Reasoning About Iteration and Recursion Uniformly Based on Big-Step Semantics

Ximeng Li[1,3](✉), Qianying Zhang[2], Guohui Wang[2], Zhiping Shi[1](✉), Yong Guan[3](✉)

[1] Beijing Key Laboratory of Electronic System Reliability and Prognostics, Capital Normal University, Beijing, China
{lixm,shizp}@cnu.edu.cn

[2] Beijing Engineering Research Center of High Reliable Embedded System, Capital Normal University, Beijing, China

[3] Beijing Advanced Innovation Center for Imaging Theory and Technology, Capital Normal University, Beijing, China
guanyong@cnu.edu.cn

Abstract. A reliable technique for deductive program verification should be proven sound with respect to the semantics of the programming language. For each different language, the construction of a separate soundness proof is often a laborious undertaking. In language-independent program verification, common aspects of computer programs are addressed to enable sound reasoning for all languages. In this work, we propose a solution for the sound reasoning about iteration and recursion based on the big-step operational semantics of any programming language. We give inductive proofs on the soundness and relative completeness of our reasoning technique. We illustrate the technique at simplified programming languages of the imperative and functional paradigms, with diverse features. We also mechanize all formal results in the Coq proof assistant.

1 Introduction

It is commonly accepted that a reliable technique for deductive program verification should be designed with the formal semantics of the programming language as foundation. With the formal semantics used as axioms, a mathematical proof of a desired property for the target program can be constructed. Direct program proofs based on operational semantics are often cumbersome. Due to language constructs that may incur unbounded program behavior, inductive proofs along the structure of semantic derivations (e.g., [27]) are expected.

An established method for simplifying the verification is by devising a program logic (e.g., [18,34]) for the programming language. Program logics effectively reduce the burdens in dealing with many aspects of the verification, such as the reasoning about loops, recursive function calls, memory layout of objects, concurrency, etc. The effectiveness of program logics has been demonstrated by powerful tools (e.g., [6,9,10,20]) and significant projects (e.g., [30,37]).

A price to pay for enjoying the power of program logics, however, is the considerable amount of effort often needed in establishing their soundness and

S. Qin et al. (Eds.): SETTA 2021, LNCS 13071, pp. 61–80, 2021.
https://doi.org/10.1007/978-3-030-91265-9_4

completeness wrt. the baseline semantics – often an operational semantics. There have been a plethora of programming languages designed and implemented to meet the needs of different domains. The recent development of blockchain technology alone has led to the creation of multiple languages, such as Solidity [5], Yul [7], Scilla [36], Move [3], Michelson [2], EVM bytecode language [41], etc. Developing one program logic for each language that could be used in scenarios where correctness is of serious concern would require a huge amount of efforts.

To combat the cumbersomeness of direct program proofs based on operational semantics, while avoiding the full complexity in the development of program logics, one could seek to establish the infrastructure necessary for reasoning about specific kinds of language features, for any languages with those features. The results in [26] and [25] show how to deal with fundamental language features that may cause unbounded behavior, such as iteration and recursion, in a language-independent fashion. In [26], a technique is proposed to generate inductive invariants from annotated loop invariants. In [25], a method is presented to turn the semantics of a programming language into a program verifier by applying coinductive reasoning principles. Both developments are built on the small-step execution relation of programs.

Small-step semantics [31] is known to be a fine-grained approach to the definition of operational semantics. It supports a way to model concurrent execution. It also enables the differentiation of looping and abnormal termination. Big-step semantics (or natural semantics [15,21]), on the other hand, can be easier to formulate. For instance, the design of the semantic configurations need not track the intermediate control states. Big-step semantics can also be easier to use. It does not require the consideration of both derivation sequences and derivation trees at the same time, in performing proofs. There exist many formalizations of big-step semantics (e.g., [4,11,17,23,28,42]) with practical uses.

In this work, we propose a technique for reasoning about iteration and recursion in deductive program verification based on big-step operational semantics. For any programming language with a big-step semantics, once a generic predicate is defined to hold on the premises and corresponding conclusions for the semantic rules, a theorem becomes available – the theorem turns the verification of partial correctness results into symbolic execution of the target program with auxiliary information from the user specification. For loops and recursive function calls, this auxiliary information is provided in the same form via the specification, enabling the same pattern of reasoning. We illustrate our technique using verification tasks involving simplified imperative and functional languages. We mechanize the proofs of all formal results [8] in the Coq proof assistant [1].

The main technical contributions of this article are:

– a language-independent technique simplifying the deductive verification of iterative and recursive program structures based on big-step semantics,
– proofs for the soundness and relative completeness of the technique,
– illustration of the technique with the verification of example programs in simplified programming languages of different paradigms,
– mechanization of proofs and verification examples in the Coq proof assistant.

We provide an infrastructure that handles the routine part of the work in reasoning about programming constructs with potentially unbounded behavior, based on a common model of big-step execution in a proof assistant. This provides a basis for a language-independent deductive program verifier.

Structure. The remaining part of this article is structured as follows. In Sect. 2, we introduce the reasoning technique, and prove its soundness. In Sect. 3, we illustrate the technique with a toy example that is developed in detail. In Sect. 4, we present further verification examples targeting simplified imperative and functional languages. In Sect. 5, we discuss the completeness of the technique. In Sect. 6, we discuss related work. In Sect. 7, we discuss potential extensions of the current development. Finally, we conclude in Sect. 8.

2 The Technique

The proposed verification technique can be used to check that the potential execution results of a program satisfy pre-specified conditions. The potential execution results are estimated by a combination of concrete computation according to the big-step semantics of the programming language, and abstract inference according to the auxiliary information in the specification. The abstract inference helps realize what is usually accomplished with loop invariants in reasoning about loops, and with function contracts in reasoning about function calls.

2.1 Specifications

We capture the execution status of programs by *configurations*. We capture the results of program execution by *result configurations*. For imperative languages, a configuration can be a pair of a program and a state, and a result configuration can be a state. For functional languages, a configuration can be a functional expression, and a result configuration can be a canonical form.

Let C be the set of all possible configurations ranged over by c, for programs written in some language. Let R be the set of all possible result configurations ranged over by r, for programs in the same language. We do not rely on any assumptions about the structure of the elements in C or in R.

A *specification* is a function $\Phi \in C \to \mathcal{P}(R)$. For a configuration c, if c contains the complete program to be verified, then $\Phi(c)$ is the set of result configurations capturing the required range for the execution results of the program. Otherwise, $\Phi(c)$ is the expected set of potential results obtained by executing some statement within the overall program. This set provides auxiliary information for the verification.

2.2 Semantic Derivation and Correctness

We model the set of rules of a big-step operational semantics by a predicate $\mathit{rule} \in (C \times R)^* \to (C \times R) \to \{\mathit{tt}, \mathit{ff}\}$. Each semantic rule is captured as

$$\mathit{rule}\ [(c_1, r_1), \ldots, (c_n, r_n)]\ (c, r)$$

Here, the list $[(c_1, r_1), \ldots, (c_n, r_n)]$ models the list of premises of the rule, and (c, r) models the conclusion of the rule. Each premise or conclusion consists of a configuration in the set C and a corresponding result configuration in the set R. A side condition in a semantic rule can be captured by a condition on the parameters $c_1, \ldots, c_n, r_1, \ldots, r_n, c$, and r, in a concrete definition of *rule*.

A semantic derivation concluding that the configuration c can be evaluated to the result configuration r in the big-step semantics is captured by

$$\begin{aligned} \mathit{deriv}(c,r) := \exists k\colon \exists c_1,\ldots,c_k\colon \exists r_1,\ldots,r_k\colon \\ \mathit{rule}\ [(c_1,r_1),\ldots,(c_k,r_k)]\ (c,r) \wedge \\ \forall i \in \{1,\ldots,k\}\colon \mathit{deriv}(c_i,r_i) \end{aligned}$$

Hence, the configuration c can be evaluated to the result configuration r, or (c, r) can be derived in the big-step semantics, if there is a semantic rule with (c, r) as conclusion, and each premise of the rule can itself be derived in the big-step semantics. Intuitively, if $\mathit{deriv}(c, r)$ can be established, then there is a finite derivation tree rooted at (c, r).

With the notion of semantic derivation defined above, we formalize the notion of partial correctness as the validity of specifications.

$$\mathit{valid}(\Phi) := \forall c, r : \mathit{deriv}(c, r) \Rightarrow r \in \Phi(c)$$

A specification Φ is valid, if for each configuration c, any result configurations semantically derivable from c is a member of $\Phi(c)$.

