Formal Verification of Symbolic Bug Finders

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Arthur Correnson

Saarbrücken, January 2024

Supervisor:	Prof. Bernd Finkbeiner
Reviewer:	Prof. Bernd Finkbeiner
	Dr. Dominic Steinhöfel
Submission:	January 12, 2024

Abstract

Testing the robustness of software systems is a fundamental concern. To do so, a common approach is to rely on automated testing methods to search for *bugs* and potential errors before deploying systems. To be reliable, a testing tool needs to be precise and exhaustive: it should not report false alarms, and not miss too many bugs. In this thesis, we use the Coq proof assistant to implement and prove the correctness of an automated bug finder based on symbolic execution. More precisely, we prove that our bug finder is precise (it cannot report false alarms) and relatively exhaustive (it will enumerate all bugs) against the formal semantics of a target programming language.

Acknowledgements

First, I would like to thank Dr. Dominic Steinhöfel for introducing me to the exciting world of symbolic execution, and supporting me through the development and the publication of the research results presented in this thesis. Then, I would also like to thank professor Finkbeiner for welcoming me in his fantastic research group. Finally, I especially thank Rahel Pauli for being an extraordinary life partner and for supporting me daily through the long process of writing a thesis.

Saarbrücken, 12 January, 2024

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, 12 January, 2024

Contents

Chapter 1

Introduction

1.1 Background and motivations

On the need to verify critical software. Computer programs are everywhere, including in applications where human lives are at stake, such as public transport, medical facilities, or military equipment. In such use cases, program failures (so-called *bugs*) cannot be tolerated, and programs have to be carefully checked before being deployed. When programs are reasonably small, their authors can manually review them. However, for large software with millions of lines of source code, manual verification becomes intractable, and we need to rely on automated methods to check that programs behave as intended. There exist two main categories of such automated verification techniques: *automated testing* and *automated proving*. Automated testing, also called automated bug finding, aims to find scenarios where programs fail. For example, fuzz-testing [Fio+20] is an automated bug finding methods that repeatedly execute programs with random inputs until the program crashes. Other methods, such as symbolic execution [Kin76] or bounded model-checking [Bie+09] can also simulate program executions simultaneously to find more bugs faster. On the other hand, automated provers aim at formally proving the absence of bugs using mathematical reasoning. Examples of automated proving techniques are abstract interpretation [CC77; Bla+03], automated deductive verification [Dij75; Bau+21], and model checking [CES09].

The limitations of automated verifiers. Using automated verifiers (automated provers or automated bug finders[1]) can greatly improve the reliability of critical software. How-

[1]The word "automated verifier" is often used as a synonym for "automated prover" in the literature. In this thesis, we call "verifier" any automatic tool used to check the correctness of programs. This includes automated bug finders.

ever, relying on automated verification is not enough. Indeed, why should we trust automated verifiers? In particular, an automated verifier could contain a bug leading to inconsistent results. For example, a bug finder could miss a critical bug or, more subtly, spuriously detect a bug in a bug-free program. Program provers suffer from the same problem: an error in a program prover could lead to an incorrect program being labeled as correct. To overcome these limitations, it is crucial to question the very correctness of automated verification tools in general [God05; Ler11]. In this thesis, we focus on automated testing and we propose to formally prove the correctness of an automated bug finder.

Proof assistants as a trusting basis. One way to truly trust program verifiers could be to apply automated proof methods to establish once and for all the correctness of a program prover or a bug finder. Unfortunately, this would only slightly shift the problem from trusting the verifier to trusting the verifier's verifier. To break this seemingly endless loop, a widespread solution is to use a *proof assistant* as a trusting basis. Proof assistants are software allowing to write programs, state theorems, and develop mathematical proofs thereof in one unified formal language. Contrary to other tools such as automated theorem provers, a proof assistant requires the proofs to be completely written by the users. The sole task of the assistant is then to validate the correctness of every reasoning step in the proofs. This task is elementary enough to be performed by a very small and trustworthy proof-checking algorithm (called the *kernel* of the proof assistant) that implements a minimal set of reasoning principles and inference rules. Typically, the kernel of a proof assistant is small enough to be manually verified by an expert. All other features of the proof assistant are built on top of the proof kernel making it the only critical component that needs to be trusted. Thanks to their incomparable level of reliability, proof assistants are tools of choice to prove the correctness of critical programs such as automated verifiers.

1.2 Contributions

Summary of the contributions. In this thesis, we propose to formally prove the correctness of an automated bug finder using the Coq proof assistant[2]. Our bug finder is based on symbolic execution, a common method used in automated testing to explore many executions of a program at once. It targets a simple imperative programming language with integer arithmetic and loops. The source code of this bug finder is accompanied with a machined-checkable proof that it is precise and exhaustive: it finds only true bugs, and it can find all the bugs (if enough time and memory is provided). The correctness of tools based on symbolic execution has already been studied [Kne91;

[2]`https://coq.inria.fr`

BB19; Ste20; Por+22; Keu+22b]. However, to the best of our knowledge, this is the first attempt at verifying the correctness of an automated bug finder based on symbolic execution using a proof assistant.

Publications. The work presented in this thesis builds on the following peer reviewed publications:

[CS23] *Engineering a Formally Verified Automated Bug Finder*

[Cor] *À la recherche de tous les vrais bugs*

Data Availability. The source code and the Coq proofs described in this thesis can be found online at the address `https://github.com/acorrenson/master_thesis_data`.

