Ghost Code in Action: Automated Verification of a Symbolic Interpreter

Benedikt Becker, Claude Marché

▶ **To cite this version:**

Benedikt Becker, Claude Marché. Ghost Code in Action: Automated Verification of a Symbolic Interpreter. VSTTE 2019 - 11th Working Conference on Verified Software: Tools, Techniques and Experiments, Jul 2019, New York, United States. 10.1007/978-3-030-41600-3_8 . hal-02276257

HAL Id: hal-02276257
https://inria.hal.science/hal-02276257

Submitted on 2 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Ghost Code in Action: Automated Verification of a Symbolic Interpreter*

Benedikt Becker[1] and Claude Marché[1]

Inria & LRI (CNRS, Univ. Paris-Sud), Université Paris-Saclay, Orsay, France
{benedikt.becker,claude.marche}@inria.fr

Abstract. Symbolic execution is a basic concept for the static analysis of programs. It amounts to representing sets of concrete program states as a logical formula relating the program variables, and interpreting sets of executions as a transformation of that formula. We are interested in formalising the correctness of a symbolic interpreter engine, expressed by an over-approximation property stating that symbolic execution covers all concrete executions, and an under-approximation property stating that no useless symbolic states are generated. Our formalisation is tailored for automated verification, that is the automated discharge of verification conditions to SMT solvers. To achieve this level of automation, we appropriately annotate the code of the symbolic interpreter with an original use of both *ghost* data and *ghost* statements.

Keywords: Deductive Program Verification · Symbolic Execution · Automated Theorem Proving · Ghost code

1 Introduction

Symbolic execution is one of the basic approaches for analysing program code. It amounts to representing a set of program states as a symbolic formula on the program variables, and interpreting each concrete execution in one of these states as one unique transformation of that formula. This technique allows one to detect if an undesired program state is reachable from a given set of initial inputs. In such a basic form, symbolic execution is a common idea behind more elaborated approaches to the static analysis of programs, such as *model checking* and *abstract interpretation*, and also for *test generation*. Formalisations of such approaches exist, for example the Verasco abstract interpreter [10] is formally specified and verified using the Coq proof assistant [3], but we are not aware of simpler, more basic formalisation of a symbolic execution engine. In the light of advanced verification efforts such as Verasco, the verification of a symbolic execution engine may seem an easy task. However, instead of using a proof assistant like Coq, we aim in this work at the use of *automated verifiers*, where proofs are not constructed interactively but carried out by automated theorem provers. In this work we use the Why3 environment [4] to achieve this task.

* This work has been partially supported by the ANR project CoLiS, contract number ANR-15-CE25-0001.

Informal Presentation of Symbolic Execution. Let us consider the toy program below from a mini-language in the style of IMP [11].

```
y := x - y - 1;
if y < 10 then x := y - 1 else y := 4 - x
```

Symbolically executing such a program means considering at once a set of possible inputs, a *symbolic state*, defined by a mapping from program variables to logical variables and a logical constraint. For example, let's consider

$$(x \mapsto u, y \mapsto v \mid 0 \leq u \leq 7 \wedge 1 \leq v \leq 11)$$

as an initial symbolic state for our toy program. The symbolic execution of the first assignment produces the new symbolic state

$$(x \mapsto u, y \mapsto w \mid \exists\, v.\ 0 \leq u \leq 7 \wedge 1 \leq v \leq 11 \wedge w = u - v - 1)$$

which represents the collection of concrete states that can be reached from any concrete state satisfying the initial constraint. The symbolic execution of the `if` statement results in a pair of symbolic states corresponding to the two branches:

$$\begin{aligned} & (x \mapsto t, y \mapsto w \mid \exists\, u, v.\ 0 \leq u \leq 7 \wedge 1 \leq v \leq 11 \wedge \\ & \qquad\qquad w = u - v - 1 \wedge w < 10 \wedge t = w - 1) \\ \cup\ & (x \mapsto u, y \mapsto t \mid \exists\, v, w.\ 0 \leq u \leq 7 \wedge 1 \leq v \leq 11 \wedge \\ & \qquad\qquad w = u - v - 1 \wedge w \geq 10 \wedge t = 4 - u) \end{aligned}$$

Using a constraint solver that is able to detect if a constraint is unsatisfiable, it is possible to discard some of these symbolic execution branches.

Handling Loops. Loops can be handled similarly to conditionals by executing all possible paths of concrete execution. In most cases, however, an infinite set of symbolic states will be produced, and symbolic execution will be non-terminating even on terminating programs. A simple solution is to restrict the number of loop iterations by a limit N given as a parameter of the symbolic execution, resulting in a N-bounded symbolic execution, also called *loop-k* [1].

When using a loop limit, we would like to distinguish between the executions that terminate normally and executions that reach the loop limit. For example, the symbolic execution of the program

```
y := 1;
while x > 1 do y := y * x; x := x - 1 done
```

with loop limit $N = 3$ and initial state $(x \mapsto v_1 \mid \texttt{true})$, results in two sets of symbolic states, where intermediate logical variables are existentially quantified in the constraint. One set of symbolic states represent normal *behaviour*:

$$\begin{aligned} & (x \mapsto v_1, y \mapsto v_2 \mid v_1 \leq 1 \wedge v_2 = 1) \\ \cup\ & (x \mapsto v_3, y \mapsto v_4 \mid \exists\, v_1, v_2.\ v_1 = 2 \wedge v_2 = 1 \wedge v_3 = 1 \wedge v_4 = 2) \\ \cup\ & (x \mapsto v_5, y \mapsto v_6 \mid \exists\, v_1, v_2, v_3, v_4.\ v_1 = 3 \wedge v_2 = 1 \wedge \\ & \qquad\qquad v_3 = 2 \wedge v_4 = 2 \wedge v_5 = 1 \wedge v_6 = 6) \end{aligned}$$

and one represents the abnormal behaviour obtained from reaching the loop limit:

$$(x \mapsto v_5, y \mapsto v_6 \mid \exists\, v_1, v_2, v_3, v_4.\ v_1 > 3 \wedge v_2 = 1 \wedge v_3 > 2 \wedge v_4 = 2 \wedge v_5 > 1 \wedge v_6 = 6)$$

In the remainder of this article we consider an additional abnormal behaviour obtained when trying to evaluate a variable that is not bound in the context. Symbolic execution thus produces a triple of finite sets of symbolic states, respectively for each of the Normal, LoopLimit and UnboundVar behaviours.