2.3 Specification-Aware Inference and Verification

We infer the potential execution results of a configuration under a given specification Φ according to the following definition.

$$\begin{aligned} \mathit{infer}^{\Phi}(c,r) := \exists k\colon \exists c_1,\ldots,c_k\colon \exists r_1,\ldots,r_k\colon \\ \mathit{rule}\ [(c_1,r_1),\ldots,(c_k,r_k)]\ (c,r) \wedge \\ \forall i \in \{1,\ldots,k\} : \mathit{res}^{\Phi}(c_i,r_i) \\ \mathit{res}^{\Phi}(c,r) := r \in \Phi(c) \wedge (\Phi(c) = R \Rightarrow \mathit{infer}^{\Phi}(c,r)) \end{aligned}$$

The result configuration r is infered from the configuration c with the help of the specification Φ, if there is a semantic rule with (c, r) as conclusion, and for each premise (c_i, r_i) of the semantic rule, r_i is a potential result for c_i according to Φ, as is captured by the auxiliary predicate res^{Φ}. The expression $\mathit{res}^{\Phi}(c_i, r_i)$ says that the possible candidates for r_i are constrained by the information contained in the specification about c_i. In addition, if Φ does not provide any useful information about c_i (i.e., $\Phi(c_i) = R$), then r_i should be inferable from c_i.

Intuitively, the application of the semantic rules in the inference corresponds to the symbolic execution of the target program. The information in the specification can be used to overcome the inability to symbolically execute the constructs with potentially unbounded behavior, such as iteration and recursion.

We formulate the condition to be verified on specifications Φ using the predicate *verif*. In other words, $\mathit{verif}(\Phi)$ is the syntactical correctness condition.

$$\mathit{verif}(\Phi) := \forall c, r : \mathit{infer}^{\Phi}(c, r) \Rightarrow r \in \Phi(c)$$

A specification Φ is verified, if for each configuration c, any result configurations that can be infered from c with the help of Φ are contained in $\Phi(c)$.

2.4 Soundness

We prove the implication from $\mathit{verif}(\Phi)$ to $\mathit{valid}(\Phi)$. The following lemma is a key component of this proof.

Lemma 1. *If $\mathit{verif}(\Phi)$ holds, and $\mathit{deriv}(c, r)$ holds, then $\mathit{infer}^{\Phi}(c, r)$ holds.*

Proof. According to the definition of $\mathit{deriv}(c, r)$, if this predicate holds, then there is a finite derivation tree generated by the following inference rule.

$$\frac{\mathit{deriv}(c_1, r_1) \quad \ldots \quad \mathit{deriv}(c_m, r_m) \qquad\qquad \mathit{rule}\ [(c_1, r_1), \ldots, (c_m, r_m)]\ (c, r)}{\mathit{deriv}(c, r)}$$

The proof is by induction on the derivation tree for $\mathit{deriv}(c, r)$.

From $\mathit{deriv}(c, r)$, we have $\mathit{deriv}(c_1, r_1)$, ..., $\mathit{deriv}(c_m, r_m)$, and

$$\mathit{rule}\ [(c_1, r_1), \ldots, (c_m, r_m)]\ (c, r) \tag{1}$$

for some m, c_1, ..., c_m, r_1, ..., r_m.

For each $i \in \{1, \ldots, m\}$, we have $\mathit{infer}^{\Phi}(c_i, r_i)$ from $\mathit{deriv}(c_i, r_i)$ and the induction hypothesis. We show that $\mathit{res}^{\Phi}(c_i, r_i)$ holds by distinguishing between the cases where $\Phi(c_i) = R$ and where $\Phi(c_i) \neq R$.

– Suppose $\Phi(c_i) = R$. Then, it holds that $r \in \Phi(c_i)$. Hence, we have $\mathit{res}^{\Phi}(c_i, r_i)$ because of $\mathit{infer}^{\Phi}(c_i, r_i)$, and the definition of res^{Φ}.
– Suppose $\Phi(c_i) \neq R$. From $\mathit{infer}^{\Phi}(c_i, r_i)$, and $\mathit{verif}(\Phi)$, we have $r_i \in \Phi(c_i)$. Hence, we have $\mathit{res}^{\Phi}(c_i, r_i)$ according to the definition of res^{Φ}.

Hence, for each $i \in \{1, \ldots, m\}$, we have $\mathit{res}^{\Phi}(c_i, r_i)$. Thus, we can deduce $\mathit{infer}^{\Phi}(c, r)$ using (1) and the definition of infer^{Φ}. This completes the proof. □

Using this lemma, the soundness theorem can be obtained directly.

Theorem 1 (Soundness). *If $\mathit{verif}(\Phi)$ can be established, then $\mathit{valid}(\Phi)$ holds.*

Proof. Assume $\mathit{verif}(\Phi)$ and $\mathit{deriv}(c, r)$. Then, we have $\mathit{infer}^{\Phi}(c, r)$ according to Lemma 1. Thus, we can deduce $r \in \Phi(c)$ using $\mathit{verif}(\Phi)$. □

The application of this theorem reliably turns the problem of establishing the validity of a specification Φ into the problem of proving $\mathit{verif}(\Phi)$, irrespective of the language used for the program that is specified in Φ. The examples in Sect. 3 and Sect. 4 show that the proof of $\mathit{verif}(\Phi)$ is free from induction for reasoning about iterative and recursive programming constructs, once auxiliary information summarizing the effects of these constructs is provided.

Remark 1. Lemma 1 suggests that an abstract form of computation is obtained leveraging verified user specification. This abstract computation over-approximates the concrete computation, which indicates a potential connection with abstract interpretation [16]. However, we do not attempt at a formal interpretation of our technique in the framework of abstract interpretation in this work.

3 Illustrative Example

In this section, we illustrate our technique using a toy example. In this example, a program computing the factorial of a natural number is written in the While language [27]. We show how the big-step semantics of the While language can be formulated with the *rule* predicate introduced in Sect. 2.2. We then show how the functional correctness of the factorial program can be specified and proven.

3.1 Big-Step Semantics of the While Language

The main syntactical categories of the While language are arithmetic expressions a, Boolean expressions b, and statements S. A statement can be skip that performs no operation, an assignment $x := a$, a sequential composition $S_1; S_2$, a branching statement $\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$, or a loop $\mathsf{while}\ b\ \mathsf{do}\ S$. Let the set of all statements be *Stmt*.

For programs in the While language, the states σ are elements of $\Sigma := Var \to \mathbb{Z}$. Here, *Var* is the set of variables and $\mathbb{Z}$ is the set of integers. The evaluation of arithmetic expressions and Boolean expressions in states can be formalized by defining evaluation functions $\mathcal{A}$ and $\mathcal{B}$, respectively, as in [27].

For the semantics of the While language, the set C of configurations is $Stmt \times \Sigma$, and the set R of result configurations is Σ. We formulate the big-step semantics by defining the predicate *rule*, as in Fig. 1. In each line, a combination of the parameter values for which *rule* holds is given.

$$
\begin{array}{l}
\mathit{rule}\ [\,]\ (\langle \mathsf{skip}, \sigma\rangle, \sigma)\\
\mathit{rule}\ [\,]\ (\langle x := a, \sigma\rangle, \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma])\\
\mathit{rule}\ [(\langle S_1, \sigma\rangle, \sigma''), (\langle S_2, \sigma''\rangle, \sigma')]\ (\langle S_1; S_2, \sigma\rangle, \sigma')\\
\mathit{rule}\ [(\langle S_1, \sigma\rangle, \sigma')]\ (\langle \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma\rangle, \sigma') \quad \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{tt}\\
\mathit{rule}\ [(\langle S_2, \sigma\rangle, \sigma')]\ (\langle \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma\rangle, \sigma') \quad \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{ff}\\
\mathit{rule}\ [(\langle S, \sigma\rangle, \sigma''), (\langle \mathsf{while}\ b\ \mathsf{do}\ S, \sigma''\rangle, \sigma')]\ (\langle \mathsf{while}\ b\ \mathsf{do}\ S, \sigma\rangle, \sigma')\ \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{tt}\\
\mathit{rule}\ [\,]\ (\langle \mathsf{while}\ b\ \mathsf{do}\ S, \sigma\rangle, \sigma)\ \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{ff}
\end{array}
$$

Fig. 1. The semantic rules for the statements of the While language

There is a direct correspondence between the formulation in Fig. 1 and a formulation using inference rules (e.g., [27]). For instance, the inference rule for the loop $\mathsf{while}\ b\ \mathsf{do}\ S$ in the case where the conditional expression evaluates to true can be formulated as

$$\frac{\langle S, \sigma\rangle \to \sigma'' \quad \langle \textsf{while}\ b\ \textsf{do}\ S, \sigma''\rangle \to \sigma'}{\langle \textsf{while}\ b\ \textsf{do}\ S, \sigma\rangle \to \sigma'} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

It is captured exactly by the second last line in the definition of *rule* in Fig. 1.