1.3 Structure of the thesis

The thesis is structured in 7 chapters:

- Chapter 2 gives an overview of symbolic execution and highlights some of the challenges that arise when studying the correctness of symbolic execution-based bug finders. It also introduces the Coq proof assistant.
- Chapter 3 focuses on the formalization of a toy programming language called BUG, and presents the notion of symbolic semantics as a trustworthy basis to understand what it means to symbolic execute programs.
- Chapter 4 discusses how to implement and verify the correctness of a symbolic interpreter based on a symbolic semantics.
- Chapter 5 presents a slightly improved symbolic semantics that can perform on-the-fly pruning of the state space. The implementation of this semantics as an interpreter is also discussed.
- Chapter 6 explains how to build and verify a usable bug finder based on the symbolic interpreters presented in the two previous chapters.
- Chapter 7 presents a selection of related projects and research papers and compares them with the contributions of this thesis.
- Chapter 8 concludes the thesis and presents future research directions.

Chapter 2

Preliminaries

2.1 Automated bug-finding by symbolic execution

To find bugs, one could repeatedly execute programs with various combinations of inputs to identify these that make a program crash. This relatively simple idea led to the development of automated bug-finding tools called *fuzzers* [MFS90; Fio+20]. While this brute-force approach can find many flaws in large software, it still has limitations. In some cases, the odds of identify a bug-triggering inputs are so low that fuzzers are of little use [Bun+21]. Consider for example the following simple program:

```
def crash_if_42(x : int)
  if x == 42:
    fail()
  return
```

Executing the python function `suspicious` with $\mathtt{x} = 42$ as an input makes the program crash. However, assuming a 32 bits representation of integers, the probability of randomly generating exactly the input 42 is only $\frac{1}{2^{32}}$! To overcome this intrinsic limitation of fuzzing, one can use *symbolic execution* techniques. Symbolic execution was first introduced by King [Kin76] as a method to explore many program executions at once. The main idea is to execute the program under test with symbolic placeholders that represent an arbitrary value instead of fixed inputs. Initially, all program variables store a unique symbol (typically, their own name). When we need to evaluate an expression that depends on some program variables, we replace the free variables with their current symbolic values and return the resulting expression instead.

Example 2.1.1.

```python
def incr_and_double(x : int):
  x = x + 1
  x = 2 * x
  return x
```

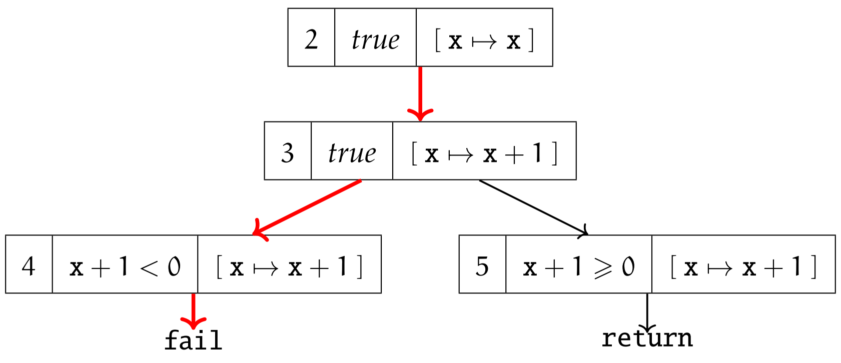
Symbolically executing the function `incr_and_double` returns the symbolic expression $2 * (x + 1)$. △

When a conditional instruction is encountered during the execution (for example, when executing an "if then else"), we cannot always determine the truth value of the condition as it may depends on the exact value of the inputs. Instead, we explore all possible ways to pursue the execution. For example, in the case of an "if then else", we execute the "then" branch under the assumption that the condition holds, and we also execute the "else" branch under the assumption that the condition does not hold. To remember which assumptions have been made through the execution, we store the conditions explicitly in the form of a boolean formula (often called *path condition*). As a result, symbolic execution produces a tree representing all possible step-by-step executions of a program. Each node of the tree is labeled with the condition that needs to be satisfied in order to reach the corresponding program state.

Example 2.1.2.

```python
def incr_and_test(x : int):
  x = x + 1
  if x < 0:
    fail()
  return
```

Symbolically executing the python function `fail_if_neg` generates the following symbolic execution tree. Each node is labeled with a line number, a path condition, and the current content of the variables. Initially, the input `x` is treated as the opaque symbol x.



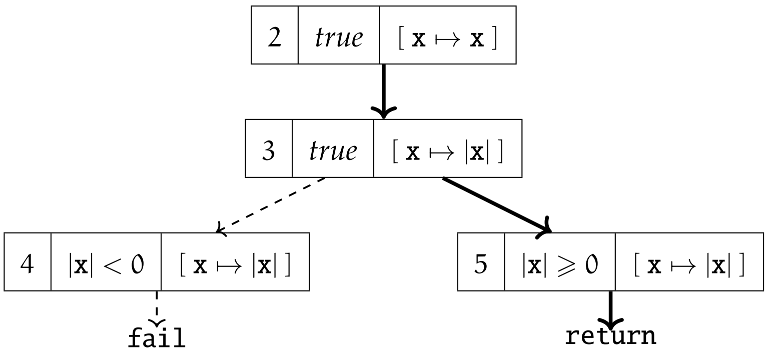
△

Symbolic execution gives a simple method to find bugs. It suffices to inspect the symbolic execution tree to see if there exists a branch leading to a failure state. To check that this branch of the symbolic execution tree is actually *feasible*, we additionally need to check that the corresponding path condition is satisfiable. For example, in example 2.1.2, a bug is reached when the condition $\mathtt{x} + 1 < 0$ is satisfied. This condition is satisfiable (for example, we can pick $\mathtt{x} = -2$) and therefore the program `incr_and_test` can crash. Testing the satisfiability of path conditions is necessary to avoid false alarms (reporting bugs that are not really bugs).

