the definition when allowed, deciding what abstract value, if any, is represented at a given address in a partially unfolded heap. This is further used to decide how another claim applies to the current state when attempting a transitivity step.

4.3 Specifying Reachability Claims

As mentioned, claims in $C \times \mathcal{P}(C)$ can be specified using any logical formalism, here the full power of Coq. An explicit specification can be verbose and low-level,

Table 1. Example list specifications

$$\begin{array}{l}
\mathit{call}(\text{Head}, [x], [H] \wedge \text{list}(v : l, x), \lambda r.\langle r = v\rangle * [H]) \\
\mathit{call}(\text{Tail}, [x], [H] \wedge \text{list}(v : l, x), \lambda r.[H] \wedge \_ * \text{list}(l, r)) \\
\mathit{call}(\text{Add}, [y, x], \text{list}(l, x), \lambda r.\text{list}(y : l, r)) \\
\mathit{call}(\text{Add}', [y, x], [H] \wedge \text{list}(l, x), \lambda r.\text{list\_seg}([y], x, r) * [H]) \\
\mathit{call}(\text{Swap}, [x], \text{list}(a : b : l, x), \lambda r.\text{list}(b : a : l, x)) \\
\mathit{call}(\text{Dealloc}, [x], \text{list}(l, x), \lambda r.\mathbf{emp}) \\
\mathit{call}(\text{Length}, [x], [H] \wedge \text{list}(l, x), \lambda r.\langle r = \mathit{len}(l)\rangle * [H]) \\
\mathit{call}(\text{Sum}, [x], [H] \wedge \text{list}(l, x), \lambda r.\langle r = \mathit{sum}(l)\rangle\rangle * [H]) \\
\mathit{call}(\text{Reverse}, [x], \text{list}(l, x), \lambda r.\text{list}(\mathit{rev}(l), r)) \\
\mathit{call}(\text{Append}, [x, y], \text{list}(a, x) * \text{list}(b, y), \lambda r.\text{list}(a{+\!\!+}b, r)) \\
\mathit{call}(\text{Copy}, [x], [H] \wedge \text{list}(l, x), \lambda r.\text{list}(l, r) * [H]) \\
\mathit{call}(\text{Delete}, [v, x], \text{list}(l, x), \lambda r.\text{list}(\mathit{delete}(v, l), r))
\end{array}$$

especially when many semantic components in the configuration stay unchanged. However, any reasonable logic allows making definitions to reduce verbosity and redundancy. Our use of matching logic particularly facilitates framing conditions, allowing us to regain the compactness and elegance of Hoare logic or separation logic specifications with definable syntactic sugar. For example, defining

$$\begin{array}{l}
\mathit{call}(f(\mathit{formals})\{\mathit{body}\}, \mathit{args}, P_{in}, P_{out}) = \\
\{(\langle f(\mathit{args}) \curvearrowright \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}, \mathit{funs}\rangle,\ \{\langle r \curvearrowright \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}', \mathit{funs}\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \forall r, \mathit{heap}'.\ \mathit{heap}' \vDash P_{out}(r) * [H_f]\}) \\
\mid \forall \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}, H_f, \mathit{funs}.\ \mathit{heap} \vDash P_{in} * [H_f] \wedge f {\mapsto} f(\mathit{formals})\{\mathit{body}\} \in \mathit{funs}\}
\end{array}$$

gives the equivalent of the usual Hoare pre-/post-condition on function calls, including heap framing (in separation logic style). The notation $x \curvearrowright y$ represents the order of evaluation: evaluate x first followed by y. This is often used when y can depend on the value x takes after evaluation.

The first parameter is the function definition. The second is the arguments. The heap effect is described as a pattern P_{in} for the allowable initial states of the heap and function P_{out} from returned values to corresponding heap patterns. For example, we specify the definition D of append in Fig. 4 by writing $\mathit{call}(D, [x, y], (\text{list}(a, x) * \text{list}(b, y)), (\lambda r.\text{list}(a{+\!\!+}b, r)))$, which is as compact and elegant as it can be. More specifications are given in Table 1. A number of specifications assert that part of the heap is left entirely unchanged by writing $[H] \wedge \ldots$ in the precondition to bind a variable H to a specific heap, and using the variable in the postcondition (just repeating a representation predicate might permit a function to reallocate internal nodes in a data structure to different addresses). The specifications Add and Add' show that it can be a bit more complicated to assert that an input list is used undisturbed as a suffix of a result list. Specifications such as Length, Append, and Delete are written in terms of corresponding mathematical functions on the lists represented in the heap, separating those functional descriptions from details of memory layout.