Introduction to Ghost Annotations. Adding *ghost annotations* to a program is a powerful versatile approach to proving advanced program properties [7]. A ghost annotation can be used at the level of data structures, e.g. as a ghost field in a record, or at the level of the code, e.g. as an assignment to a ghost variable. Ghost annotations may even be required to simply express a property, and they can greatly improve the degree of automation when using automated verifiers [5]. An essential property of ghost code is that it must never interfere with the execution of regular code [7], a property that is ensured statically in the verifier Why3 [4] we use in this work.

A specific use case of ghost code is the handling of existential quantifiers, for example in the post-condition of a program. Assume one has a program with a post-condition of the following form:

```
function f (x) returns y
  ensures { ∀ z. P(x, z) → ∃ t. Q(x, y, z, t) }
```

Proving a quantified post-condition of this form is generally out of reach of automated provers. A workaround is to turn the quantified variables into ghost parameters and results of the function:

```
function f (x, ghost z) returns (y, ghost t)
  requires { P(x, z) }
  ensures { Q(x, y, z, t) }
```

This reformulation is logically equivalent to the former one because the VC generation will universally quantify over parameters and existentially quantify over results. An important advantage of the ghost reformulation is that ghost code can be added to the body of f to compute an appropriate value of t. Moreover, when f is defined recursively, appropriate values of z can be passed to the recursive calls.

Structure of this paper. In Section 2, we present the toy mini-IMP language for which we formalise our symbolic execution engine. We present its syntax and the formal semantics of its concrete execution. As a first exercise, we develop a concrete interpreter and prove its correctness. In Section 3, we introduce our symbolic interpreter engine and a formalisation of the expected properties, and

$$
\begin{array}{ll}
\text{Literals:} & \bar{n} \in \bar{\mathbb{N}} \\
\text{Variables:} & x \in PVar \\
\text{Expressions:} & e ::= \bar{n} \mid x \mid e - e \\
\text{Instructions:} & i ::= \texttt{skip} \mid x \texttt{ := } e \mid i\texttt{;}i \mid \texttt{if } e \texttt{ then } i \texttt{ else } i \mid \texttt{while } e \texttt{ do } e
\end{array}
$$

Fig. 1. Syntax of the IMP language

present a technique based on ghost annotations to prove the properties using automated theorem provers. We conclude by a discussion of related and future work in Section 4.

Our Why3 development of the concrete and symbolic interpreters, and the material required to replay the proofs and compile and execute the interpreters, is available at http://toccata.lri.fr/gallery/symbolic_imp.en.html.

2 Presentation of the IMP Language

We consider a simple, imperative language with close resemblance to the *imp* language [11]. The abstract syntax of this language, called IMP, is shown in Fig. 1, and features two syntactic categories: expressions and instructions. An expression has type integer and is either an integer literal, a program variable from an infinite set $PVar$, or a subtraction operation. An instruction is either a variable assignment, or it combines other instructions into a sequence, a conditional, or a WHILE loop. Expressions are used as tests in conditionals and loops, where non-zero values represent the Boolean value TRUE.

2.1 Formal Semantics

We formalise the natural semantics of the IMP language using inductive rules, encoded by inductive predicates in Why3.

The natural semantics of expressions is defined by a judgement $e/_\Gamma \Downarrow \alpha$ with the inductive rules shown in Fig. 2. Expressions are evaluated in the context of a *concrete variable environment* $\Gamma : PVar \nrightarrow \mathbb{Z}$, a partial function from program variables to integers with domain $\mathrm{dom}(\Gamma)$. The behaviour of an expression is either normal and carries an integer, or it indicates the use of an unbound variable. A literal evaluates with normal behaviour to its integer value. A variable evaluates with normal behaviour to its value in the variable environment, if the variable is defined. The binary operation has normal behaviour and evaluates to the subtraction of the values of its operands, if both operands have normal behaviour. The evaluation of an unbound variable triggers the behaviour UnboundVar, which is propagated across binary operations.

The natural semantics of instructions is defined by a judgement $i/_\Gamma \Downarrow^N \beta/_{\Gamma'}$. The inductive rules are shown in Fig. 3. A concrete variable environment Γ con-