Example 2.1.3.

```
def abs_and_test(x : int):
  x = abs(x)
  if x < 0:
    fail()
  return
```

The symbolic execution tree of `abs_and_test` has a branch leading to a failure. However, its path condition is unsatisfiable which means the bug cannot be reached, regardless of the input.



△

To check the satisfiability of path conditions, we can rely on constraint solvers (for example Z3, VeriT, or CVC4 [DB08; Bou+09; Bar+11]). We can then derive a systematic algorithm to find bugs by symbolic execution.

```
L ← leaves(symbolic_execution_tree(P))
for all state ∈ L do
    if is_error(state) and state.path is satisfiable then
        return P has a bug
    end if
end for
```

```
return P has no bug
```

Assuming that we have a sound and complete constraint solver, and the symbolic execution is correctly implemented (we will discuss what this means formally later), this bug finding algorithm is precise and exhaustive. **Precise** means that it detects only real runtime-errors, **exhaustive** means it can detect all of them. However, in practice, this bug-finding algorithm cannot be implemented as it is. First of all, a solver can fail to decide the satisfiability of a constraint. This can happen if the solver is incomplete (i.e., it does not know how to solve certain constraints), or because not enough resources (time and memory) are available. Furthermore, even state of the art constraint solvers can produce inconsistent results [WZS20] which could lead to a bug miss (for example, if the solver mistakenly treats a satisfiable constraint labeling an error state as unsatisfiable) or to a spurious bug report (if the solver treats an unsatisfiable constraint labeling an error state as satisfiable).

Another major limitation is that symbolically executing an arbitrary program may never terminate. In particular, when programs contains unbounded loops, the symbolic execution tree can have infinite branches. As a consequence, only a subset of the reachable program states can be computed in finite time.

Example 2.1.4. The following program with an unbounded while loop generates an infinite symbolic execution tree:

```python
while True: x = x + 1
```

$$\begin{tikzcd} \boxed{1 \mid \mathit{true} \mid \mathtt{x} := \mathtt{x}} \arrow[r] & \boxed{1 \mid \mathit{true} \mid \mathtt{x} := \mathtt{x}+1} \arrow[r] & \boxed{1 \mid \mathit{true} \mid \mathtt{x} := \mathtt{x}+2} \arrow[r] & (\ldots) \end{tikzcd}$$

△

Due to potential non-termination, it is *a priori* impossible to cover all reachable states of any arbitrary program in finite time. In practice, we can only compute a subset of the state space which means our algorithm do detect bugs is not exhaustive. Nonetheless, if the symbolic execution tree is expanded *lazily*, it is possible to make the algorithm *relatively exhaustive*: it will enumerate all potential bugs if enough time and resources are allocated. We propose such a lazy implementation of this algorithm in Coq in chapter 4.

Finally, to be correct, a symbolic interpreter should strictly follow the same *execution model* that is used to run programs in production. To demonstrate why this is important, we take a simple program as an example. Below are two snippets of code for a function `suspicous` in Python and Java.

```java
public void suspicious(int x) {
  if (x >= -5 && x / 3 == -2) {
    fail();
  }
}
```

```python
def suspicious(x : int):
  if x >= -5 and x // 3 == -2:
    fail()
```

At first sight, both functions are equivalent. However, according to the semantics of Python, the function has a bug (it crashes if executed with input $\texttt{x} = -5$). On the other hand, in Java, there is no input that causes the function `suspicious` to crash. This is due to different interpretations of integer division in these programming languages. If a bug finder misinterprets the semantics of an operator, it can lead to a bug miss (e.g., if we analyze the Python code with Java semantics) or to a false alarm (e.g., if we analyze the Java code with Python semantics). To avoid this problem, a reliable symbolic interpreter should be provably faithful to the reference semantics of the analyzed programming language. In this thesis, we use the Coq proof assistant to prove that a symbolic interpreter is correct with respect to the semantics of the programming language it interprets. From the correctness of the symbolic interpreter, we derive a bug finder and prove its precision and its exhaustiveness.

2.2 The Coq proof assistant

Through this thesis, we will use the Coq proof assistant as our main tool to write code, specifications and proofs. In this section, we give a brief introduction to Coq.

2.2.1 Functional programming

The Coq proof assistant [1] features a rich purely functional programming language with a syntax similar to OCaml[2]. Inductive data-types are defined using the `Inductive` command, and recursive functions using the `Fixpoint` command. As an example, we define a representation of natural numbers and equip it with an addition function.

```coq
Inductive Nat : Type :=
  | O
  | S (n : Nat).

Fixpoint add (n1 n2 : Nat) :=
  match n1 with
  | O => n2
  | S n1 => S (add n1 n2)
  end.
```

[1]https://coq.inria.fr
[2]https://v2.ocaml.org/index.html

Coq programs such as the add function can be executed in place using the Compute command.

```
Compute (add (S O) (S (S O))).
(* prints S (S (S O)) *)
```