When a function contains loops, proving that it meets a specification often requires making some additional claims about configurations which are just about to enter loops, as we saw in Sect. 2.2. We support this with another pattern that takes the current code at an intermediate point in the execution of a function, and a description of the environment:

$$\begin{aligned}&\mathit{stmt}(\mathit{code}, \mathit{env}, P_{in}, P_{out}) = \\ &\{(\langle \mathit{code}, (\mathit{env}, e_f), \mathit{stk}, \mathit{heap}, \mathit{funs}\rangle, \{\langle \texttt{return}\ r \curvearrowright \mathit{rest}, \mathit{env}', \mathit{stk}, \mathit{heap}', \mathit{funs}\rangle \\ &\qquad\qquad \mid \forall r, \mathit{rest}, \mathit{env}', \mathit{heap}'.\mathit{heap}' \vDash P_{out}(r) * [H_f]\}) \\ &\qquad\qquad \mid \forall e_f, \mathit{stk}, \mathit{heap}, H_f, \mathit{funs}\ .\ \mathit{heap} \vDash P_{in} * [H_f]\}\end{aligned}$$

Verifying the definition of append in Fig. 4 meets the call specification above requires an auxiliary claim about the loop, which can be written using *stmt* as

$$\begin{aligned}&\mathit{stmt}(\texttt{while (*(p+1)<>0)}\ldots, (\texttt{x} \mapsto x, \texttt{y} \mapsto y, \texttt{p} \mapsto p), \\ &\quad (\text{list\_seg}(l_x, p, x) * \text{list}(l_p, p) * \text{list}(l_y, y)), (\lambda r.\text{list}(l_x +\!\!+ l_p +\!\!+ l_y, r)))\end{aligned}$$

The patterns above were described using HIMP's configurations; we defined similar ones for Stack and Lambda also.

4.4 Proofs and Automation

The basic heuristic in our proofs, which is also the basis of our proof automation, is to attack a goal by preferring to prove that the current configuration is in the target set if possible, then trying to use claims in the specification by transitivity, and only last resorting to taking execution steps according to the operational semantics or making case distinctions. Each of these operations begins, as in the example proofs, with certain manipulations of the definitions and fixpoints in the language-independent core. Our heuristic is reusable, as a proof tactic parameterized over sub-tactics for the more specific operations. A prelude to the main loop begins by applying the main theorem to move from claiming validity to showing a coinduction-style inclusion, and breaking down a specification with several classes of claims into a separate proof goal for each family of claims.

Additionally, our automation leverages support offered by the proof assistant, such as handling conjuncts by trying to prove each case, existentials by introducing a unification variable, equalities by unification, and so on. Moreover, we added tactics for map equalities and numerical formulae, which are shared among all languages involving maps and integers. The current proof goal after each step is always a reachability claim. So even in proofs which are not completely automatic, the proof automation can give up by leaving subgoals for the user, who can reinvoke the proof automation after making some proof steps of their own as long as they leave a proof goal in the same form.

Proving the properties in Table 1 sometimes required making additional claims about while loops or auxiliary recursive functions. All but the last four were proved automatically by invoking (an instance of) our heuristic proof tactic:

```
  Proof. list_solver. Qed.
```

Append and copy needed to make use of associativity of list append. Reverse used a loop reversing the input list element by element onto an output list, which required relating the tail recursive $\mathit{rev\_app}(x : l, y) = \mathit{rev\_app}(l, x : y)$ with the Coq standard library definition $\mathit{rev}(x : l) = \mathit{rev}(l) +\!\!+ [x]$. Manually applying these lemmas merely modified the proof scripts to

```
list_solver. rewrite app_ass in * |- . list_run.
list_solver. rewrite <- rev_alt in * |- . list_run.
```

These proofs were used verbatim in each of our example languages. The only exceptions were append and copy for Lambda, for which the `app_ass` lemma was not necessary. For Delete, simple reasoning about $\mathit{delete}(v, l)$ when v is and is not at the head of the list is required, though the actual reasoning in Coq varies between our example languages. No additional lemmas or tactics equivalent to Hoare rules are needed in any of these proofs.

4.5 Other Data Structures

Matching logic allows us to concisely define many other important data structures. Besides lists, we also have proofs in Coq with trees, graphs, and stacks [16]. These data structures are all used for proving properties about the Schorr-Waite algorithm. In the next section we go into more detail about these data structures and how they are used in proving the Schorr-Waite algorithm.