3.2 Factorial Program and Its Specification

Consider the program S_{fac} in the While language. The program computes the factorial $m!$ where m is the initial value of the program variable $\mathtt{m}$.

$$\begin{aligned} S_{\text{fac}} &:= (\mathtt{fac} := \mathtt{m}; S_{\text{wh}}) \\ S_{\text{wh}} &:= (\textsf{while}\ 1 < \mathtt{m}\ \textsf{do}\ (\mathtt{m} := \mathtt{m} - 1; \mathtt{fac} := \mathtt{fac} * \mathtt{m})) \end{aligned}$$

Let P_m be the set of states where $\mathtt{fac}$ has the value $m!$. Let $P'_{m,fac}$ be the set of states where $\mathtt{fac}$ has the value $fac * (m-1)!$.

$$\begin{aligned} P_m &:= \{\sigma'[\mathtt{fac} \mapsto m!] \mid \sigma' \in \Sigma\} \\ P'_{m,fac} &:= \{\sigma'[\mathtt{fac} \mapsto fac * (m-1)!] \mid \sigma' \in \Sigma\} \end{aligned}$$

We consider the following specification for the program.

$$\begin{aligned} \Phi_{\text{fac}}(\langle S_{\text{fac}}, \sigma\rangle) &:= P_m && \text{if } m = \sigma(\mathtt{m}) \wedge m > 0 \wedge \sigma \in \Sigma \\ \Phi_{\text{fac}}(\langle S_{\text{wh}}, \sigma\rangle) &:= P'_{m,fac} && \text{if } m = \sigma(\mathtt{m}) \wedge m > 0 \wedge fac = \sigma(\mathtt{fac}) \wedge \sigma \in \Sigma \\ \Phi_{\text{fac}}(c) &:= \Sigma && \text{if } c \text{ is not of the above forms} \end{aligned}$$

The specification says that when S_{fac} finishes execution started in a state where the value of $\mathtt{m}$ is $m > 0$, the value of $\mathtt{fac}$ will be $m!$. The specification also contains the auxiliary claim that when the loop S_{wh} finishes execution started in a state where $\mathtt{fac}$ has the value fac and $\mathtt{m}$ has the value $m > 0$, the value of $\mathtt{fac}$ will be equal to the product of fac and $(m-1)!$ (noting that $0! = 1$).

3.3 Proof of the Factorial Program

A direct proof of the factorial program S_{fac} based on the big-step operational semantics of the While language would require an induction on the shape of derivation trees (e.g., [27]) to establish a suitable invariant for the loop S_{wh}.

Using the technique of Sect. 2, we aim at establishing $valid(\Phi_{\text{fac}})$. With Theorem 1, it suffices to show $verif(\Phi_{\text{fac}})$ – for all c and r, assuming $infer^{\Phi_{\text{fac}}}(c, r)$, we attempt to show $r \in \Phi_{\text{fac}}(c)$.

1. Firstly, assume c is $\langle S_{\text{fac}}, \sigma\rangle$, where $\sigma(\mathtt{m}) > 0$. Then, $\Phi_{\text{fac}}(c)$ is P_m, where $m = \sigma(\mathtt{m})$. Using $infer^{\Phi_{\text{fac}}}(\langle S_{\text{fac}}, \sigma\rangle, r)$ and the semantics of the While language in Fig. 1, it is not difficult to obtain

$$rule\ [(\langle \mathtt{fac} := \mathtt{m}, \sigma\rangle, \sigma''), (\langle S_{\text{wh}}, \sigma''\rangle, r)]\ (\langle S_{\text{fac}}, \sigma\rangle, r)$$

for some σ'' such that $res^{\Phi_{\text{fac}}}(\langle \mathtt{fac} := \mathtt{m}, \sigma\rangle, \sigma'')$ and $res^{\Phi_{\text{fac}}}(\langle S_{\text{wh}}, \sigma''\rangle, r)$. Since $\Phi_{\text{fac}}(\langle \mathtt{fac} := \mathtt{m}, \sigma\rangle) = R$, we deduce $infer^{\Phi_{\text{fac}}}(\langle \mathtt{fac} := \mathtt{m}, \sigma\rangle, \sigma'')$ from

$res^{\Phi_{\text{fac}}}(\langle \texttt{fac} := \texttt{m}, \sigma\rangle, \sigma'')$. Hence, we deduce $\sigma'' = \sigma[\texttt{fac} \mapsto \sigma(\texttt{m})]$. Hence, we have $\sigma''(\texttt{m}) = \sigma(\texttt{m}) > 0$. Hence, we have $\Phi_{\text{fac}}(\langle S_{\text{wh}}, \sigma''\rangle) = P'_{\sigma''(\texttt{m}),\sigma''(\texttt{fac})} = \{\sigma'[\texttt{fac} \mapsto \sigma''(\texttt{fac}) * (\sigma''(\texttt{m}) - 1)!] \mid \sigma' \in \Sigma\} = \{\sigma'[\texttt{fac} \mapsto \sigma(\texttt{m})!] \mid \sigma' \in \Sigma\} = P_m$. Moreover, from $res^{\Phi_{\text{fac}}}(\langle S_{\text{wh}}, \sigma''\rangle, r)$ we have $r \in \Phi_{\text{fac}}(\langle S_{\text{wh}}, \sigma''\rangle)$. Ultimately, we have $r \in P_m$.

2. Secondly, assume c is $\langle S_{\text{wh}}, \sigma\rangle$, where $\sigma(\texttt{m}) > 0$. Then, $\Phi_{\text{fac}}(c)$ is $P'_{m,fac}$, where $m = \sigma(\texttt{m})$, and $fac = \sigma(\texttt{fac})$. Using $infer^{\Phi_{\text{fac}}}(\langle S_{\text{wh}}, \sigma\rangle, r)$ and the semantics of the While language in Fig. 1, we have the following two cases.
   (a) We have $m \leq 1$, $rule\ [\,]\ (\langle S_{\text{wh}}, \sigma\rangle, \sigma)$, and $r = \sigma$. Since $m > 0$ and $m \leq 1$, we have $m = 1$. Hence, it is not difficult to deduce $r \in P'_{m,fac}$.
   (b) We have $m > 1$, and

$$rule\ [(\langle \texttt{m} := \texttt{m} - \texttt{1}; \texttt{fac} := \texttt{fac} * \texttt{m}, \sigma\rangle, \sigma''), (\langle S_{\text{wh}}, \sigma''\rangle, r)]\ (\langle S_{\text{wh}}, \sigma\rangle, r)$$

   for some σ'' such that $res^{\Phi_{\text{fac}}}(\langle \texttt{m} := \texttt{m} - \texttt{1}; \texttt{fac} := \texttt{fac} * \texttt{m}, \sigma\rangle, \sigma'')$ and $res^{\Phi_{\text{fac}}}(\langle S_{\text{wh}}, \sigma''\rangle, r)$. From the former we have

$$infer^{\Phi_{\text{fac}}}(\langle \texttt{m} := \texttt{m} - \texttt{1}; \texttt{fac} := \texttt{fac} * \texttt{m}, \sigma\rangle, \sigma'')$$

   The specification Φ_{fac} provides no information about the two assignments, $\texttt{m} := \texttt{m} - \texttt{1}$ and $\texttt{fac} := \texttt{fac} * \texttt{m}$. Hence, $infer^{\Phi_{\text{fac}}}$ applies to these two individual assignments, and it can be deduced that $\sigma'' = \sigma[\texttt{m} \mapsto m - 1, \texttt{fac} \mapsto fac * (m-1)]$. Hence, we have $\sigma''(\texttt{m}) = m - 1 > 0$. Hence, $\Phi_{\text{fac}}(\langle S_{\text{wh}}, \sigma''\rangle) = P'_{\sigma''(\texttt{m}),\sigma''(\texttt{fac})} = \{\sigma'[\texttt{fac} \mapsto (fac * (m-1)) * (m-1-1)!] \mid \sigma' \in \Sigma\} = P'_{m,fac}$. Moreover, from $res^{\Phi_{\text{fac}}}(\langle S_{\text{wh}}, \sigma''\rangle, r)$ we have $r \in P'_{\sigma''(\texttt{m}),\sigma''(\texttt{fac})}$. Ultimately, we have $r \in P'_{m,fac}$.