Coq also includes a mechanism called *extraction* [Let08] to generate OCaml code equivalent to a Coq definition. Functions can be extracted using the Extraction command. For example, we can extract the add function with the command Extraction add. The result of the extraction is the following OCaml snippet:

```
type nat =
  | O
  | S of nat

let rec add n1 n2 =
  match n with
  | O -> n2
  | S n1 -> S (add n1 n2)
```

2.2.2 Logic programming

To define predicates over data-types, a possibility is to use boolean functions. For example, one could define a function is_even : Nat -> bool that checks if a natural number is even. However, this would limit us to the expression of predicates that are *decidable* (i.e. for which there exists a function that decides, in finite time, whether the predicate holds for a given argument). We can also define potentially non-decidable predicates using custom inference-rules. For example, the predicate is_even can be defined with the following rules.

$$\frac{}{\texttt{is\_even}\ 0} \qquad \frac{\texttt{is\_even}\ n}{\texttt{is\_even}\ (n+2)}$$

In Coq, relations defined by a set of inference rules are expressed using *inductive predicates*. As an example, the predicate is_even is defined as follows.

```
Inductive is_even : Nat -> Prop :=
  | is_even_O : is_even O
  | is_even_2_plus_n : is_even n -> is_even (S (S n)).
```

Instead of returning a boolean, an inductive predicate defines a family of *propositions* (Prop in Coq). To prove a proposition such as is_even (S (S (S (S O)) (4 is even), the only way is to use the inference rules is_even_O and is_even_2_plus_n. A possible proof is is_even_2_plus_n (is_even_2_plus_n (is_even_O)).

2.2.3 Interactive proofs

Apart from defining functional programs and inductive predicates, one can also state theorems and develop their proofs in Coq. For example, it is clear that 0 is neutral for the addition over natural numbers. This simple statement can be formally stated and proved by providing a *proof script*. A proof script is a sequence of commands called *tactics* guiding the assistant through the proof of the desired result. For example, the proof that `add 0 n = n` is straightforward: it suffices to unfold the definition of `add 0 n` to obtain a trivial equality `n = n`. This proof can be done using the tactics `simpl` (to simplify the expression) and `reflexivity` (to prove the remaining trivial equality).

```
Theorem add_zero_left :
  forall (n : Nat), add 0 n = n.
Proof.
  simpl. reflexivity.
Qed.
```

The proof that `add n 0 = n` requires a little more work. Indeed, this result does not immediately follow from the definition of the `add` function. To prove this result, one needs to reason by induction on `n`.

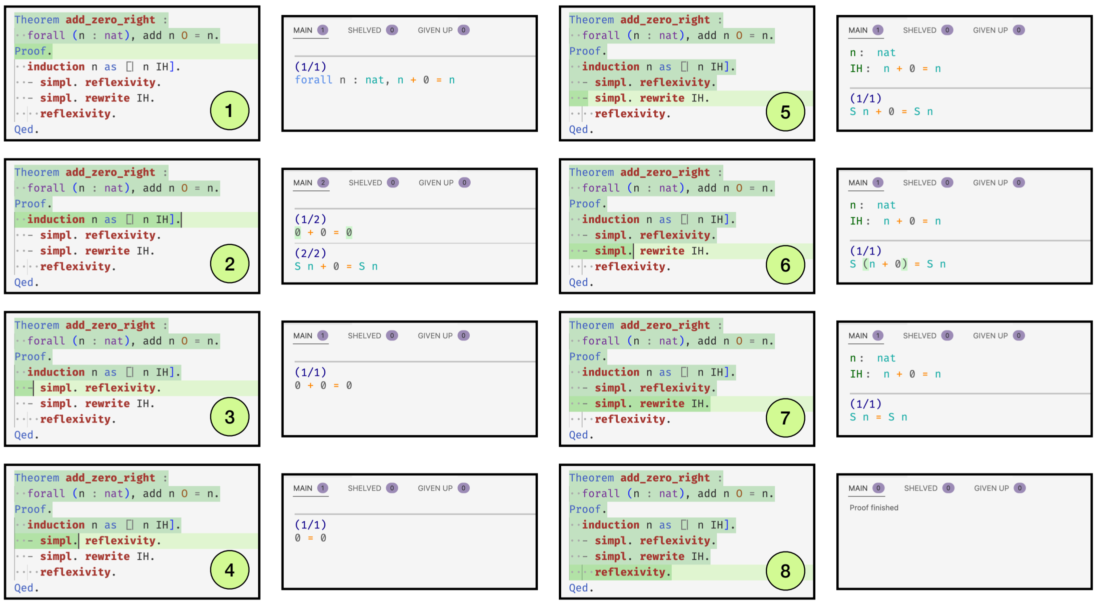
```
Theorem add_zero_right :
  forall (n : Nat), add n 0 = n.
Proof.
  induction n as [| n IH ].
  - simpl. reflexivity
  - simpl. rewrite IH.
    reflexivity.
Qed.
```

Longer proof scripts like the one above are never written blindly. Instead, they are built interactively in a step-by-step fashion. After each tactic issued by the user, the proof assistant display the current *goals* that remains to be proved together with the available hypothesis. The complete sequence of interactions for the proof of theorem `add_zero_right` is pictured in figure 2.1.