4.6 Schorr-Waite

Our experiments so far demonstrate that our coinductive verification approach applies across languages in different paradigms, and can handle usual heap programs with a high degree of automation. Here we show that we can also handle the famous Schorr-Waite graph marking algorithm [25], which is a well-known verification challenge, “The Schorr-Waite algorithm is the first mountain that any formalism for pointer aliasing should climb” [26]. To give the reader a feel for what it takes to mechanically verify such an algorithm, previous proofs in [27] and [28] required manually produced proof scripts of about 470 and, respectively, over 1400 lines and they both used conventional Hoare logic. In comparison our proof is 514 lines. Line counts are a crude measure, but we can at least conclude that the language independence and generality of our approach did not impose any great cost compared to using language-specific program logics.

The version of Schorr-Waite that we verified is based on [29]. First, however, we verify a simpler property of the algorithm, showing that the given code correctly marks a tree, in the absence of sharing or cycles. Then we prove the same code works on general graphs by considering the tree resulting from a depth first traversal. We define graphs by extending the definition of trees to allow a child of a node in an abstract tree to be a reference back to some existing node, in addition to an explicit subtree or a null pointer for a leaf. To specify that graph nodes are at their original addresses after marking, we include an address along with the mark flag in the abstract data structure in the pattern

$$\begin{aligned}
&\mathrm{grph}(\mathrm{leaf}, m, p') = \langle p' = 0\rangle\\
&\mathrm{grph}(\mathrm{backref}(p), m, p') = \langle p' = p\rangle\\
&\mathrm{grph}(\mathrm{node}(p, l, r), m, p') = \langle p'{=}p\rangle * \exists p_l, p_r\,.\\
&\qquad p \mapsto \{m, p_l, p_r\} * \mathrm{grph}(l, m, p_l) * \mathrm{grph}(r, m, p_r)
\end{aligned}$$

The overall specification is $\mathit{call}(\mathit{Mark}, [p], \mathrm{grph}(G, 0, p), \lambda r.\mathrm{grph}(G, 3, p))$.

To describe the intermediate states in the algorithm, including the clever pointer-reversal trick used to encode a stack, we define another data structure for the context, in zipper style. A position into a tree is described by its immediate context, which is either the topmost context, or the point immediately left or right of a sibling tree, in a parent context. These are represented by nodes with intermediate values of the mark field, with one field pointing to the sibling subtree and the other pointing to the representation of the rest of the context.

$$\begin{aligned}
\mathrm{stack}(\mathrm{Top}, p) &= \langle p = 0\rangle\\
\mathrm{stack}(\mathrm{LeftOf}(r, k), p) &= \exists p_r, p_k\,.\ p \mapsto \{1, p_r, p_k\} * \mathrm{grph}(r, 0, p_r) * \mathrm{stack}(k, p_k)\\
\mathrm{stack}(\mathrm{RightOf}(l, k), p) &= \exists p_l, p_k\,.\ p \mapsto \{2, p_k, p_l\} * \mathrm{stack}(k, p_k) * \mathrm{grph}(l, 3, p_l)
\end{aligned}$$

This is the second data structure needed to specify the main loop. When it is entered, there are only two live local variables, one pointing to the next address to visit and the other keeping context. The next node can either be the root of an unmarked subtree, with the context as stack, or the first node in the implicit stack when ascending after marking a tree, with the context pointing to the node that was just finished. For simplicity, we write a separate claim for each case.

$$\begin{aligned}
&\mathit{stmt}(\mathit{Loop}, (\mathtt{p} \mapsto p, \mathtt{q} \mapsto q), (\mathrm{grph}(G, 0, p) * \mathrm{stack}(S, q)), \lambda r.\mathrm{grph}(\mathit{plug}(G, S), 3))\\
&\mathit{stmt}(\mathit{Loop}, (\mathtt{p} \mapsto p, \mathtt{q} \mapsto q), (\mathrm{stack}(S, p) * \mathrm{grph}(G, 3, q)), \lambda r.\mathrm{grph}(\mathit{plug}(G, S), 3))
\end{aligned}$$

The application of all the semantic steps was handled entirely automatically, the manual proof effort being entirely concerned with reasoning about the predicates above, for which no proof automation was developed.