Expression behaviour: $\alpha ::= \mathsf{Normal}\ \mathbb{Z} \mid \mathsf{UnboundVar}$

$$\textsc{Literal}\ \frac{}{\bar{n}/_{\Gamma} \Downarrow \mathsf{Normal}\ n} \qquad \textsc{Var}\ \frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma[x] = n}{x/_{\Gamma} \Downarrow \mathsf{Normal}\ n} \qquad \textsc{Var-err}\ \frac{x \notin \mathrm{dom}(\Gamma)}{x/_{\Gamma} \Downarrow \mathsf{UnboundVar}}$$

$$\textsc{Sub}\ \frac{e_1/_{\Gamma} \Downarrow \mathsf{Normal}\ n_1 \qquad e_2/_{\Gamma} \Downarrow \mathsf{Normal}\ n_2}{e_1 - e_2/_{\Gamma} \Downarrow \mathsf{Normal}\ (n_1 - n_2)}$$

$$\textsc{Sub-err-1}\ \frac{e_1/_{\Gamma} \Downarrow \mathsf{UnboundVar}}{e_1 - e_2/_{\Gamma} \Downarrow \mathsf{UnboundVar}} \qquad \textsc{Sub-err-2}\ \frac{e_1/_{\Gamma} \Downarrow \mathsf{Normal}\ n_1 \qquad e_2/_{\Gamma} \Downarrow \mathsf{UnboundVar}}{e_1 - e_2/_{\Gamma} \Downarrow \mathsf{UnboundVar}}$$

Fig. 2. Semantics of expressions, which is formalized an inductive predicate in Why3

Instruction behaviour: $\beta ::= \mathsf{Normal} \mid \mathsf{UnboundVar} \mid \mathsf{LoopLimit}$

$$\textsc{Skip}\ \frac{}{\mathtt{skip}/_{\Gamma} \Downarrow^{N} \mathsf{Normal}/_{\Gamma}} \qquad \textsc{Assign}\ \frac{e/_{\Gamma} \Downarrow \mathsf{Normal}\ n}{x\ \mathtt{:=}\ e/_{\Gamma} \Downarrow^{N} \mathsf{Normal}/_{\Gamma[x \leftarrow n]}}$$

$$\textsc{Assign-err}\ \frac{e/_{\Gamma} \Downarrow \mathsf{UnboundVar}}{x\ \mathtt{:=}\ e/_{\Gamma} \Downarrow^{N} \mathsf{UnboundVar}/_{\Gamma}} \qquad \textsc{Seq}\ \frac{i_1/_{\Gamma} \Downarrow^{N} \mathsf{Normal}/_{\Gamma_1} \qquad i_2/_{\Gamma_1} \Downarrow^{N} \beta/_{\Gamma_2}}{i_1\,\mathtt{;}\,i_2/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma_2}}$$

$$\textsc{Seq-err}\ \frac{i_1/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma_1} \qquad \beta \neq \mathsf{Normal}}{i_1\,\mathtt{;}\,i_2/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma_1}} \qquad \textsc{Cond-true}\ \frac{e/_{\Gamma} \Downarrow \mathsf{Normal}\ n \qquad n \neq 0 \qquad i_1/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}}{\mathtt{if}\ e\ \mathtt{then}\ i_1\ \mathtt{else}\ i_2/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}}$$

$$\textsc{Cond-false}\ \frac{e/_{\Gamma} \Downarrow \mathsf{Normal}\ 0 \qquad i_2/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}}{\mathtt{if}\ e\ \mathtt{then}\ i_1\ \mathtt{else}\ i_2/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}}$$

$$\textsc{Cond-err}\ \frac{e/_{\Gamma} \Downarrow \mathsf{UnboundVar}}{\mathtt{if}\ e\ \mathtt{then}\ i_1\ \mathtt{else}\ i_2/_{\Gamma} \Downarrow^{N} \mathsf{UnboundVar}/_{\Gamma}} \qquad \textsc{While}\ \frac{e, i/_{\Gamma} \Downarrow^{0/N} \beta/_{\Gamma'}}{\mathtt{while}\ e\ \mathtt{do}\ i/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}}$$

Fig. 3. Semantics rules for instructions

$$\textsc{While-limit}\ \frac{n = N}{e, i/_\Gamma \Downarrow^{n/N} \mathsf{LoopLimit}/_\Gamma} \qquad \textsc{While-false}\ \frac{n < N \qquad e/_\Gamma \Downarrow \mathsf{Normal}\ 0}{e, i/_\Gamma \Downarrow^{n/N} \mathsf{Normal}/_\Gamma}$$

$$\textsc{While-loop}\ \frac{n < N \qquad e/_{\Gamma_1} \Downarrow \mathsf{Normal}\ m \qquad m \neq 0 \qquad i/_{\Gamma_1} \Downarrow^{N} \mathsf{Normal}/_{\Gamma_2} \qquad e, i/_{\Gamma_2} \Downarrow^{(n+1)/N} \beta/_{\Gamma_3}}{e, i/_{\Gamma_1} \Downarrow^{n/N} \beta/_{\Gamma_3}}$$

$$\textsc{While-test-err}\ \frac{n < N \qquad e/_\Gamma \Downarrow \mathsf{UnboundVar}}{e, i/_\Gamma \Downarrow^{n/N} \mathsf{UnboundVar}/_\Gamma}$$

$$\textsc{While-body-err}\ \frac{n < N \qquad e/_{\Gamma_1} \Downarrow \mathsf{Normal}\ m \qquad m \neq 0 \qquad i/_{\Gamma_1} \Downarrow^{N} \beta/_{\Gamma_2} \qquad \beta \neq \mathsf{Normal}}{e, i/_{\Gamma_1} \Downarrow^{n/N} \beta/_{\Gamma_2}}$$

Fig. 4. Semantics rules for an optionally bounded while loop

stitutes the program state, which can be modified by the evaluation of an instruction to Γ'. The value $N \in \mathbb{N} \cup \{\infty\}$ specifies an optional iteration limit for WHILE loops, where $N = \infty$ disables the iteration limit. N is a parameter and kept constant across the evaluation of instructions. The instruction behaviour β is either normal, indicates an unbound variable in a sub-expression, or the reaching of the loop limit. An assignment changes the value of the variable in the environment to the value of an expression, if the expression evaluates with normal behaviour. Expressions are used as tests in conditionals and WHILE loops, and the integer 0 is interpreted as the Boolean value FALSE, and any non-zero value is interpreted as TRUE. Abnormal behaviour ($\mathsf{UnboundVar}$, $\mathsf{LoopLimit}$) of sub-expressions and sub-instructions is propagated to the behaviour of the instruction.