Providing every details of such a simple proof seems a little bit overwhelming. Fortunately, such proofs can most of the time be *automated*. There exists many tactics for automation. One of the most simple and general one is `auto`. For example, the two cases in the proof of `add_zero_right` are simple enough to be resolved automatically by `auto`.

```
Theorem add_zero_right :
  forall (n : Nat), add n 0 = n.
Proof.
```

```
  induction n; auto.
Qed.
```



```
Theorem add_zero_right :
  forall (n : nat), add n O = n.
Proof.
  induction n as [| n IH].
  - simpl. reflexivity.
  - simpl. rewrite IH.
    reflexivity.
Qed.
```

```
MAIN 1   SHELVED 0   GIVEN UP 0
(1/1)
forall n : nat, n + 0 = n
```

```
MAIN 2   SHELVED 0   GIVEN UP 0
(1/2)
0 + 0 = 0
(2/2)
S n + 0 = S n
```

```
MAIN 1   SHELVED 0   GIVEN UP 0
(1/1)
0 + 0 = 0
```

```
MAIN 1   SHELVED 0   GIVEN UP 0
(1/1)
0 = 0
```

```
MAIN 1   SHELVED 0   GIVEN UP 0
n :  nat
IH :  n + 0 = n
(1/1)
S n + 0 = S n
```

```
MAIN 1   SHELVED 0   GIVEN UP 0
n :  nat
IH :  n + 0 = n
(1/1)
S (n + 0) = S n
```

```
MAIN 1   SHELVED 0   GIVEN UP 0
n :  nat
IH :  n + 0 = n
(1/1)
S n = S n
```

```
MAIN 0   SHELVED 0   GIVEN UP 0
Proof finished
```

Figure 2.1: Step by step interactive proof of a simple theorem

2.2.4 Coinduction and lazy computations

As discussed in the previous section, we will have to model potentially infinite tree-shaped data-structures and algorithms over them. Coq provides a mechanism called coinduction to reason with such data-structures, and a specific type of recursive functions called cofixpoints to traverse and generate infinite objects.

coinductive definitions are, in appearance, similar to inductive ones. For example, we can define potentially infinite lists (also called streams) coinductively as follows:

```
CoInductive stream (A : Type) :=
  | snil
  | scons (x : A) (xs : stream A).
```

At first sight, this definition looks exactly the same as the one of (inductive) lists. The key difference is in the interpretation of this type. Contrary to inductive types, the elements of a coinductive types are exactly these that can be obtained by finite **or infinite** applications of the constructors. In other word, the type of streams contains all finite lists, but also the infinite lists obtained by infinite application of the scons constructor. For example, the infinite sequence $1, 2, 3, \ldots$ can be modeled using the infinite stream `scons 0 (scons 1 (scons 2 ...))`. To define such an infinite stream, we intuitively need recursive definitions. However, in this context, Coq fixpoints will not be enough. Indeed, `Fixpoint` definitions are required to be structurally recursive on one of their arguments to ensure termination. Such a requirement does not make much sense if we try to produce infinite data-structures with a recursive function. Instead, we use `CoFixpoint` definitions that can perform arbitrary recursive calls under the condition that all calls are guarded by a constructor. For example, the stream of all natural numbers starting from n can be defined as follows:

```
CoFixpoint naturals (n : nat) : stream nat :=
  scons n (naturals (S n)).
```

This recursive definition is accepted by Coq even though the recursive call is performed on a structurally increasing argument. The recursive call `(naturals (S n))` is below a constructor `scons` which fulfills the *guardedness criterion*. This strict discipline ensures that infinite objects defined by cofixpoints can be evaluated *lazily*: it suffices to evaluate the definition up to the next constructor. The process can be iterated to progressively unfold the definition in a step-by-step fashion. The guardedness criterion ensures that there is always a "next constructor" to punctuate the lazy evaluation of recursive definitions.

To consume a coinductive stream, we can use a regular fixpoint. For example, we can define a function `get` to access the n-th element of a stream. When the n-th element is not defined, the function returns `None`. In the remainder of this thesis, we will note $s[n]$ for `get s n`.

```
Fixpoint get {A} (s : stream A) (n : nat) : option A :=
  match n, s with
  | 0, scons x _ => Some x
  | S n, scons _ s => get n s
  | _, _ => None
  end.
```

In Coq, the laziness of coinductive definitions is implicit. In OCaml, coinductive definitions are translated to data-structures with explicit suspensions to introduce laziness. Suspensions are implemented using the module `Lazy.t` of the OCaml standard library[3]. For example, the definition of coinductive streams is translated to the following OCaml code:

```
type 'a stream = 'a __stream Lazy.t
and 'a __stream =
  | Snil
  | Scons of 'a * 'a stream
```

[3]https://v2.ocaml.org/api/Lazy.html

Cofixpoints are converted to standard recursive functions with all constructors explicitly guarded by the `lazy` keyword to delay recursive computations. For example, the definition of the stream `naturals` is translated to the following OCaml code:

```
let rec naturals n =
  lazy (Scons (n, (naturals (S n))))
```

Functions that are consuming streams also need to explicitly *force* the delayed computations using the operator `Lazy.force`. For example, the function `get` is translated to the following OCaml code:

```
let rec get s n =
  match n, Lazy.force s with
  | O, Scons (x, _) -> Some x
  | S n, Scons (_, xs) -> get s n
  | _, _ -> None
```

Chapter 3

Formal foundations of symbolic execution

To reason formally about bug-finding, we need to fix a target programming language and properly understand its execution model. In this chapter, we formalize a simple programming language (we call it BUG) and its formal semantic. Then, we define a symbolic semantics [BB19] to formally describe what it means to symbolically execute BUG programs. Finally, we establish a formal connection between the symbolic semantics and the concrete semantics. This connection will be the basis to justify the correctness of a bug finder based on symbolic execution.