4.7 Divergence

Our coinductive framework can also be used to verify a program is divergent. Such verification is often a topic that is given its own treatment, as in [30,31], though in our framework, no additional care is needed. To prove a program is divergent on all inputs, one verifies a set of claims of the form $(c, \emptyset)$, so that no configuration can be determined valid by membership in the final set of states. We have verified the divergence of a simple program under each style of IMP semantics in Fig. 3, as well as programs in each language from Sect. 4.1. These program include the omega combinator and the `sum` program from Sect. 3.2 with `true` replacing the loop guard.

4.8 Summary of Experiments

Statistics are shown in Table 2. For each example, size shows the amount of code to be verified, the size of the specification, and the size of the proof script. If verifying an example required auxiliary definitions or lemmas specific to that example, the size of those definitions were counted with the specification or proof. Many examples were verified by a single invocation of our automatic proof tactic, giving 1-line proofs. Other small proofs required human assistance only in the form of applying lemmas about the domain. Proofs are generally smaller than the specifications, which are usually about as large as the code. This is similar to the results for Bedrock [32], and good for a foundational verification system.

Table 2. Proof statistics

	Size (lines)			Time (s)	
Example	Code	Spec	Proof	Prove	Check
Simple					
undefined	2	3	1	2.1	1.1
average3	2	5	1	2.3	0.8
min	3	4	2	2.1	0.7
max	3	4	2	2.1	0.7
multiply	9	6	1	7.2	1.4
sum(rec)	6	7	6	4.2	1.0
sum(iter)	6	11	8	6.0	1.0
Trees					
height	8	3	3	20.5	4.1
size	5	3	1	8.0	2.2
find	6	9	1	15.5	3.1
mirror	7	6	1	19.0	4.2
dealloc	15	7	1	19.6	4.1
flatten(rec)	12	10	1	30.9	6.8
flatten(iter)	24	17	4	150.3	22.8
Lists: head	2	4	1	2.1	0.8
tail	2	4	1	2.2	0.9
add	4	4	1	4.8	1.2
swap	6	4	1	19.6	3.6
dealloc	6	4	1	6.3	1.3
length(rec)	4	4	1	4.8	1.4
length(iter)	4	8	1	7.2	1.5
sum(rec)	4	4	1	8.2	2.0
sum(iter)	4	8	1	9.11	1.7
reverse	8	5	3	15.0	2.2
append	7	9	3	19.4	3.6
copy	14	11	3	55.0	9.3
delete	16	18	9	44.6	6.0
Schorr-Waite					
tree	14	91	116	60.1	7.6
graph	14	91	203	133.6	18.2

The reported "Proof" time is the time for Coq to process the proof script, which includes running proof tactics and proof searches to construct a complete proof. If this run succeeds, it produces a proof certificate file which can be rechecked without that overhead. For an initial comparison with Bedrock we timed their `SinglyLinkedList.v` example, which verifies length, reverse, and append functions that closely resemble our example code. The total time to run the Bedrock proof script was 93 s, and 31 s to recheck the proof certificate, distinctly slower than our times in Table 2. To more precisely match the Bedrock examples we modified our programs to represent lists nodes with fields at successive addresses rather than using HIMP's records, but this only improved performance, down to 20 s to run the proof scripts, and 4 s to check the certificates.

5 Subsuming Reachability Logic

Reachability logic [33] is a closely related approach to program verification using operational semantics. In fact, our coinductive approach came about when trying to distill reachability logic into its mathematical essence. The practicality of reachability logic has recently been demonstrated, as the reachability logic proof system has been shown to work with several independently developed semantics of real-world languages, such as C, Java, and JavaScript [15].

5.1 Advantages of Coinduction

A mechanical proof of our soundness theorem gives a more usable verification framework, since reachability logic requires operational semantics to be given as a set of rewrite rules, while our approach does not. Further, reachability logic fixes a set of syntactic proof rules, while in our approach the mathematical fixpoints and functions act as proof rules without explicitly requiring any. In fact, the generality of our approach allows introductions of other derived rules that do not compromise the soundness result. Similarly, the generality allows higher-order verification, which reachability logic cannot handle.