The semantics of loops is defined by a judgement $e, i/_\Gamma \Downarrow^{n/N} \beta/_{\Gamma'}$ for a test expression e and a loop body i. The inductive rules are shown in Fig. 4. When the loop counter $n \in \mathbb{N}$ reaches the loop limit N, the evaluation of the loop terminates with behaviour $\mathsf{LoopLimit}$. All other rules require the number of previous iterations smaller than the loop limit. The evaluation of a loop terminates with normal behaviour when the test expression evaluates to zero. Otherwise, and if the evaluation of the loop body has normal behaviour, the evaluation of the loop continues with an increased loop counter, and the result of the continuation determines the result of the loop. Abnormal behaviour in the test expression or loop body is propagated to the behaviour of the loop.

2.2 Concrete Execution

A concrete interpreter of the IMP language is implemented in the programming language of Why3. A global, imperative variable environment from module `Env`

```
module Env
  type env = abstract { mutable Γ: PVar ↛ ℤ }

  val empty (_:unit) : env ensures { ∀ x. x ∉ dom(result.Γ) }

  val set (e:env) (x:PVar) (n:ℤ) : unit
    writes { e }
    ensures { e.Γ = (old e.Γ)[x ← n] }

  exception Unbound_var

  val find (e:env) (x:PVar) : ℤ
    ensures { x ∈ dom(e.Γ) ∧ result = (e.Γ x) }
    raises { Unbound_var → x ∉ dom(e.Γ) }
end
```

Listing 1. The imperative variable environment `Env`

(Listing 1) constitutes the program state. The type of the variable environment is left abstract in the Why3 program, and can be used only through the functions of module `Env`. These are specified via post-conditions in terms of the partial function Γ that represents the environment in the logical specifications. The interpretation of expressions and instructions is implemented by two functions that map the language constructs of the IMP language to the corresponding language constructs of Why3 (see Listing 2). The functions of the interpreter return normally in case of a normal evaluation, and exceptions are raised for unbound variables. The interpretation of loops is unbound in the concrete interpreter.

The soundness of the concrete interpreter with respect to the formal semantics without loop limit ($N = \infty$) is expressed by post-conditions for normal behaviour (keyword `ensures`) and exceptional behaviour (keyword `raises`) (see Listing 2). Each post-condition contains a semantic judgement that describes the transformation of the global variable environment before running the interpreter (`old env`) to the variable environment after running the interpreter (`env`). A loop invariant is required to verify the soundness of the while loop in Why3 that executes the loops of IMP. A ghost variable Γ_0 retains the initial variable environment and a mutable ghost variable n acts as an iteration counter. The loop invariant then states that the result of the loop when starting in the current program state (`env.`Γ) is the result of the loop when starting in the initial variable environment Γ_0.

Why3 splits the proof into 8 verification goals for `interp_exp` (covering also the termination) and 16 goals for `interp_ins`, corresponding to the execution paths of the interpreter functions. All verification conditions were automatically proven by either CVC4 or Alt-Ergo in fractions of a second after applying basic interactive proof steps in Why3 [6].

```
let env = Env.empty () (* A global, imperative variable environment *)

let rec interp_exp (e : expression) : int
  variant { e }
  ensures { e/_{env.Γ} ⇓ Normal result }
  raises  { Env.Unbound_var → e/_{env.Γ} ⇓ UnboundVar }
= match e with
    | Lit n → n
    | Var x → Env.find env x
    | Sub e1 e2 → interp_exp e1 - interp_exp e2
  end

let rec interp_ins (i : instruction) : unit
  writes  { env.Γ }
  ensures { i/_{(old env.Γ)} ⇓^∞ Normal/_{env.Γ} }
  raises  { Env.Unbound_var → i/_{(old env.Γ)} ⇓^∞ UnboundVar/_{env.Γ} }
= match i with
  | Skip → ()
  | Assign x e → Env.set env x (interp_exp e)
  | Seq i1 i2 → interp_ins i1; interp_ins i2
  | If e i1 i2 → if interp_exp e ≠ 0 then interp_ins i1 else interp_ins i2
  | While e i →
    let ghost Γ0 = env.Γ in let ghost ref n = 0 in
    while interp_exp e ≠ 0 do
      invariant { ∀β/_{Γ'}. e, i/_{env.Γ} ⇓^{n/∞} β/_{Γ'} → e, i/_{Γ0} ⇓^{0/∞} β/_{Γ'} }
      interp_ins i; n ← n + 1
    done
  end
```

Listing 2. A concrete interpreter for the IMP language

3 Symbolic Execution

Following the informal presentation of symbolic execution in Section 1, we recall that a symbolic state $(\sigma \mid C)$ has two components: a symbolic environment $\sigma : PVar \nrightarrow SVar$, which is a partial function from program variables to symbolic variables with domain $\mathrm{dom}(\sigma)$, and a constraint C on symbolic variables. Finite sets of symbolic states are designated by symbol Σ. A symbolic interpreter is a function $\texttt{sym\_interp\_ins}_N(\sigma \mid C)(i)$ that is parameterised by a loop limit $N \in \mathbb{N}$, and takes an initial symbolic state and an instruction as arguments. The symbolic interpreter returns a set Σ_* of symbolic result states $(\sigma \mid C)_\beta$, i.e., symbolic states annotated with a behaviour $\beta \in \{\mathsf{Normal}, \mathsf{UnboundVar}, \mathsf{LoopLimit}\}$. For any behaviour β, Σ_β designates the symbolic states in Σ_* with behaviour β.