In the other cases, we have $\Phi_{\text{fac}}(c) = R$. Hence, it trivially holds that $r \in \Phi_{\text{fac}}(c)$ The proof is thus complete. □

The above proof of the factorial program does not require the use of induction. Essentially, the induction required for the loop is already encapsulated in the proof of Theorem 1.

3.4 Comparison with Hoare-Style Program Verification

A Hoare-style specification of the factorial program would be $\{\texttt{m} = n \wedge n > 0\}\ S_{\text{fac}}\ \{\texttt{fac} = n!\}$ Here, n is a logical variable that is used to record the initial value of the program variable m. This specification corresponds to our definition of $\Phi_{\text{fac}}(\langle S_{\text{fac}}, \sigma\rangle)$ for $\sigma(\texttt{m}) > 0$. The latter is more verbose for its explicit reference to states. On the other hand, the use of the latter specification spares the efforts to define an assertion language for each specific programming language.

In Hoare logic, the verification of the program can be performed with the loop invariant $1 \leq \texttt{m} \leq n\ \wedge\ \texttt{fac} = n * (n-1) * \cdots * \texttt{m}$. It captures the condition that is preserved under the effects of a single round of loop. In comparison, the specification Φ_{fac} features the loop variant $\Phi_{\text{fac}}(\langle S_{\text{wh}}, \sigma\rangle)$ (with $\sigma(\texttt{m}) > 0$). It captures the cumulative effects of the loop from the start of any round to the end of the last round. It can be seen that different ways of thinking are required in coming up with the two kinds of specifications. With the proposed technique, the same style as Φ_{fac} can be used for different programming languages, for both loops and recursive functions, as can be seen in Sect. 4.

In Hoare logic, the reasoning about programs is often performed in a backward fashion. For a statement that is neither a loop nor a function call, a pre-condition is derived from the post-condition based on the logical rule for the statement. For a loop or a function call, the pre-condition is inferred based on the invariant of the loop or the contract of the function. In our technique, the reasoning is performed in a forward fashion. If a specification provides no information about a configuration, a result configuration is derived directly using the semantics. For instance, the result configuration $\sigma[\mathtt{fac} \mapsto \sigma(\mathtt{m})]$ is derived from the configuration $\langle \mathtt{fac} := \mathtt{m}, \sigma\rangle$ using the semantics in the factorial example. Otherwise, the specification is used to infer the potential result configurations. For instance, the specification Φ_{fac} is used to infer the potential result configurations for the configuration $\langle S_{\mathrm{wh}}, \sigma[\mathtt{fac} \mapsto \sigma(\mathtt{m})]\rangle$ in the factorial example.

In Hoare-style program verification, a loop invariant is justified by assuming that it holds after a round of loop, and showing that it also holds before that round. In our technique, a loop variant is justified by executing one round of loop from each configuration satisfying the pre-condition of the loop variant, and showing that no more result configurations are possible according to the loop variant for each configuration reached after that round (e.g., $\langle S_{\mathrm{wh}}, \sigma[\mathtt{m} \mapsto m-1, \mathtt{fac} \mapsto \mathit{fac} * (m-1)]\rangle$ in the factorial example), than for the original configuration (e.g., $\langle S_{\mathrm{wh}}, \sigma\rangle$ in the factorial example) before that round.

4 Verification of Iterative and Recursive Programs

In this section, we evaluate our technique with two further examples. In the two examples, programming languages of the imperative and functional paradigms are used, respectively, to implement the functionality of merging two sorted lists of integers into a single sorted list of integers.

4.1 Extended While Language and Array-Merging Program

Extended While Language. The programming language of this section is an extension of the While language. This extension contains the extra features of one-dimensional arrays and functions.

We give the syntax for arithmetic expressions a, Boolean expressions b, and statements S. We then explain the constructs present in the extension only.

$$\begin{aligned}
a &::= n \mid x \mid X \mid X[a] \mid a + a \mid a - a \mid a * a \mid a \,/\, a\\
b &::= \mathsf{true} \mid \mathsf{false} \mid a = a \mid a < a \mid b \,\&\&\, b \mid !b\\
S &::= \mathsf{var}\ x \mid \mathsf{arr}\ X[n] \mid x := a \mid X[a] := a \mid \mathsf{skip} \mid\\
&\quad\ \ \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \mathsf{while}\ b\ \mathsf{do}\ S \mid S; S \mid f(a, \ldots, a) \rightarrow [x, \ldots, x]
\end{aligned}$$

$$\rho_{\mathrm{mg}} := [\,\mathsf{merge} \mapsto ([\mathsf{S}, \mathsf{T}, \mathtt{i}, \mathtt{m}, \mathtt{n}], [\,], S_{\mathrm{mg}})\,]$$

$$\begin{aligned}
S_{\mathrm{mg}} &:= \mathsf{var}\ \mathtt{j}; \mathsf{var}\ \mathtt{k}; \mathtt{j} := \mathtt{m} + 1; \mathtt{k} := \mathtt{i}; S_{\mathrm{wh}}; S_{\mathtt{i},\mathtt{m}}; S_{\mathtt{j},\mathtt{n}}\\
S_{\mathrm{wh}} &:= \mathsf{while}\ \mathtt{i} \leq \mathtt{m} \,\&\&\, \mathtt{j} \leq \mathtt{n}\ \mathsf{do}\ (\\
&\qquad (\mathsf{if}\ \mathsf{S}[\mathtt{i}] \leq \mathsf{S}[\mathtt{j}]\ \mathsf{then}\ \mathsf{T}[\mathtt{k}] := \mathsf{S}[\mathtt{i}]; \mathsf{i} := \mathsf{i} + 1\ \mathsf{else}\ \mathsf{T}[\mathtt{k}] := \mathsf{S}[\mathtt{j}]; \mathsf{j} := \mathsf{j} + 1);\\
&\qquad \mathtt{k} := \mathtt{k} + 1\,)\\
S_{\mathtt{i},\mathtt{m}} &:= \mathsf{while}\ \mathtt{i} \leq \mathtt{m}\ \mathsf{do}\ (\mathsf{T}[\mathtt{k}] := \mathsf{S}[\mathtt{i}]; \mathsf{i} := \mathsf{i} + 1; \mathsf{k} := \mathsf{k} + 1)\\
S_{\mathtt{j},\mathtt{n}} &:= \mathsf{while}\ \mathtt{j} \leq \mathtt{n}\ \mathsf{do}\ (\mathsf{T}[\mathtt{k}] := \mathsf{S}[\mathtt{j}]; \mathsf{j} := \mathsf{j} + 1; \mathsf{k} := \mathsf{k} + 1)
\end{aligned}$$

Fig. 2. The program ρ_{mg} that merges sorted array fragments

Here, X is an array identifier, and $X[a]$ is the expression used to retrieve the element of the array X at the index a. In addition, $\mathsf{var}\ x$ is the declaration of the variable x, $\mathsf{arr}\ X[n]$ is the declaration of the array with identifier X and size n, $X[a_1] := a_2$ is an assignment of the result of a_2 to the element of the array X indexed at a_1, and $f(a_1, \ldots, a_m) \to [x_1, \ldots, x_n]$ is a call to the function with identifier f with arguments a_1, ..., a_m and return variables x_1, ..., x_n. If some argument a_i is an array, then it is passed by reference in the call. We denote the set of all statements of the extended While language by $\mathit{Stmt}_{\mathrm{ewh}}$.

A *program* in the extended While language is a mapping ρ from each function identifier f to a triple $([w_1, \ldots, w_m], [x_1, \ldots, x_n], S)$ or $\bot$. Here, each w_i $(i \in \{1, \ldots, m\})$ is a parameter of the function that is either a variable x or an array X. Each x_i $(i \in \{1, \ldots, n\})$ is a return variable of the function. The S is the statement of the function. If $\rho(f) = \bot$, then there is no function defined for the function identifier in the program.

For programs of the extended While language, a *state* σ is a pair (s, ι). Here, $s \in (\mathit{Var} \cup \mathit{Arr} \to \mathbb{Z}_\bot) \cup (\mathbb{Z} \to \mathbb{Z})$ is a *store* that maps each variable to an optional integer that is the value of the variable, maps each array name to an optional integer representing the starting location of the array, and maps each location to an integer that is the value stored at the location. In addition, $\iota \in \mathbb{Z}$ is the *next fresh location* that can be used as the starting location of an array. For $\sigma = (s, \iota)$, we write $\sigma(a)$ for $s(a)$. We denote the set of all states by Σ_{ewh}.