Further, we saw in Sect. 3 that the general proof of our theorem is entirely mathematical. We instantiate it with the step_R function to get a program verification framework. However, if we instantiate it with other functions, we could get frameworks for proving different properties, such as all-path validity or the "until" notion of validity previously mentioned. Reachability logic does not support any other notion of validity without changes to its proof system, which then require new proofs of soundness and relative

Axiom :

$$\frac{\varphi \Rightarrow \varphi' \ \in \mathcal{A}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Reflexivity :

$$\mathcal{A} \vdash \varphi \Rightarrow \varphi$$

Transitivity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^+ \varphi_2 \qquad \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_3}$$

Logic Framing :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad \psi \text{ is a FOL formula}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_2}$$

Case Analysis :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Abstraction :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash_{\mathcal{C}} \exists X\ \varphi \Rightarrow \varphi'}$$

Circularity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Fig. 6. Reachability Logic proof system. Sequent $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is a shorthand for $\mathcal{A} \vdash_{\emptyset} \varphi \Rightarrow \varphi'$.

completeness. For our framework, the proof of the main theorem does not need to be modified at all, and one only needs to prove that all-path validity is a greatest fixpoint (see Sect. 3). The same is true for any property. In this sense, this coinduction framework is much more general than the reachability logic proof system presented in [34].

5.2 Reachability Logic Proof System

The key construct in reachability logic is the notion of circularity. Circularities, represented as $\mathcal{C}$ in Fig. 6, intuitively represent claims that are conjectured to be true but have not yet been proved true. These claims are proved using the Circularity rule, which is analogous in our coinductive framework to referring back to claims previously seen. Most of the other rules in Fig. 6 are not as interesting. Transitivity requires progress before the circularities are flushed as axioms. This corresponds to the outer step_R in our coinductive framework.

Clearly, there are obvious parallels between the Reachability Logic proof system and our coinductive framework. We have formalized and mechanically verified a detailed proof that reachability logic is an instance of our coinductive verification framework. One can refer to [16] for full details, but we briefly discuss the nature of the proof below.

5.3 Reachability Logic is Coinduction

To formalize what it means for reachability logic to be an instance of coinduction, we first need some definitions. First, we need a translation from a reachability rule to a set of coinductive claims. In a reachability rule $\varphi \Rightarrow \varphi'$, both φ and φ' are patterns which respectively describe (symbolically) the starting and the reached configurations. Both φ and φ' can have free variables. Let *Var* be the set of variables. Then, we define the set of claims

$$S_{\varphi \Rightarrow \varphi'} \equiv \{(c, \overline{\rho}(\varphi')) \mid c \in \overline{\rho}(\varphi),\ \forall \rho : \mathit{Var} \to \mathit{Cfg}\}$$

where *Cfg* is the model of configurations and $\overline{\rho}(\cdot)$ is the extension of the valuation ρ to patterns [15]. Also, let the claims derived from a set of reachability rules $X = \{\varphi_1 \Rightarrow \varphi'_1, \ldots, \varphi_n \Rightarrow \varphi'_n\}$ be:

$$\overline{X} \equiv \bigcup_{\varphi_i \Rightarrow \varphi'_i \in X} S_{\varphi_i \Rightarrow \varphi'_i}$$

In reachability logic, programming language semantics are defined as *theories*, that is, as sets of (one-step) reachability rules $\mathcal{A}$ with patterns over a given signature of symbols. Each theory $\mathcal{A}$ defines a transition relation over the configurations in *Cfg*, say $R_{\mathcal{A}}$, which is then used to define the semantic validity in reachability logic, $\mathcal{A} \models \varphi \Rightarrow \varphi'$. It is possible and easier to prove our main theorem more generally, for any transition relation R that satisfies $R \vDash^{+} \mathcal{A}$:

$$R \vDash^{+} \mathcal{A} \text{ if } R \vDash^{+} \varphi \Rightarrow \varphi' \text{ for each } \varphi \Rightarrow \varphi' \in \mathcal{A}$$

where $R \vDash^{+} \varphi \Rightarrow \varphi'$ if for each $\rho : \mathit{Var} \to \mathit{Cfg}$ and $\gamma : \mathit{Cfg}$ such that $(\rho, \gamma) \vDash \varphi$ [33], there is a γ' such that $\gamma \to_R \gamma'$ and $(\gamma', \overline{\rho}(\varphi'))$ is a valid reachability claim.