The constraint language for IMP is shown in Fig. 5. A constraint is either trivially true, the equality or disequality between two symbolic expressions, the conjunction of two constraints, or an existential quantification of a variable over a constraint. A symbolic expression corresponds to a program expression with

$$
\begin{array}{lll}
\text{Symbolic variables:} & v & \in \mathit{SVar} \\
\text{Symbolic expressions:} & se & ::= n \mid v \mid se - se \\
 & & \in \mathit{Sym\text{-}expr} \\
\text{Constraints:} & C & ::= \top \mid se = se \mid se \neq se \mid C \wedge C \mid \exists\, v.C \\
 & & \in \mathit{Constraint}
\end{array}
$$

Fig. 5. Constraint language for the symbolic interpretation of IMP

symbolic variables in place of program variables. The application of a symbolic environment to a program expression, $\sigma(e)$, is defined by the natural extension of the symbolic environment to program expressions. The application results in a symbolic expression, if the symbolic environment is defined on all variables occurring in the expression, and is undefined otherwise.

An interpretation, $\rho : \mathit{SVar} \nrightarrow \mathbb{Z}$, is a partial function from symbolic variables to integers with domain $\mathrm{dom}(\rho)$. The application of an interpretation to a symbolic expression, $\rho(se)$, is defined as the natural extension of ρ to symbolic expressions, if all variables in se are in $\mathrm{dom}(\rho)$, and undefined otherwise. An interpretation ρ is a solution of a constraint C, denoted $\rho \models C$, if the domain of ρ contains all symbolic variables in C and the formula obtained from C by substituting all symbolic variables by their values in ρ, is true. A solution $\rho \models C$ to the constraint of a symbolic state $(\sigma \mid C)$ defines a concrete variable environment by composition with the symbolic environment, $\Gamma = \rho \circ \sigma$. These concrete environments are the instances of the symbolic state.

3.1 Correctness Properties of a Symbolic Interpreter

The correctness of a concrete interpreter is defined by two properties: that every evaluation result derivable by the semantic rules is produced by the interpreter (completeness), and that every result of the interpreter is derivable by the semantic rules (soundness). Symbolic interpretation, however, describes program execution non-deterministically: the initial symbolic state represents a potentially infinite set of initial concrete environments, and the result states represent all concrete environments resulting from all initial environments. The correctness of a symbolic interpreter is defined by two properties that relate the instances of the symbolic states with semantic judgements of the concrete semantics: *over-approximation* and *under-approximation*. Simplified, a symbolic execution is an over-approximation of the concrete execution, if executing an instruction in an instance of the initial symbolic state results in an instance of one of the symbolic result states. Over-approximation is also called *coverage* [2]. A symbolic interpretation is an under-approximation, if every instance of the symbolic result states is the evaluation result of an instance of the initial symbolic state. Under-approximation is also called *precision* [2].

Given the symbolic result states Σ_* resulting from the symbolic execution $\texttt{interp\_ins}_N(\sigma \mid C)(i)$, the correctness properties can be formalised as follows.

Property 1 (Over-approximation) *The symbolic execution is an over-approximation of the concrete semantics, if for any solution ρ of the initial constraint, $\rho \models C$, and concrete evaluation result $\beta/_{\Gamma'}$ with $i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\Gamma'}$, there exists a symbolic result state $(\sigma' \mid C')_\beta \in \Sigma_*$ and a solution $\rho' \models C'$ such that $\Gamma' = \rho' \circ \sigma'$.*

Property 2 (Under-approximation) *The symbolic execution is an under-approximation of the concrete semantics, if for any symbolic result state $(\sigma' \mid C')_\beta \in \Sigma_*$ and solution $\rho' \models C'$, there exists a solution ρ of the initial constraint, $\rho \models C$, such that $i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\rho'\circ\sigma'}$.*

Reformulation using Ghost Annotations. The correctness properties contain existential quantifications over solutions for the initial and result symbolic states. As explained in Section 1, existential quantifications can be challenging for automatic theorem provers, but are a typical use case of ghost annotations. We thus associate each symbolic state with a ghost interpretation. A ghost-extended symbolic state, $(\sigma \mid C;\ \rho)$ thus consists of a symbolic environment, a constraint, and a ghost interpretation. From now on, we will only refer to ghost-extended symbolic states. When used in a symbolic interpreter the ghost interpretations do not influence the symbolic execution, but only support proving the properties. They allow for substituting universally or existentially quantified solutions in the properties by the ghost interpretations associated with the initial symbolic state or a resulting symbolic state.

Given the set of extended symbolic result states Σ_* resulting from a symbolic execution $\texttt{interp\_ins}_N(\sigma \mid C;\ \rho)(i)$, the above correctness properties can be reformulated equivalently as follows:

Property 3 (Ghost-reformulated Over-approximation) *The symbolic execution is an over-approximation of the concrete semantics, if, assuming that the interpretation ρ of the initial state is a solution of the initial constraint, $\rho \models C$, and given a concrete evaluation result $\beta/_{\Gamma'}$ with $i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\Gamma'}$, there exists a symbolic result state $(\sigma' \mid C';\ \rho')_\beta \in \Sigma_*$ such that $\rho' \models C'$ and $\Gamma' = \rho' \circ \sigma'$.*

Property 4 (Ghost-reformulated Under-approximation) *The symbolic execution is an under-approximation of the concrete semantics, if for symbolic result state $(\sigma' \mid C';\ \rho')_\beta \in \Sigma_*$ such that $\rho' \models C'$, it holds that $\rho \models C$ and $i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\rho'\circ\sigma'}$.*