For the extended While language, the set C of configurations is $\mathit{Stmt}_{\mathrm{ewh}} \times \Sigma_{\mathrm{ewh}}$, and the set R of result configurations is Σ_{ewh}. For space reasons, we omit the definition of the *rule* predicate that captures the big-step semantics of the extended While language. This definition can be found in the extended version of this paper, as well as the formalization in the Coq proof assistant.

Array-Merging Program and Its Verification. The program ρ_{mg} as shown in Fig. 2 merges the elements in two sorted fragments of an array S into one sorted fragment in a different array T.

The only function in this program is merge. Formally, this function is the triple $([\mathsf{S}, \mathsf{T}, \mathtt{i}, \mathtt{m}, \mathtt{n}], [\,], S_{\mathrm{mg}})$. The parameters $\mathtt{i}$ and $\mathtt{m}$ represent the initial and final index, respectively, for the first fragment of the array S participating in the merger. The second fragment participating in the merger is from the index represented by $\mathtt{m} + 1$ to the index represented by $\mathtt{n}$ in the same array S. The target array fragment of the merger is from the index represented by $\mathtt{i}$ to the index represented by $\mathtt{n}$, in the array T.

For the specification of the program, we use a few pieces of auxiliary notation. We write X_l^h for a triple (X, l, h) that represents the fragment of the array X from the index l to the index h. We write $(\!| X_l^h |\!)_\sigma$ for the list $[\sigma(\ell + l), \ldots, \sigma(\ell + h)]$ where $\ell = \sigma(X)$, i.e., the list of elements of the array X from the index l to the index h. We write $\mathit{occ}\,[z_1, \ldots, z_n]$ for the function h mapping each integer z to the number of occurrences of z in the list $[z_1, \ldots, z_n]$ of integers. For two such functions h_1 and h_2, we write $h_1 \oplus h_2$ for the function $\lambda z. h_1(z) + h_2(z)$. We write $\mathit{sorted}\,[z_1, \ldots, z_n]$ to express that the list $[z_1, \ldots, z_n]$ of integers is sorted in ascending order. We write $\mathit{sep}(X_{l_1}^{h_1}, Y_{l_2}^{h_2}, \sigma)$ to express that the elements of the array X from the index l_1 to the index h_1 occupy a separate memory area from that occupied by the elements of the array Y from the index l_2 to the index h_2, in the state σ. In addition, we write $[u_1, \ldots, u_n]_\sigma^{\sigma'}$ to express for each $i \in \{1, \ldots, n\}$, the value of each u_i is the same in the states σ and σ'. Here, u_i can be a variable x or an array fragment X_l^h. In the latter case, that the value of X_l^h is the same in the two states means $\forall i : l \leq i \leq h \Rightarrow \sigma(\sigma(X) + i) = \sigma'(\sigma'(X) + i)$.

For the program ρ_{mg}, we devise the specification $\varPhi_{\mathrm{mga}}$. We denote the starting index for the first source array fragment in S as well as for the target array fragment in T by l. We use l as a global parameter in the specification.

We specify the function merge as

$$\begin{aligned}
&\varPhi_{\mathrm{mga}}(\langle \mathsf{merge}(X, Y, a_{\mathrm{l}}, a_{\mathrm{m}}, a_{\mathrm{h}}) \to [], \sigma \rangle_{\rho_{\mathrm{ms}}}) := \\
&\quad \{\sigma' \mid \mathit{occ}\,(\!| X_l^h |\!)_\sigma = \mathit{occ}\,(\!| Y_l^h |\!)_{\sigma'} \wedge \mathit{sorted}\,(\!| Y_l^h |\!)_{\sigma'} \} \\
&\quad \text{if } \mathcal{A}[\![a_{\mathrm{l}}]\!]\sigma = l \wedge 0 \leq l \leq m < h \wedge \mathit{sorted}\,(\!| X_l^m |\!)_\sigma \wedge \mathit{sorted}\,(\!| X_{m+1}^h |\!)_\sigma \wedge \mathit{sep}(X_l^h, Y_l^h, \sigma) \\
&\quad \text{where } m = \mathcal{A}[\![a_{\mathrm{m}}]\!]\sigma \wedge h = \mathcal{A}[\![a_{\mathrm{h}}]\!]\sigma
\end{aligned}$$

This specification says that if we call the function merge with two array identifiers X and Y, and expressions a_{l}, a_{m}, a_{h} that evaluate to l, m and h, such that

- $0 \leq l \leq m < h$ holds,
- the array fragments X_l^m and X_{m+1}^h are sorted in the pre-state,
- the array fragments X_l^m and X_{m+1}^h are separated in the pre-state,

then the number of occurrences of each integer in the target array fragment Y_l^h in the post-state is the same as its number of occurrences in the source array fragment X_l^h in the pre-state, and the target array fragment Y_l^h is sorted in ascending order in the post-state.

The core part of the function merge is the loop statement S_{wh} (see Fig. 2). We specify this loop as

$$
\begin{aligned}
&\Phi_{\text{mga}}(\langle S_{\text{wh}}, \sigma\rangle_{\rho_{\text{ms}}}) := \\
&\quad \{\sigma' \mid (i \le \sigma'(\mathtt{i}) = m+1 \wedge j \le \sigma'(\mathtt{j}) \le n \vee j \le \sigma'(\mathtt{j}) = n+1 \wedge i \le \sigma'(\mathtt{i}) \le m) \wedge \\
&\qquad \sigma'(\mathtt{k}) = k + \sigma'(\mathtt{i}) - i + \sigma'(\mathtt{j}) - j \ \wedge\ [\mathtt{m}, \mathtt{n}, \mathsf{S}, \mathsf{T}, \mathsf{S}_l^n, \mathsf{T}_l^{k-1}]_\sigma^{\sigma'} \wedge \\
&\qquad \mathit{occ}\,(\!|\mathsf{S}_i^{\sigma'(\mathtt{i})-1}|\!)_\sigma \oplus \mathit{occ}\,(\!|\mathsf{S}_j^{\sigma'(\mathtt{j})-1}|\!)_\sigma = \mathit{occ}\,(\!|\mathsf{T}_k^{\sigma'(\mathtt{k})-1}|\!)_{\sigma'} \ \wedge\ \mathit{sorted}\,(\!|\mathsf{T}_l^{\sigma'(\mathtt{k})-1}|\!)_{\sigma'} \wedge \\
&\qquad (\sigma'(\mathtt{i}) \le m \wedge \sigma'(\mathtt{k}) \ge l+1 \Rightarrow \mathcal{A}[\![\mathsf{S}[\mathtt{i}]]\!]\sigma' \ge \mathcal{A}[\![\mathsf{T}[\mathtt{k}-1]]\!]\sigma') \wedge \\
&\qquad (\sigma'(\mathtt{j}) \le n \wedge \sigma'(\mathtt{k}) \ge l+1 \Rightarrow \mathcal{A}[\![\mathsf{S}[\mathtt{j}]]\!]\sigma' \ge \mathcal{A}[\![\mathsf{T}[\mathtt{k}-1]]\!]\sigma')\,\} \\
&\quad \text{if } 0 \le l \le i \le m < j \le n \ \wedge\ k = i + j - m - 1 \wedge \\
&\qquad (k \ge l+1 \ \Rightarrow\ \mathcal{A}[\![\mathsf{S}[\mathtt{i}]]\!]\sigma \ge \mathcal{A}[\![\mathsf{T}[\mathtt{k}-1]]\!]\sigma \wedge \mathcal{A}[\![\mathsf{S}[\mathtt{j}]]\!]\sigma \ge \mathcal{A}[\![\mathsf{T}[\mathtt{k}-1]]\!]\sigma) \wedge \\
&\qquad \mathit{sorted}\,(\!|\mathsf{S}_i^m|\!)_\sigma \ \wedge\ \mathit{sorted}\,(\!|\mathsf{S}_j^n|\!)_\sigma \ \wedge\ \mathit{sorted}\,(\!|\mathsf{T}_l^{k-1}|\!)_\sigma \ \wedge\ \mathit{sep}(\mathsf{S}_l^n, \mathsf{T}_l^n, \sigma) \\
&\quad \text{where } i = \sigma(\mathtt{i}) \ \wedge\ j = \sigma(\mathtt{j}) \ \wedge\ k = \sigma(\mathtt{k}) \ \wedge\ m = \sigma(\mathtt{m}) \ \wedge\ n = \sigma(\mathtt{n})
\end{aligned}
$$

In the specification, we are concerned with pre-states in which either the overall loop is yet to be executed, or some rounds of the loop have been completed and some further rounds are to be executed. We constrain these pre-states with a few further conditions. One of these conditions states that the elements with indexes `i` and `j` that are to be compared in the next round are both greater than or equal to the last element that has been set in the target array fragment. For each pre-state that satisfies all the conditions in the "if" part, several conditions are asserted for the potential post-state σ'. A key condition here says that the two fragments $\mathsf{S}_i^{\sigma'(\mathtt{i})-1}$ and $\mathsf{S}_j^{\sigma'(\mathtt{j})-1}$ in the source array that are scanned between the reaching of the pre-state and the post-state agree with the fragment $\mathsf{T}_k^{\sigma'(\mathtt{k})-1}$ that is filled between the reaching of the pre-state and the post-state. Another key condition says that the fragment $\mathsf{T}_l^{\sigma'(\mathtt{k})-1}$ of the target array that is already filled in the post-state for the loop is sorted in ascending order.