Lemma 3. *$R_{\mathcal{A}} \vDash^{+} \mathcal{A}$ and if $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_{R_{\mathcal{A}}}$ then $\mathcal{A} \vDash \varphi \Rightarrow \varphi'$.*

This lemma suggests what to do: take any reachability logic proof of $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ and any transition relation R such that $R \vDash^{+} \mathcal{A}$, and produce a coinductive proof of $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_R$. This gives us not only a procedure to associate coinductive proofs to reachability logic proofs, but also an alternative method to prove the soundness of reachability logic. This is what we do below:

Theorem 2. *If there is a reachability logic proof derivation for $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ and a transition relation R such that $R \vDash^{+} \mathcal{A}$, then $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_R$, and in particular this holds by applying Theorem 1 to an inclusion $\overline{\mathcal{C}} \subseteq \text{step}_R(\text{derived}^*_R(\overline{\mathcal{C}}))$. Here, derived_R is a particular function satisfying the conditions for G in Theorem 1 (see [16] for more details), and $\mathcal{C}$ is a set of reachability rules consisting of $\varphi \Rightarrow \varphi'$ along with those reachability rules which appear as conclusions of instances of the Circularity proof rule in the proof tree of $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$.*

To prove Theorem 2, we apply the Set Circularity theorem of reachability logic [35], which states that any reachability logic claim $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is provable iff there is some set of claims $\mathcal{C}$ such that $\varphi \Rightarrow \varphi' \in \mathcal{C}$ and for each $\varphi_i \Rightarrow \varphi'_i \in \mathcal{C}$ there is a proof of $\mathcal{A} \vdash_{\mathcal{C}} \varphi_i \Rightarrow \varphi'_i$ which does not use the Circularity proof rule. In the forward direction, we can take $\mathcal{C}$ as defined in the statement of Theorem 2. The main idea is to convert proof trees into inclusions of sets of claims:

Lemma 4. *Given a proof derivation of $\mathcal{A} \vdash_{\mathcal{C}} \varphi_a \Rightarrow \varphi_b$ which does not use the Circularity proof rule (last rule in Fig. 6), if $R \vDash^{+} \mathcal{A}$ and $\mathcal{C}$ is nonempty then $S_{\varphi_a \Rightarrow \varphi_b} \subseteq \text{step}_R(\text{derived}^*_R(\overline{\mathcal{C}}))$.*

This lemma is proven by strengthening the inclusion into one that can be proven by structural induction over the Reachability Logic proof rules besides Circularity.

Combining this lemma with Set Circularity shows that $\overline{\mathcal{C}} = \cup_i S_{\varphi_i \Rightarrow \varphi'_i} \subseteq \text{valid}_R$ which implies that $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_R$ exactly as desired. We have mechanized the proofs of Lemmas 3 and 4 in Coq [16]. This is a major result, constituting an independent soundness proof for Reachability Logic, and helps demonstrate the strength of our coinductive framework, despite its simplicity. Moreover, this allows proofs done using reachability logic as in [15] to be translated to mechanically verified proofs in Coq, immediately allowing foundational verification of programs written in *any language*.

6 Other Related Work

Here we discuss work other than reachability logic that is related to our coinductive verification system. We discuss commonly used program verifiers, including approaches based on operational semantics and Iris [36], an approach with some language independence. We also discuss related coinduction schemata.

6.1 Current Verification Tools

A number of prominent tools such as Why [37], Boogie [38,39], and Bedrock [24,32] provide program verification for a fixed language, and support other languages by translation if at all. For example, Frama-C and Krakatoa, respectively, attempt to verify C and Java by translation through Why. Also, Spec# and Havoc, respectively, verify C# and C by translation through Boogie. We are not aware of soundness proofs for these translations. Such proofs would be highly non-trivial, requiring formal semantics of both source and target languages.

All of these systems are based on a verification condition (VC) generator for their programming language. Bedrock is closest in architecture and guarantees to our system, as it is implemented in Coq and verification results in a Coq proof certificate that the specification is sound with respect to a semantics of the object language. Bedrock supports dynamically created code, and modular verification of higher-order functions, for which our framework has preliminary support. Bedrock also makes more aggressive attempts at complete automation, which costs increased runtime. Most fundamentally, Bedrock is built around a VC generator for a fixed target language.

In sharp contrast to the above approaches, we demonstrated that a small-step operational semantics suffices for program verification, without a need to define any other semantics, or verification condition generators, for the same language. A language-independent, sound and (relatively) complete coinductive proof method then allows us to verify properties of programs using directly the operational semantics. As seen in Sect. 4.8 this language independence does not compromise other desirable properties. The required human effort and the performance of the verification task compare well with foundational program verifiers such as Bedrock, and we provide the same high confidence in correctness: the trust base consists of the operational semantics only.