3.1 BUG: a target programming language

3.1.1 Syntax

The language we focus on is a simple imperative programming languages with support loops, conditionals, and integer arithmetic. This core language is sufficient to demonstrate the techniques developed in the thesis. The syntax of the language is described in figure 3.1. Note that we add an instruction `fail` that can interrupt the execution of a program with an error. This instruction can be used to instrument the code with assertions.

Definition 3.1 (Syntax of BUG)

$$
\begin{array}{rrl}
\textit{Variables} & \mathrm{var} \in & \mathbb{V} \\
\textit{Arithmetic} & \mathrm{aexpr} ::= & c \in \mathbb{Z} \mid \mathrm{var} \\
 & \mid & \mathrm{aexpr}\ (\ \texttt{+} \mid \texttt{-}\ )\ \mathrm{aexpr} \\
\textit{Boolean} & \mathrm{bexpr} ::= & \texttt{true} \mid \texttt{false} \\
 & \mid & \mathrm{aexpr}\ (\ \texttt{<} \mid \texttt{=}\ )\ \mathrm{aexpr} \\
 & \mid & \mathrm{bexpr}\ (\ \texttt{and} \mid \texttt{or}\ )\ \mathrm{bexpr} \\
 & \mid & \texttt{not}\ \mathrm{bexpr} \\
\textit{Statements} & \mathrm{stmt} ::= & \texttt{skip} \\
 & \mid & \texttt{fail} \\
 & \mid & \mathrm{var}\ \texttt{=}\ \mathrm{expr} \\
 & \mid & \mathrm{stmt}\ \texttt{;}\ \mathrm{stmt} \\
 & \mid & \texttt{if}\ \mathrm{bexpr}\ \texttt{then}\ \mathrm{stmt}_1\ \texttt{else}\ \mathrm{stmt}_2 \\
 & \mid & \texttt{while}\ \mathrm{bexpr}\ \texttt{do}\ \mathrm{stmt}
\end{array}
$$

3.1.2 Semantics

To be able to specify the correctness of a bug finder, we first need to formalize the execution model of our target programming language. Programs are executed in a memory M assigning integer values to every variable. A program state is a pair of a memory and a statement to execute.

$$
\begin{array}{rl}
\textit{Memories} & \mathbb{M} = \mathbb{V} \to \mathbb{Z} \\
\textit{States} & \mathbb{S} = \mathbb{M} \times \mathrm{stmt}
\end{array}
$$

The evaluation of boolean and arithmetic expressions is formalized in a *denotational* style. We write $[\![e]\!]_M$ to denote the result of evaluating expression e in a memory M.

Definition 3.2 (Denotational semantics of expressions)

$$
\begin{array}{c|c}
[\![x]\!]_M := M(x),\ x \in \mathbb{V} & [\![e_1 \texttt{ < } e_2]\!]_M := [\![e_1]\!]_M <_{\text{bool}} [\![e_2]\!] \\
[\![z]\!]_M := z,\ z \in \mathbb{Z} & [\![e_1 \texttt{ = } e_2]\!]_M := [\![e_1]\!]_M =_{\text{bool}} [\![e_2]\!]_M \\
[\![e_1 \texttt{ + } e_2]\!]_M := [\![e_1]\!]_M + [\![e_2]\!]_M & [\![b_1 \texttt{ and } b_2]\!]_M := [\![b_1]\!]_M \wedge_{\text{bool}} [\![b_2]\!]_M \\
[\![e_1 \texttt{ - } e_2]\!]_M := [\![e_1]\!]_M - [\![e_2]\!]_M & [\![b_1 \texttt{ or } b_2]\!]_M := [\![b_1]\!]_M \vee_{\text{bool}} [\![b_2]\!]_M \\
 & [\![\texttt{not } b]\!]_M := \neg_{\text{bool}} [\![b]\!]_M
\end{array}
$$

Remark To make a clear distinction between symbolic expressions and their mathematical value, we use color codes. Symbols are represented in brown and using a mono font while mathematical values are noted in black using the usual math font. For example, $(1 + 2) \in \mathbb{Z}$ denotes the actual value 3 but $(1 \texttt{ + } 3) \in$ *aexpr* denotes the abstract syntax tree `Add (Cst 1) (Cst 2)`. We use the notations $<_{\text{bool}}, \wedge_{\text{bool}}, \vee_{\text{bool}}, \neg_{\text{bool}}$ to denote the boolean comparisons and boolean operators.

The semantics of statements is given in a *small-step* style as a transition relation between program states. Given two states s_1 and s_2, we note $s_1 \hookrightarrow s_2$ to express that continuing execution from state s_1 leads to successor state s_2. For a memory $M \in \mathbb{M}$, a variable $x \in \mathbb{V}$ and a constant $c \in \mathbb{Z}$, we note $M[x \leftarrow c]$ the memory M where x has been updated to contain c.