Without specification inference, the two remaining loops in the array-merging program also need to be explicitly specified. The specification of these two loops is much less involved than that for the first loop, and it is omitted here. With the technique of Sect. 2, the validity of Φ_{mga} can be established.

Theorem 2. *It holds that valid*(Φ_{mga}).

With the help of Theorem 1, the proof requires no induction for reasoning about the loops. This proof boils down to symbolic execution with the help of a series of auxiliary lemmas about the memory layout.

Remark 2. The global parameter l in the specification Φ_{mga} relates the auxiliary information about calls to merge and about the loops in this function. The role of l can be compared to that of a logical variable in a concrete program logic. Such global parameters are captured in the Coq formalization by an explicit argument in the specifications. The type of this argument can be instantiated according to the needs in verifying each specific program. The verification of a program is required to go through for all possible values of this argument.

4.2 Eager Functional Language and List-Merging Program

Eager Functional Language. The language considered in this section is a fragment of the eager functional language as discussed in [33].

$$
\begin{array}{rl}
e ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \\
& e + e \mid e - e \mid e * e \mid e/e \mid \\
& e = e \mid e < e \mid \neg e \mid e \wedge e \mid \\
& \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\
& \mathsf{nil} \mid e :: e \mid \mathsf{listcase}\ e\ \mathsf{of}\ (e, e) \mid \\
& x \mid e\, e \mid \lambda x.e \mid \mathsf{letrec}\ x = \lambda x'.e\ \mathsf{in}\ e
\end{array}
\qquad
\begin{array}{rl}
cf & ::= icf \mid bcf \mid fcf \mid lcf \\
icf & ::= \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
bcf & ::= \mathsf{true} \mid \mathsf{false} \\
fcf & ::= \lambda x.e \\
lcf & ::= \mathsf{nil} \mid cf :: cf
\end{array}
$$

Fig. 3. The expressions and canonical forms of the eager functional language

A program of the eager functional language is an expression. The syntax for expressions is given in the left part of Fig. 3. Here, n is a numeral, x is a variable, $e\, e'$ is an application, $\lambda x.e$ is a lambda abstraction, nil is the empty list, and $e_1 :: e_2$ is the list obtained by prefixing the list e_2 with the element e_1. The expression $\mathsf{listcase}\ e\ \mathsf{of}\ (e', e'')$ branches to e' or e'' depending on whether the result of e is the empty list nil. The expression $\mathsf{letrec}\ x = \lambda x'.e'\ \mathsf{in}\ e$ binds x to $\lambda x'.e'$ in e. This expression allows x to be used in e', thereby allowing recursion. We denote the set of all expressions by *Expr*.

The evaluation of expressions results in *canonical forms* cf as given in the right part of Fig. 3. A canonical form cf can be a canonical form icf for integers, a canonical form bcf for Boolean values, a canonical form fcf for functions, or a canonical form lcf for lists. We denote the set of all canonical forms by Cf.

For the eager functional language, the set C of configurations is *Expr*, and the set R of result configurations is Cf. For space reasons, we omit the definition of the *rule* predicate that captures the big-step semantics of the eager functional language. This definition can be found in the extended version of this paper, as well as the formalization in the Coq proof assistant.

List-Merging Program and Its Verification. The program $e_{\mathrm{mg}}(lcf_1, lcf_2)$ below merges two sorted lists, lcf_1 and lcf_2, into a single sorted list. More concretely, the variable $\mathtt{merge}$ is bound to the expression $\lambda \mathtt{x}.\lambda \mathtt{x}'.e_{\mathrm{lcase}}$ that destructs the lists that are bound to $\mathtt{x}$ and $\mathtt{x}'$, respectively. In case one of the lists is empty, the result of the merger is the other list. Otherwise, the result of the merger is obtained by prefixing the smaller head element of the two given lists over the merging result of the remaining parts of the lists.

$$
\begin{aligned}
e_{\mathrm{mg}}(lcf_1, lcf_2) &:= \mathsf{letrec}\ \mathtt{merge} = (\lambda \mathtt{x}.\lambda \mathtt{x}'.e_{\mathrm{lcase}})\ \mathsf{in}\ \mathtt{merge}\ lcf_1\ lcf_2 \\
e_{\mathrm{lcase}} &:= \mathsf{listcase}\ \mathtt{x}\ \mathsf{of}\ (\mathtt{x}', \lambda \mathtt{i}.\lambda \mathtt{r}.\mathsf{listcase}\ \mathtt{x}'\ \mathsf{of}\ (\mathtt{x}, \lambda \mathtt{i}'.\lambda \mathtt{r}'.e_{\mathrm{if}})) \\
e_{\mathrm{if}} &:= \mathsf{if}\ \mathtt{i} \leq \mathtt{i}'\ \mathsf{then}\ \mathtt{i} :: \mathtt{merge}\, \mathtt{r}\, \mathtt{x}'\ \mathsf{else}\ \mathtt{i}' :: \mathtt{merge}\, \mathtt{x}\, \mathtt{r}'
\end{aligned}
$$

To develop a specification for the list-merging program, we define a piece of auxiliary notation. We write $\langle\!|\mathit{lcf}|\!\rangle$ for the mathematical list of integers represented by the canonical form lcf for lists. Formally, we define $\langle\!|\mathsf{nil}|\!\rangle:=[\,]$, $\langle\!|\mathit{icf} :: \mathit{lcf}|\!\rangle := \mathit{icf} :: \mathit{zs}$ if $\mathit{zs} = \langle\!|\mathit{lcf}|\!\rangle \wedge \mathit{zs} \in \mathbb{Z}^*$, and $\langle\!|\mathit{lcf}|\!\rangle:=\bot$ otherwise.

We devise the a specification for the list-merging program, Φ_{mgl}. Using the function *occ* and the predicate *sorted* introduced in Sect. 4.1, we specify the expression $e_{\mathrm{mg}}(\mathit{lcf}_1, \mathit{lcf}_2)$ as

$$\begin{aligned}&\Phi_{\mathrm{mgl}}(e_{\mathrm{mg}}(\mathit{lcf}_1, \mathit{lcf}_2)) :=\\ &\qquad \{\mathit{lcf} \mid \exists \mathit{zs} \in \mathbb{Z}^* \colon \mathit{zs} = \langle\!|\mathit{lcf}|\!\rangle \wedge \mathit{occ}\,\mathit{zs} = \mathit{occ}\,\mathit{zs}_1 \oplus \mathit{occ}\,\mathit{zs}_2 \wedge \mathit{sorted}\,\mathit{zs}\}\\ &\text{if } \mathit{zs}_1 \in \mathbb{Z}^* \wedge \mathit{zs}_2 \in \mathbb{Z}^* \wedge \mathit{sorted}\,\mathit{zs}_1 \wedge \mathit{sorted}\,\mathit{zs}_2\\ &\text{where } \mathit{zs}_1 = \langle\!|\mathit{lcf}_1|\!\rangle \wedge \mathit{zs}_2 = \langle\!|\mathit{lcf}_2|\!\rangle\end{aligned}$$

This specification says that given list canonical forms lcf_1 and lcf_2 that are both sorted in ascending order, the result of executing $e_{\mathrm{mg}}(\mathit{lcf}_1, \mathit{lcf}_2)$ is a list canonical form lcf. The list canonical form lcf contains the elements as contained in either lcf_1 or lcf_2. Furthermore, the list canonical form lcf is sorted in ascending order.