3.2 Implementation of the Symbolic Interpreter

We implemented a symbolic interpreter for IMP in the Why3 programming language with main function `sym_interp_ins` (see Listing 3), which is recursively defined over the structure of the instruction. The reformulated correctness properties are formalised as post-conditions of the function.

```
val sym_interp_ins_N(σ | C; ρ)(i) : Σ*
  ensures { (* Over-approximation *) ρ ⊨ C → ∀ β/Γ'. (i/ρ∘σ ⇓^N β/Γ') →
    ∃(σ' | C'; ρ')_β ∈ result. ρ' ⊨ C' ∧ Γ' = ρ' ∘ σ' }
  ensures { (* Under-approximation *)
    ∀(σ' | C'; ρ')_β ∈ result. ρ' ⊨ C' → ρ ⊨ C ∧ (i/ρ∘σ ⇓^N β/ρ'∘σ') }
```

Listing 3. Signature of the basic symbolic interpreter function with correctness properties encoded as post-conditions

```
1  val fresh (ghost ρ) : SVar
2    ensures { result ∉ dom(ρ) }
3
4  val existentially_quantify (v) (C) : Constraint
5    ensures { vars(result) ⊆ vars(∃ v. C) }
6    ensures { ∀ ρ.  ρ ⊨ result ↔ ρ ⊨ ∃ v. C }
7
8  predicate ρ ⊑ ρ' =
9    dom(ρ) ⊆ dom(ρ') ∧ ∀ v ∈ dom(ρ). ρ(v) = ρ'(v)
10
11 type sym_state = (σ | C; ghost ρ)
12   invariant { codom(σ) ∪ vars(C) ⊆ dom(ρ) }
13
14 let rec sym_interp_ins_N(σ | C; ρ)(i) : Σ*
15   ensures { ... (* Over-approximation and under-approximation *) }
16   ensures { (* Result interpretations extend the initial interpretation *)
17     ∀(σ' | C'; ρ')_β ∈ result → ρ ⊑ ρ' }
18   = match i with ...
19   | Assign x e →
20     try
21       let se = σ(e) in
22       let v = fresh ρ in
23       let σ' = σ[x ← v] in
24       let C' =
25         if x ∈ dom(σ) then existentially_quantify (σ(x)) (C ∧ (v = se))
26         else C ∧ (v = se) in
27       let ghost ρ' = ρ[v ← ρ(se)] in
28       {(σ' | C'; ρ')_Normal}
29     with UnboundVar (* from σ(e) *) → {(σ | C; ρ)_UnboundVar} end
```

Listing 4. Symbolic execution of the assignment, with additional post-condition and state invariant to ensure the correct use of the interpretation as an environment of witnesses to existentially quantified variables

Assignment and Quantification over a Variable. To execute an assignment x `:=` e, the variable x is assigned in the symbolic environment to a fresh symbolic variable v, and an equality constraint between v and the symbolic expression corresponding to e is added, if the symbolic environment is defined on all variables in

e (see Listing 4, Line 21 et seq.). The interpretation ρ is updated by assigning the fresh variable to the value of the symbolic expression in the interpretation. If the program variable was already bound to a variable v' in σ, then the variable v' becomes inaccessible from the symbolic environment and the remaining program. The constraint is replaced by an existential quantification over v'. More precisely, we apply the function `existentially_quantify` (Line 25) to delegate the construction of the quantification to a constraint solver, which allows for simplifying the constraint by quantifier elimination. The requirements for function `existentially_quantify` are expressed by its post-conditions: it does not introduce any new variables and it produces a constraint equivalent to an explicit existential quantification.

If the symbolic environment is undefined on any variable in e, the initial symbolic state annotated with behaviour UnboundVar comprises the singleton result set (Line 29).

Updating the ghost interpretation when executing an assignment implies that the interpretation retains values of symbolic variables that satisfy the constraint, and that the interpretation serves as an environment of witnesses to existentially quantified variables. This original aspect must be reflected by the following definition of the predicate $\rho \models C$, in which the last case, the one of existential quantifiers, is to be particularly emphasised:

$$\rho \models C \text{ iff. } \mathrm{vars}(C) \subseteq \mathrm{dom}(\rho) \wedge \begin{cases} \top & \text{when } C = \top \\ \rho(se_1) = \rho(se_2) & \text{when } C = (se_1 = se_2) \\ \rho(se_1) \neq \rho(se_2) & \text{when } C = (se_1 \neq se_2) \\ \rho \models C_1 \wedge \rho \models C_2 & \text{when } C = C_1 \wedge C_2 \\ \rho \models C_1 & \text{when } C = \exists\, v. C_1 \end{cases}$$

The case of the existential quantifier is non-standard: instead of pretending that "there exists a value for the quantified variable v such that ...", we go even further in the use of ghost annotations by requiring that the ghost interpretation ρ already holds the adequate value for v. This choice is crucial to facilitate the application of automatic theorem provers. A drawback, however, is that we lose invariance with respect to α-renaming, because the witnesses in the interpretation are identified by their exact variable names. This requires extra care with the concept of "fresh" variables: these have to be fresh even with respect to existentially quantified variables. We ensure this property in the implementation by three means:

1. The function `fresh` has an interpretation as a ghost argument and ensures that the resulting variable is not in the domain of the interpretation (Listing 4, Line 2).
2. An invariant of the symbolic state ensures that the domain of the interpretation covers all variables in the codomain of the symbolic environment and all variables in the constraint, including existentially quantified variables (Line 12).

```
let rec sym_interp_ins_N(s)(i) =
  match i with ...
  | Seq i1 i2 →
    let Σ* = sym_interp_ins_N(s)(i1) in
    let Σ'* = sym_interp_ins'_N(Σ_Normal)(i2) in
    Σ_UnboundVar ∪ Σ_LoopLimit ∪ Σ'*
```

Listing 5. Symbolic execution of sequences

3. The symbolic execution function always *extends* the interpretation, i.e., the domain of the initial interpretation is a subset of the domains of the resulting interpretation, and all values of the initial interpretation are retained in the resulting interpretations (defined by predicate $\sqsubseteq$, Line 9). This property is ensured by a post-condition (Line 17).