6.2 Operational Semantics Based Approaches

Verifiable C [40] is a program verification tool for the C programming language based on an operational semantics for C defined in Coq. Hoare triples are then proved as lemmas about the operational semantics. However, in this approach and other similar approaches, it is necessary to prove such lemmas. Without them, verification of any nontrivial C program would be nearly impossible. In our approach, while we can also define and prove Hoare triples as lemmas, doing so is not needed to make program verification feasible, as demonstrated in the previous sections. We only need some additional domain reasoning in Coq, which logics like Verifiable C require *in addition* to Hoare logic reasoning. Thus, our approach automatically yields a program verification tool for any language with minimal additional reasoning, while approaches such as Verifiable C need over 40,000 lines of Coq to define the program logic. We believe this is completely unnecessary, and hope our coinductive framework will be the first step in eliminating such superfluous logics.

The work by the FLINT group [41–43] is another approach to program verification based on operational semantics. Languages developed use shallowly embedded state predicates in Coq, and inference rules are derived directly from the operational semantics. However, their work is not generic over operational semantics. For example, [43] is developed in the context of a particular machine model, with a fixed memory representation and register file. Even simple changes such as adding registers require updating soundness proofs. Our approach has a single soundness theorem that can be instantiated for *any* language.

Iris [36] is a concurrent separation logic that has language independence, with operational semantics formalized in Coq. Iris adds monoids and invariants to the program logic in order to facilitate verification. It also derives some Hoare-style rules for verification from the semantics of a language. However, there are still structural Hoare rules that depend on the language that must be added manually. Additionally, once proof rules are generated, they are specialized to that particular language. Further, the verification in the paper relies on Hoare style reasoning, while in our approach, we do not assume any such verification style, as we work directly with the mathematical specifications. Finally, the monoids used are not generated and are specific to the program language used.

6.3 Other Coinduction Schemata

A categorical generalization of our key theorem was presented as a recursion scheme in [12,13]. The titular result of the former is the dual of the λ-coiteration scheme of the latter, which specializes to preorder categories to give our Theorem 1. A more recent and more general result is [14], which also generalized other recent work on coinductive proofs such as [44]. Unlike these approaches, which were presented for showing bisimilarity, the novelty of our approach stems in the use of these techniques directly to show Hoare-style functional correctness claims, and in the development of the afferent machinery and automation that makes it work with a variety of languages, and not in advancing the already solid mathematical foundations of coinduction. Various weaker coinduction schemes are folklore, such as Isabelle/HOL's standard library's lemma `coinduct3`: $\mathit{mono}(f) \wedge A \subseteq f(\mu x.\, f(x) \cup A \cup \nu f) \implies A \subseteq \nu(f)$.

7 Conclusion and Future Work

We presented a language-independent program verification framework. Proofs can be as simple as with a custom Hoare logic, but only an operational semantics of the target language is required. We have mechanized a proof of the correctness of our approach in Coq. Combining this with a coinductive proof thus produces a Coq proof certificate concluding that the program meets the specification according to the provided semantics. Our approach is amenable to proof automation. Further automation may improve convenience and cannot compromise soundness of the proof system. A language designer need only give an authoritative semantics to enable program verification for a new language, rather than needing to have the experience and invest the effort to design and prove the soundness of a custom program logic.

One opportunity for future work is using our approach to provide proof certificates for reachability logic program verifiers such as K [9]. The K prover was used to verify programs in several real programming languages [15]. While the proof system is sound, trusting the results of these tools requires trusting the implementation of the K system. Our translation in Sect. 5 will allow us to produce proof objects in Coq for proofs done in K's backend, which will make it sufficient to trust only Coq's proof checker to rely on the results from K's prover.

Another area for future work is verifying programs with higher-order specifications, where a specification can make reachability claims about values quantified over in the specification. This allows higher-order functions to have specifications that require functional arguments to themselves satisfy some specification. We have begun preliminary work on proving validity of such specifications using the notions of compatibility up-to presented in [14]. Combining this with more general forms of claims may allow modular verification of concurrent programs, as in RGsep [45]. See [16] for initial work in these areas.