To support the verification of the specification for $e_{\mathrm{mg}}(\mathit{lcf}_1, \mathit{lcf}_2)$, we specify an unfolded form of this expression. The execution of this unfolded form either terminates directly, or gives the same form again.

$$\begin{aligned}&\Phi_{\mathrm{mgl}}(\,(\lambda\mathsf{x}.\mathsf{letrec\ merge} = \lambda\mathsf{x}.\lambda\mathsf{x}'.e_{\mathrm{lcase}}\ \mathsf{in}\ \lambda\mathsf{x}'.e_{\mathrm{lcase}})\ \mathit{lcf}_1\ \mathit{lcf}_2\,) :=\\ &\qquad \{\mathit{lcf} \mid \exists \mathit{zs} \in \mathbb{Z}^* \colon \mathit{zs} = \langle\!|\mathit{lcf}|\!\rangle \wedge \mathit{occ}\,\mathit{zs} = \mathit{occ}\,\mathit{lcf}_1 \oplus \mathit{occ}\,\mathit{lcf}_2 \wedge \mathit{sorted}\,\mathit{zs}\}\\ &\text{if } \mathit{zs}_1 \in \mathbb{Z}^* \wedge \mathit{zs}_2 \in \mathbb{Z}^* \wedge \mathit{sorted}\,\mathit{zs}_1 \wedge \mathit{sorted}\,\mathit{zs}_2\\ &\text{where } \mathit{zs}_1 = \langle\!|\mathit{lcf}_1|\!\rangle \wedge \mathit{zs}_2 = \langle\!|\mathit{lcf}_2|\!\rangle\end{aligned}$$

This specification reflects that the unfolded expression $(\lambda\mathsf{x}.\mathsf{letrec\ merge} = \lambda\mathsf{x}.\lambda\mathsf{x}'.\ e_{\mathrm{lcase}}\ \mathsf{in}\ \lambda\mathsf{x}'.e_{\mathrm{lcase}})\ \mathit{lcf}_1\ \mathit{lcf}_2$ delivers analogous guarantees to those delivered by the original expression $e_{\mathrm{mg}}(\mathit{lcf}_1, \mathit{lcf}_2)$.

With the technique of Sect. 2, the validity of Φ_{mgl} can be established.

Theorem 3. *It holds that valid*(Φ_{mgl}).

With the help of Theorem 1, the proof requires no induction for reasoning about the recursive applications of the function bound to merge. This proof boils down to symbolic execution with the help of a few auxiliary lemmas about substitution and evaluation related to canonical forms.

Remark 3. It might appear that the auxiliary information needed for the verification of the list-merging program should be for expressions of the form merge _ _. However, these expressions cannot be evaluated, because information about the actual function bound to merge is missing. The form that recurs in the evaluation of $e_{\mathrm{mg}}(\mathit{lcf}_1, \mathit{lcf}_2)$ is actually $(\lambda\mathsf{x}.\mathsf{letrec\ merge} = \lambda\mathsf{x}.\lambda\mathsf{x}'.e_{\mathrm{lcase}}\ \mathsf{in}\ \lambda\mathsf{x}'.e_{\mathrm{lcase}})$ _ _.

5 On Completeness of the Technique

It is untrue that any valid specification can be verified. Intuitively, a specification Φ that is valid but missing the necessary auxiliary information such as loop variants might not be verifiable.

Consider the factorial example in Sect. 3, and the specification Φ'_{fac} that is the same as Φ_{fac} except that Φ'_{fac} maps $\langle S_{\mathrm{wh}}, \sigma\rangle$ where $\sigma(\mathtt{m}) > 0$ to Σ. The specification Φ'_{fac} is valid as the specification Φ_{fac} is. This is because Φ'_{fac} is a loosened version of Φ_{fac}. However, Φ'_{fac} cannot be verified using our proposed technique. Due to missing auxiliary information, the verification procedure leads to a non-terminating symbolic execution of the factorial program.

In the following, we show that for a given specification that is valid, there is always a more informative specification Φ' than Φ that is verifiable. Formally, a specification Φ' is *at least as informative as* a specification Φ, as denoted by $\Phi \preceq \Phi'$, if for each configuration c, it holds that $\Phi(c) \supseteq \Phi'(c)$.

The lemma below says the specification mapping each configuration to the set of all the semantically derivable result configurations can be verified.

Lemma 2. *Let $\Phi_\star$ be the specification satisfying $\Phi_\star(c) = \{r \mid \mathit{deriv}(c, r)\}$ for all configurations c. Then, $\mathit{verif}(\Phi_\star)$ can be established.*

Proof. We show that for all c and r, if $\mathit{infer}^{\Phi_\star}(c, r)$, then $r \in \Phi_\star(c)$. This boils down to showing if $\mathit{infer}^{\Phi_\star}(c, r)$, then $\mathit{deriv}(c, r)$. Below, we give an inductive proof of this statement.

Assume $\mathit{infer}^{\Phi_\star}(c, r)$. Then, there exist some m, $c_1, \ldots, c_m$, $r_1, \ldots, r_m$, such that $\mathit{res}^{\Phi_\star}(c_1, r_1), \ldots, \mathit{res}^{\Phi_\star}(c_m, r_m)$, and

$$\mathit{rule}\ [(c_1, r_1), \ldots, (c_m, r_m)]\ (c, r) \tag{2}$$

For each i, we show that $\mathit{deriv}(c_i, r_i)$ holds by distinguishing between the cases where $\Phi_\star(c_i) = R$ and $\Phi_\star(c_i) \neq R$.

- Suppose $\Phi_\star(c_i) = R$. Then we deduce $\mathit{infer}^{\Phi_\star}(c_i, r_i)$ from $\mathit{res}^{\Phi_\star}(c_i, r_i)$. Hence, we have $\mathit{deriv}(c_i, r_i)$ from the induction hypothesis.
- Suppose $\Phi_\star(c_i) \neq R$. We have $r_i \in \Phi_\star(c_i)$ using $\mathit{res}^{\Phi_\star}(c_i, r_i)$. Hence, we have $\mathit{deriv}(c_i, r_i)$ using the definition of $\Phi_\star$.

Ultimately, we have $\mathit{deriv}(c_i, r_i)$ for each $i \in \{1, \ldots, m\}$, and we obtain $\mathit{deriv}(c, r)$ using (2). This completes the proof. □

The following theorem says that for each valid specification Φ, there is a specification that is at least as informative as Φ, and that can be verified.

Theorem 4 (Relative Completeness). *For each valid specification Φ, there exists a specification Φ' such that $\Phi \preceq \Phi'$, and $\mathit{verif}(\Phi')$ can be established.*

Proof. We first show that the specification $\Phi_\star$ in Lemma 2 is at least as informative as any valid specification. Let Φ be a specification satisfying $\mathit{valid}(\Phi)$.

Let c be an arbitrary configuration. Let r be any result configuration satisfying $r \in \Phi_\star(c)$. We have $deriv(c, r)$ from the definition of $\Phi_\star$. Hence, we have $r \in \Phi(c)$ because of $valid(\Phi)$. Hence, $\Phi_\star(c) \subseteq \Phi(c)$ holds. Hence, we have $\Phi \preceq \Phi_\star$. Moreover, we have $verif(\Phi_\star)$ using Lemma 2. This completes the proof. □

If the program contained in a configuration exhibits only bounded behavior, then the corresponding result configuration can be obtained through symbolic execution. Hence, it is not necessary that a verifiable specification should cover these configurations. In an informal sense, this argument supports that for a specification to be verified, it is only necessary to provide auxiliary information about constructs such as loops and recursive function calls in the specification.

6 Related Work

Inductive invariants [24] are well-studied means to sound program verification directly based on operational execution models. An inductive invariant needs to be preserved by all the possible atomic steps that can be taken in the execution of the target program. This requirement often leads to difficulties in identifying the exact condition that qualifies as an inductive invariant, and that enables the verification of the target program.

In [26], a method is proposed to generate inductive invariants from inductive assertions. The method is based on a small-step execution relation. Minimal information about the syntactical structure of the programming language is required in the generation of the inductive invariants. In comparison, our technique targets big-step operational semantics, and its soundness does not rely on the reduction of the verification problem to the generation of inductive invariants.

In [25], a technique is proposed to generate sound program verifiers based on existing formalizations of small-step semantics in proof assistants. The soundness of the technique is established with a coinductive argument. In comparison, our technique targets big-step operational semantics, and is based on inductive reasoning. Nevertheless, we are inspired by this work in the style of language-independent program specifications and the form of completeness statements.