Sequences and Sets of Initial States. The concrete semantics of the sequence of two instructions specifies that the first instruction is evaluated first, and the second instruction is evaluated only if the behaviour of the first instruction was normal. Similarly, the symbolic execution of a sequence starts by executing the first instruction (see Listing 5). The second instruction is executed in the context of the resulting symbolic states annotated with behaviour Normal, using an auxiliary function $\texttt{sym\_interp\_ins'}_N(\Sigma)(i) : \Sigma_*$ that operates on a set of initial states Σ, applies `sym_interp_ins` on each element of Σ, and joins the resulting symbolic state sets. The result of the execution of the sequence is the union of the symbolic result states representing abnormal behaviour in the first instruction, and the symbolic result state set from the second instruction.

Conditionals and State Pruning. Listing 6 shows the symbolic execution of the conditional instruction. The test expression e is converted into a symbolic expression se by applying the symbolic environment. The exception `UnboundVar`, raised when a variable in e is undefined in σ, results in a singleton set of the initial state annotated with behaviour UnboundVar. The instructions i_1 and i_2 are interpreted in symbolic states that extend the initial symbolic state by constraints stating that se is different from 0, or equal to 0, respectively. If the constraint corresponding to one of the branches is unsatisfiable, the branch is pruned by assuming an empty set. Our symbolic interpreter uses a potentially incomplete procedure `maybe_sat` for testing unsatisfiability of a constraint. If `maybe_sat` C returns False then constraint C does not have a solution. It may or may not have a solution if the procedure returns True.

Loops. The function $\texttt{sym\_interp\_loop}_{n/N}(\sigma \mid C;\ \rho)(e, i) : (\Sigma_\beta)_\beta$ executes a loop with test expression e and body i (see Listing 7). When the loop counter n reaches the loop limit N, the initial symbolic state is returned as a singleton

```
val maybe_sat (C : Constraint) : 𝔹
  ensures { result = False → ∄ρ. ρ ⊨ C }

let rec sym_interp_ins_N(σ | C; ρ)(i) =
  match i with ...
  | If e i1 i2 →
    try
      let se = σ(e) in
      let Σ* = (* then-branch *)
        if maybe_sat (C ∧ (se ≠ 0))
        then sym_interp_ins_N(σ | C ∧ (se ≠ 0); ρ)(i1)
        else ∅ in (* prune then-branch *)
      let Σ'* = (* else-branch *)
        if maybe_sat (C ∧ (se = 0))
        then sym_interp_ins_N(σ | C ∧ (se = 0); ρ)(i2)
        else ∅ in (* prune else-branch *)
      Σ* ∪ Σ'*
    with UnboundVar (* from σ(e) *) → {(σ | C; ρ)_UnboundVar} end
```

Listing 6. Symbolic execution of conditions with state pruning

state set annotated with behaviour LoopLimit. Otherwise, the loop is executed. The normal termination of the loop is represented by the singleton set of the initial symbolic state with the additional constraint that the symbolic test expression is false and annotated with behaviour Normal. The loop body is executed in the symbolic state with the additional constraint that the symbolic test expression is true. The continuation of the loop is executed by a call to function `sym_interp_loop'` that executes the loop in the context of a set of symbolic states with an increased loop counter. If the constraint representing the termination (or continuation) of the loop is unsatisfiable according to function `maybe_sat`, the termination (or further execution) of the loop is pruned.

3.3 Proofs of the Symbolic Properties

Post-conditions ensure the under-approximation, over-approximation, and extension of interpretations of the functions implementing the symbolic interpreter, namely `sym_interp_ins` and `sym_interp_loop`, and their variants operating on sets of initial symbolic states. The post-conditions, required lemmas, and termination criteria of the symbolic interpreter functions amount to 31 verification goals, to which we applied 86 lightweight interactive transformations [6]. Most transformations were required to separate the post-conditions of function `sym_interp_loop` by its execution paths into verification conditions that are within reach of automatic theorem provers. The resulting proof tree has 186 leaf verification conditions, which were discharged to the automatic theorem provers CVC4 1.6, Alt-Ergo 2.2.0, and Eprover 2.2. Each goal was verified by one prover, trying the

```
with sym_interp_loop_{n/N}(σ | C; ρ)(e, i) : Σ_* =
  if n = N (* loop limit reached *)
  then {(σ | C; ρ)_LoopLimit}
  else
    try
      let se = σ(e) in
      let Σ_*^loop = (* continue loop *)
        if maybe_sat (C ∧ (se ≠ 0)) then
          let Σ_* = sym_interp_cmd_N(σ | C ∧ (se ≠ 0); ρ)(i) in
          let Σ'_* = sym_interp_loop'_{(n+1)/N}(Σ_Normal)(e, i) in
          Σ_UnboundVar ∪ Σ_LoopLimit ∪ Σ'_*
        else ∅ in (* prune loop continuation *)
      let Σ_*^term = (* loop termination *)
        if maybe_sat (C ∧ (se = 0))
        then {(σ | C ∧ (se = 0); ρ)_Normal}
        else ∅ in (* prune loop termination *)
      Σ_*^loop ∪ Σ_*^term
    with UnboundVar (* from σ(e) *) → {(σ | C; ρ)_UnboundVar} end
```

Listing 7. Symbolic execution of loops

Table 1. The use of different automatic theorem provers in the verification conditions of the symbolic interpreter functions with processing time in seconds.