Definition 3.3 (Operational semantics of BUG)

ASSIGN
$$\frac{}{\langle M, x \texttt{ = } e\rangle \hookrightarrow \langle M[x \leftarrow [\![e]\!]_M], \texttt{skip}\rangle}$$

SEQ-SKIP
$$\frac{}{\langle M, \texttt{skip ; } s\rangle \hookrightarrow \langle M, s\rangle}$$

SEQ-STEP
$$\frac{\langle M, s_1\rangle \hookrightarrow \langle M, s_2\rangle}{\langle M, s_1 \texttt{ ; } s_3\rangle \hookrightarrow \langle M, s_2 \texttt{ ; } s_3\rangle}$$

IF-TRUE
$$\frac{[\![b]\!]_M = \texttt{true}}{\langle M, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2\rangle \hookrightarrow \langle M, s_1\rangle}$$

IF-FALSE
$$\frac{[\![b]\!]_M = \texttt{false}}{\langle M, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2\rangle \hookrightarrow \langle M, s_2\rangle}$$

WHILE-TRUE
$$\frac{[\![b]\!]_M = \texttt{true}}{\langle M, \texttt{while } b \texttt{ do } s\rangle \hookrightarrow \langle M, s \texttt{ ; while } b \texttt{ do } s\rangle}$$

WHILE-FALSE
$$\frac{[\![b]\!]_M = \texttt{false}}{\langle M, \texttt{while } b \texttt{ do } s\rangle \hookrightarrow \langle M, \texttt{skip}\rangle}$$

We note $s_1 \hookrightarrow^* s_2$ to say that an execution can go from s_1 to s_2 in zero, one, or more steps. For convenience, we also define the set of reachable states of a program p as follows.

Definition 3.4 (Reachable state)

$\mathit{Reach}(p) := \{s \mid \exists M, \langle M, p\rangle \hookrightarrow^* s\}$

3.1.3 Runtime errors in BUG

A runtime error is any situation in which the execution of a program "is stuck". Using our semantics, we can make this intuition more formal. A *stuck state* is any execution state that cannot make progress according to the semantics. We note that the instruction `skip` has no successor (there is nothing to be done when executing `skip`) but it should not be considered as an error. This leads to the following formal definition of *stuckness*:

Definition 3.5

A state $\langle M, p\rangle \in \mathbb{S}$ is stuck (we note $\mathit{Stuck}\ \langle M, p\rangle$) if $p \neq$ `skip` and there exists no state s such that $\langle M, p\rangle \hookrightarrow s$.

From this definition, we say that a program has a bug if there exists an execution leading to a stuck state.

Definition 3.6

A program p is said to have a bug if there exists an a state $s \in \mathit{Reach}(p)$ such that $\mathit{Stuck}\ s$.

This formal definition makes it very clear that, to find a bug, it suffices to expose an initial memory configuration that leads the program to failure. Our final goal is to fully automate this process using symbolic execution. It is important to notice that stuck states can be identified *syntactically*. Indeed, the only stuck states are these where the next instruction to be executed is `fail`.

Theorem 1 (Syntactic criterion for stuckness). *The state $\langle M, p\rangle$ is stuck if and only if p is of the form* `fail ;` $\ldots$

Proof. If p is of the form `fail ; ...`, the state $\langle M, p\rangle$ is clearly stuck. It remains to prove the left-to-right implication. We proceed by induction on p.

Case of `fail` Trivial as `fail` is already of the form `fail ; ...`.

Cases of `skip`, `if`, `while`, or sequences prefixed with `skip`: These cases are trivial since these statements are never stuck (they always have a $\hookrightarrow$-successor).

Case of a sequence not prefixed with `skip`: Suppose $p = p_1$ `;` p_2 and $\langle M, p\rangle$ is stuck. By induction hypothesis, we know that if p_1 is stuck, it is of the form `fail` or `fail ; ...`. Furthermore, p_1 is necessarily stuck. Indeed, if p_1 had a successor, then p_1 `;` p_2 would also have a successor. Consequently, p_1 is of the form `fail ; ...` and so is p_1 `;` p_2. □

As simple as it may seem, theorem 1 plays an important role because it provides an effective way to detect runtime errors during the step-by-step execution of a program.

3.1.4 Formalizing the semantics in Coq

In the previous subsections, we presented the syntax and the semantics of a simple programming language. In this subsection, we briefly present the Coq formalization of this language. First, the abstract syntax of arithmetic and boolean expressions is expressed by two inductive types `aexpr` and `bexpr`.

```
Inductive aexpr : Type :=
  | Var (x : string)
  | Cst (c : Z)
  | Add (e1 e2 : aexpr)
  | Sub (e1 e2 : aexpr).
```

```
Inductive bexpr : Type :=
  | Eq (e1 e2 : aexpr)
  | Lt (e1 e2 : aexpr)
  | Bool (b : bool)
  | And (b1 b2 : bexpr)
  | Or (b1 b2 : bexpr)
  | Neg (b : bexpr).
```

Similarly, the abstract syntax of instructions is modeled by a type `stmt` of program statements.

```
Inductive stmt :=
  | Skip
  | Fail
  | Assign (x : string) (e : aexpr)
  | Seq (p1 p2 : stmt)
  | Ite (b : bexpr) (p1 p2 : stmt)
  | While (b : bexpr) (p : stmt).
```

Expressions and programs are interpreted with respect to a memory mapping variable names to integer values so we define a type `store` = `string` → `Z` to represent memories. The denotation of expressions is implemented by two recursive functions `aeval` and `beval` to evaluate arithmetic and boolean expressions respectively. To evaluate arithmetic and boolean operators, we use their implementation provided in the Coq standard library.

```
Fixpoint aeval (env : store) (e : aexpr) :=
  match e with
  | Var x => env x
  | Cst c => c
  | Add e1 e2 => aeval env e1 + aeval e2
  | Sub e1 e2 => aeval env e1 - aeval e2
  end.

Fixpoint beval (env : store) (b : bexpr) :=
  match e with
  | Eq e1 e2 => aeval env e1 =? aeval env e2
  | Lt e1 e2 => aeval env e1 <? aeval env e2
  | Bool b => b
  | And b1 b2 => beval env b1 && aeval b2
  | Or b1 b2 => beval env b1 || beval env b2
  | Neg b => beval env b
end.
```