In [40], a language-independent verification technique based on reachability logics and semantics formulated in rewriting systems is introduced. In comparison, our technique can only be used for big-step semantics. However, our technique can be used with semantic definitions using inductive predicates in a proof assistant, and requires only the logical foundation of the proof assistant to function. Our technique also has a succinct, inductive argument for soundness.

Several developments provide means to systematically derive abstract semantics from concrete semantics such as big-step operational semantics and its variants [12,13,35]. Among these, [12] proposes a language-independent notion of skeletal semantics that can be instantiated to obtain concrete and abstract semantic interpretations. However, the emphasis of these developments is in obtaining automated static analyses of programs, rather than in exploiting user-provided specification in the deductive verification of deep correctness properties.

To some extent, language-independent program verification can also be supported by encoding the target languages or target programs in the same language (e.g., WhyML, Boogie, etc.) or calculus (e.g., CSP, the π-calculus, etc.) supporting verification. This encoding can be considerably more light-weight than the direct formalization of the syntax and semantics of the source language. However, when the features of the source language are sufficiently complicated, it can be highly non-trivial to justify the encoding.

In Unifying Theories of Programming [14,19,22,29,32,38], the semantics of programming constructs (e.g., assignment, conditional, sequential composition, parallel composition, etc.) involved in diverse languages is formulated in a relational calculus. The connection between different kinds of semantics – algebraic semantics, denotational semantics, and operational semantics – is investigated. In comparison, we study the verification of programs based on a common model of big-step opperational semantics. We do not look at concrete programming constructs, or investigate the connection between different types of semantics.

7 Future Directions

Reuse of Existing Formalization of Semantics. For the related language-independent verification techniques based on small-step operational semantics [25,26], it is not difficult to obtain a verification infrastructure by directly reusing an existing formalization of semantics. This is because a small-step semantics readily provides a step relation that can be used to interface with the verification framework. In comparison, we have only shown that our language-independent verification technique can be applied after the big-step semantics of the target language is formalized via a predicate that explicitly captures the premises and conclusions of the semantic rules. Although the big-step semantics formulated using this predicate closely resemble their classical formulation, it is desirable if a higher level of reusability can be enabled. A potential solution is to construct a program that automatically transforms a formalization of big-step semantics into a formulation with the *rule* predicate. Such transformation could be attempted using the MetaCoq framework [39] to achieve seamless integration with the Coq proof assistant.

Integration of Techniques for Other Aspects of Deductive Verification. The purpose of the present work is not to simplify the overall task of deductive program verification beyond achievable by existing techniques. Instead, the focus has been the ability to reason about different types of programming constructs that potentially cause unbounded behavior, in a uniform way. This ability helps simplify the reasoning about these programming constructs, relative to direct inductive proofs based on big-step operational semantics. To construct a full-fledged language-independent program verifier in a proof assistant, effective treatment of other aspects of deductive program verification (e.g., memory layout, mathematical reasoning in diverse problem domains, etc.) is required. In principle, it is desirable to deal with the language-generic and language specific aspects of program verification separately (as advocated in UTP [19]). Concretely, existing formalization of program logics and mathematical theories in proof assistants are expected to provide the essential technical ingredients for simplifying the remaining aspects of verification tasks.

8 Conclusion

To tackle the problem caused by the proliferation of programming languages in deductive program verification, we provide a language-independent verification technique that addresses the cross-cutting concern of reasoning about programming constructs potentially causing unbounded behavior. Typically, these constructs include loops and recursive functions in different forms. The proposed technique can be applied to any programming language with a big-step operational semantics. The user of this technique need not set up inductions for the loops and recursive calls in performing a program proof, but performs symbolic execution of the program based on the big-step semantics, and with the help of a specification containing auxiliary information about these constructs. The technique admits succinct, inductive arguments for soundness and relative completeness that are verified in the Coq proof assistant along with other formal claims [8]. It has been illustrated with verification examples targeting languages of different paradigms. It provides a basis for a language-independent program verifier based on big-step operational semantics in proof assistants.

Acknowledgments. This work was supported by the National Natural Science Foundation of China (61876111, 62002246).

References

1. The Coq proof assistant. https://coq.inria.fr/
2. Michelson - the language of Tezos. https://www.michelson.org/
3. The move language. https://developers.libra-china.org/docs/crates/move-language/index.html
4. A sequential imperative programming language - syntax, semantics, Hoare logics and verification environment. https://www.isa-afp.org/entries/Simpl.html
5. Solidity. https://docs.soliditylang.org/en/v0.8.0/
6. VCC: A verifier for concurrent C. https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/
7. Yul. https://docs.soliditylang.org/en/v0.8.0/yul.html
8. Formalization of the verification technique in Coq (2021). https://github.com/lixm/ind-verify/tree/master
9. Ahrendt, W., Beckert, B., Bubel, R. (eds.): Deductive Software Verification - The KeY Book. From Theory to Practice. Lecture Notes in Computer Science, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
10. Appel, A.W.: Verified Software Toolchain - (invited talk). In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_1

11. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reason. **43**(3), 263–288 (2009)
12. Bodin, M., Gardner, P., Jensen, T.P., Schmitt, A.: Skeletal semantics and their interpretations. Proc. ACM Program. Lang. **3**(POPL), 44:1–44:31 (2019)
13. Bodin, M., Jensen, T.P., Schmitt, A.: Certified abstract interpretation with pretty-big-step semantics. In: Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP), pp. 29–40 (2015)
14. Cavalcanti, A., Wellings, A., Woodcock, J.: The safety-critical Java memory model: a formal account. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 246–261. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_20
15. Clément, D., Despeyroux, J., Despeyroux, T., Kahn, G.: A simple applicative language: mini-ML. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP), pp. 13–27 (1986)
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth ACM Symposium on Principles of Programming Languages (POPL), pp. 238–252 (1977)
17. Hirai, Y., et al.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M. (ed.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
18. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
19. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Pearson College Div (1998)
20. Jung, R., Krebbers, R., Jourdan, J., et al.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018)
21. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0039592
22. Ke, W., Li, X., Liu, Z., Stolz, V.: rCOS: a formal model-driven engineering method for component-based software. Front. Comput. Sci. China **6**(1), 17–39 (2012)
23. Klein, G., Nipkow, T.: Jinja is not Java. Arch. Formal Proofs (2005)
24. McCarthy, J.: Towards a mathematical science of computation. In: Proceedings of the 2nd IFIP Congress on Information Processing, pp. 21–28 (1962)
25. Moore, B., Peña, L., Rosu, G.: Program verification by coinduction. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 589–618. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_21
26. Moore, J.S.: Inductive assertions and operational semantics. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 289–303. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_27
27. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science, Springer, Heidelberg (2007). https://doi.org/10.1007/978-1-84628-692-6
28. Nipkow, T., von Oheimb, D.: Java$_{light}$ is type-safe - definitely. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 161–170 (1998)
29. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects Comput. **21**(1–2), 3–32 (2009)

30. Pierce, B.C.: The science of deep specification (keynote). In: Visser, E. (ed.) Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH), p. 1 (2016)
31. Plotkin, G.D.: A structural approach to operational semantics. Lecture notes, DAIMI FN-19 (1981)
32. Qin, S., Dong, J.S., Chin, W.-N.: A semantic foundation for TCOZ in unifying theories of programming. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 321–340. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_19
33. Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press, Cambridge (1998)
34. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceeding of 17th IEEE Symposium on Logic in Computer Science (LICS), pp. 55–74 (2002)
35. Schmidt, D.A.: Natural-semantics-based abstract interpretation (preliminary version). In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 1–18. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60360-3_28
36. Sergey, I., Nagaraj, V., Johannsen, J., et al.: Safer smart contract programming with Scilla. Proc. ACM Program. Lang. **3**(OOPSLA), 185:1–185:30 (2019)
37. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 471–482 (2013)
38. Sheng, F., Zhu, H., He, J., et al.: Theoretical and practical approaches to the denotational semantics for MDESL based on UTP. Formal Aspects Comput. **32**(2–3), 275–314 (2020)
39. Sozeau, M., Anand, A., Boulier, S., et al.: The MetaCoq project. J. Autom. Reason. **64**(5), 947–999 (2020)
40. Stefanescu, A., Park, D., Yuwen, S., et al.: Semantics-based program verifiers for all languages. In: 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 74–91 (2016)
41. Wood, G.: Ethereum: a secure decentralised generlised transaction ledger. https://gavwood.com/paper.pdf
42. Yang, Z., Lei, H.: Lolisa: formal syntax and semantics for a subset of the Solidity programming language. CoRR, abs/1803.09885 (2018)