Other areas for future work are evaluating the reusability of proof automation between languages, and using the ability to easily verify programs under a modified semantics, e.g. adding time costs to allow proving real-time properties.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
2. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: PLDI, pp. 336–345. ACM (2015). https://doi.org/10.1145/2737924.2737979
3. Bogdănaş, D., Roşu, G.: K-Java: a complete semantics of Java. In: POPL, pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
4. Bodin, M., Chargueraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaScript specification. In: POPL, pp. 87–100. ACM (2014). https://doi.org/10.1145/2535838.2535876
5. Park, D., Stefănescu, A., Roşu, G.: KJS: a complete formal semantics of Javascript. In: PLDI, pp. 346–356. ACM (2015). https://doi.org/10.1145/2737924.2737991
6. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. In: OOPSLA, pp. 217–232. ACM (2013). https://doi.org/10.1145/2509136.2509536
7. Filaretti, D., Maffeis, S.: An executable formal semantics of PHP. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 567–592. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_23
8. Owens, S.: A sound semantics for OCaml$_{light}$. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_1
9. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. LAP **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012

10. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run your research: on the effectiveness of lightweight mechanization. In: POPL, pp. 285–296. ACM (2012). https://doi.org/10.1145/2103656.2103691
11. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: effective tool support for the working semanticist. In: ICFP. ACM (2007). https://doi.org/10.1017/S0956796809990293
12. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nord. J. Comput. **8**(3), 366–390 (2001)
13. Bartels, F.: On generalised coinduction and probabilistic specification formats: distributive laws in coalgebraic modelling. Ph.D. thesis, Vrije Universiteit Amsterdam (2004)
14. Pous, D.: Coinduction all the way up. In: LICS, pp. 307–316. IEEE (2016). https://doi.org/10.1145/2933575.2934564
15. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: OOPSLA, pp. 74–91. ACM (2016). https://doi.org/10.1145/2983990.2984027
16. Moore, B., Peña, L., Rosu, G.: GitHub repository (2017). https://github.com/Formal-Systems-Laboratory/coinduction. Source code
17. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1992). https://doi.org/10.1006/inco.1994.1093
18. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Program. **60–61**, 17–139 (2004). https://doi.org/10.1016/j.jlap.2004.05.001
19. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE (2002). https://doi.org/10.1109/LICS.2002.1029817
20. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symbolic Log. **5**(2), 215–244 (1999). https://doi.org/10.2307/421090
21. Felleisen, M., Friedman, D.P.: A calculus for assignments in higher-order languages. In: POPL, p. 314. ACM (1987). https://doi.org/10.1145/41625.41654
22. Felleisen, M.: The calculi of Lambda-ν-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana University (1987)
23. Roşu, G.: Matching logic – extended abstract. In: RTA, LIPIcs, pp. 5–21. Schloss Dagstuhl-LZ I (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.5
24. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI, pp. 234–245. ACM (2011). https://doi.org/10.1145/1993498.1993526
25. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM **10**(8), 501–506 (1967). https://doi.org/10.1145/363534.363554
26. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). https://doi.org/10.1007/10722010_8
27. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2), 200–227 (2005). https://doi.org/10.1016/j.ic.2004.10.007
28. Hubert, T., Marche, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: SEFM, pp. 190–199. IEEE (2005). https://doi.org/10.1109/SEFM.2005.1
29. Gries, D.: The Schorr-Waite graph marking algorithm. Acta Informatica **11**(3), 223–232 (1979). https://doi.org/10.1007/BF00289068

30. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL, pp. 147–158. ACM (2008). https://doi.org/10.1145/1328438.1328459
31. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_11
32. Chlipala, A.: The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In: ICFP, pp. 391–402. ACM (2013). https://doi.org/10.1145/2500365.2500592
33. Roşu, G., Ştefănescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: LICS, pp. 358–367. IEEE (2013). https://doi.org/10.1109/LICS.2013.42
34. Roşu, G., Ştefănescu, A.: From Hoare logic to matching logic reachability. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 387–402. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_32
35. Roşu, G., Ştefănescu, A., Ciobâcă, c., Moore, B.M.: Reachability logic. Technical report, University of Illinois, July 2012. http://hdl.handle.net/2142/32952
36. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL, pp. 637–650. ACM (2015). https://doi.org/10.1145/2775051.2676980
37. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
38. Leino, K.R.M.: This is Boogie 2. Technical report, Microsoft Research, June 2008
39. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
40. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, New York (2014)
41. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: ICFP, pp. 175–188. ACM (2004). https://doi.org/10.1145/1016850.1016875
42. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: PLDI, pp. 401–414. ACM (2006). https://doi.org/10.1145/1133981.1134028
43. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Combining domain-specific and foundational logics to verify complete software systems. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 54–69. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87873-5_8
44. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: POPL, pp. 193–206. ACM (2013)
45. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)