Prover	Verification conditions	Fastest	Slowest	Average
CVC4 1.6	162	0.03	2.57	0.26
Alt-Ergo 2.2.0	20	0.03	3.59	0.42
Eprover 2.2	4	0.09	0.31	0.20

three provers in the given order. The use and processing times of the provers is given in table Table 1 (on a machine with four cores Intel i7-8650U@1.90GHz, 16GB RAM, and running Debian 9.9).

3.4 Execution and Test of the Symbolic Interpreter

The Why3 environment offers an *extraction* feature that allows one to automatically generate OCaml code from a Why3 program. An OCaml program was extracted from our symbolic interpreter, and compiled together with hand-written OCaml code to experiment with that interpreter. All ghost annotations are removed during extraction.

To perform the extraction, some information must be given to Why3, under the form of an *extraction driver*, a simple file that explain how the abstract symbols of the Why3 code must be mapped to OCaml. In particular, the constraint solver required to execute the code must be provided by an external OCaml library: we use the one from the Alt-Ergo prover. The abstract type of symbolic

variables is substituted by an OCaml type that is private to a module. The substitution of the function `fresh` creates universally fresh variables to comply with the post-condition given in the Why3 program. The resulting code was tested against simple examples. The performance of the generated code is satisfactory in the sense that the Why3 extraction procedure, which erases the ghost code, produces an OCaml code that is as efficient as a code that would have been written by hand.

For reproducibility, the source code of the formal semantics, the concrete interpreter, the symbolic interpreter, the proof session to replay the proofs, and the extraction driver are available at http://toccata.lri.fr/gallery/symbolic_imp.en.html. See the file `README.md` for the required dependencies.

4 Conclusions, Related Work and Future Work

We presented in this article a formalisation of two correctness properties of symbolic interpreter engines, over-approximation and under-approximation. We employed advanced *ghost* annotations of data and code of the symbolic interpreter to discharge the generated verification conditions to automated theorem provers.

A natural question is whether our approach can scale to a symbolic execution tool on a more complex language. First, we believe that our technique using ghost code for automating proofs is already well demonstrated on the essential constructions of assignment, conditionals and loops, so that it should apply similarly on languages with similar control structures. Indeed, we believe the complexity of symbolic execution tools for complex languages relies more on the complexity of data, which must be handled by the constraints and not the symbolic engine itself. Second, as a matter of fact, we recently finish to transfer the correctness properties and proof techniques developed in this article to the CoLiS language [8]. The CoLiS language is an intermediate language for a subset of the POSIX shell language with formally defined and easily understandable semantics. It has been developed to statically analyse Debian maintainer scripts, and we aim at identifying errors in maintainer scripts by symbolically executing the corresponding CoLiS scripts. Indeed we have already been able to identify issues in some of those scripts. We represent the file system symbolically using *feature tree constraints* [9]. Program variables, however, are statically known in Debian maintainer scripts, and represented concretely. This results in simplified correctness properties, where the only variable of the symbolic environment represents successive values of the file system's root node.

Acknowledgements. We would like to thank Nicolas Jeannerod, Ralf Treinen, Mihaela Sighireanu and Yann Regis-Gianas, partners of the CoLiS project, for their input and remarks on the design of the symbolic interpreter and the formulation of expected properties. We also thank Burkhart Wolff for his feedback about related work on symbolic execution.

References

1. Albert, E., Arenas, P., Gómez-Zamalloa, M., Rojas, J.M.: Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures, chap. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency, pp. 263–309. Springer Verlag (2014). https://doi.org/10.1007/978-3-319-07317-0_7
2. Arusoaie, A., Lucanu, D., Rusu, V.: A Generic Framework for Symbolic Execution: Theory and Applications. Research Report RR-8189, Inria (Sep 2015), https://hal.inria.fr/hal-00766220
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science, Springer-Verlag (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer (STTT) **17**(6), 709–727 (2015). https://doi.org/10.1007/s10009-014-0314-5, http://hal.inria.fr/hal-00967132/en, see also http://toccata.lri.fr/gallery/fm2012comp.en.html
5. Clochard, M., Marché, C., Paskevich, A.: Deductive verification with ghost monitors (Nov 2018), https://hal.inria.fr/hal-01926659, working paper
6. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: Proceedings of the Fourth Workshop on Formal Integrated Development Environment, F-IDE, Oxford, UK, July 14, 2018 (2018), https://hal.inria.fr/hal-01936302
7. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods in System Design **48**(3), 152–174 (2016). https://doi.org/10.1007/s10703-016-0243-x, https://hal.archives-ouvertes.fr/hal-01396864v1
8. Jeannerod, N., Marché, C., Treinen, R.: A Formally Verified Interpreter for a Shell-like Programming Language. In: VSTTE 2017 - 9th Working Conference on Verified Software: Theories, Tools, and Experiments. Lecture Notes in Computer Science, vol. 10712. Heidelberg, Germany (Jul 2017), https://hal.archives-ouvertes.fr/hal-01534747
9. Jeannerod, N., Treinen, R.: Deciding the first-order theory of an algebra of feature trees with updates. In: International Joint Conference on Automated Reasoning. pp. 439–454. Springer (2018), https://hal.archives-ouvertes.fr/hal-01760575
10. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 247–259. ACM, Mumbai, India (Jan 2015). https://doi.org/10.1145/2676726.2676966, https://hal.inria.fr/hal-01078386
11. Winskel, G.: The formal semantics of programming languages: an introduction. MIT press (1993)