The inference rules of the small-step semantics are defined using an inductive predicate `step` : `state` → `state` → `Prop` where *state* is the type of program states (pairs of a memory and a program statement). Each constructor of the predicate corresponds to a transition rule. Below is an excerpt of the actual Coq definition.

```
Definition state : Type := store * stmt.

Inductive step : state -> state -> Prop :=
  | step_Assign env x e :
    step (env, Assign x e) (update env x (aeval env e), Skip)

  | step_Seq_Skip env p :
    step (env, Seq Skip p) (env, p)

  | step_Seq_step env1 env2 p1 p2 p3:
    step (env1, p1) (env2, p2) ->
    step (env1, Seq p1 p3) (env2, Seq p2 p3)
  | ... .
```

The function `update` used for the rule `step_Assign` is simply updating the content memory of a given variable in a memory. It can be defined in a functional style as follows:

```
Definition update (env : store) (x : string) (vx : Z) : store :=
  fun y => if x =? y then vx else env y.
```

3.2 Symbolic execution of BUG

Similar to how the concrete execution model can be formalized by a concrete semantics, symbolic execution can also be described by means of a *symbolic* semantics: a semantics that describes how to symbolically execute a program. This idea is not novel and has been investigated in multiple papers [Fra+20; BB19; Por+22]. In this section, we propose to formalize a symbolic semantics for the language BUG in the Coq proof assistant. This will allow us to formally prove a connection between the concrete and the symbolic semantics.

3.2.1 Symbolic semantics

In symbolic execution, programs are executed in a symbolic memory assigning arithmetic expressions to program variables. Symbolic execution states are triples of a path-condition (expressed as a boolean expression), a symbolic memory, and a statement to execute.

$$\begin{aligned} \textit{Symbolic memories} \qquad & \mathbb{M}^{\mathrm{sym}} = \mathit{var} \to \mathit{aexpr} \\ \textit{Symbolic states} \qquad & \mathbb{S}^{\mathrm{sym}} = \mathbb{M}^{\mathrm{sym}} \times \mathrm{bexpr} \times \mathrm{stmt} \end{aligned}$$

The "rules" of symbolic execution are also described in the form of a small-step operational semantics. Given two symbolic states $\hat{s}_1$ and $\hat{s}_2$, we note $\hat{s}_1 \hookrightarrow_{\mathrm{sym}} \hat{s}_2$ to express that it is possible to take one symbolic execution step from $\hat{s}_1$ to $\hat{s}_2$. To define the symbolic semantics of the assignment instruction, we also need to define a notion of *symbolic evaluation* of an expression in a symbolic memory. We use the notation $[\![e]\!]_{\hat{M}}$ for the evaluation of e in a symbolic memory $\hat{M}$.

Definition 3.7 (Symbolic Semantics of BUG expressions)

$$
\begin{aligned}
[\![x]\!]^{sym}_{M} &:= M(x),\ x \in \mathbb{V} \\
[\![z]\!]^{sym}_{M} &:= z,\ z \in \mathbb{Z} \\
[\![e_1 + e_2]\!]^{sym}_{M} &:= [\![e_1]\!]^{sym}_{M} + [\![e_2]\!]^{sym}_{M} \\
[\![e_1 - e_2]\!]^{sym}_{M} &:= [\![e_1]\!]^{sym}_{M} - [\![e_2]\!]^{sym}_{M}
\end{aligned}
$$

$$
\begin{aligned}
[\![b]\!]^{sym}_{M} &:= b, b \in \{\texttt{true}, \texttt{false}\} \\
[\![e_1 < e_2]\!]^{sym}_{M} &:= [\![e_1]\!]^{sym}_{M} < [\![e_2]\!] \\
[\![e_1 = e_2]\!]^{sym}_{M} &:= [\![e_1]\!]^{sym}_{M} = [\![e_2]\!]^{sym}_{M} \\
[\![b_1 \texttt{ and } b_2]\!]^{sym}_{M} &:= [\![b_1]\!]^{sym}_{M} \texttt{ and } [\![b_2]\!]^{sym}_{M} \\
[\![b_1 \texttt{ or } b_2]\!]^{sym}_{M} &:= [\![b_1]\!]^{sym}_{M} \texttt{ or } [\![b_2]\!]^{sym}_{M} \\
[\![\texttt{not } b]\!]^{sym}_{M} &:= \texttt{not } [\![b]\!]^{sym}_{M}
\end{aligned}
$$

Symbolic evaluation essentially replaces free variables with their assigned expression in the current symbolic memory, thus producing a bigger expression. As for the concrete semantics, we implement the symbolic semantics of expressions in Coq using two recursive functions `sym_aeval` and `sym_beval`. Symbolic memories are represented by the type `sym_store` : `string` → `aexpr`.

```
Fixpoint sym_aeval (env : store) (e : aexpr) :=
  match e with
  | Var x => env x
  | Cst _ => e
  | Add e1 e2 => Add (sym_aeval env e1) (sym_aeval e2)
  | Sub e1 e2 => Sub (sym_aeval env e1) (sym_aeval e2)
  end.

Fixpoint sym_beval (env : sym_store) (b : bexpr) :=
  match e with
  | Eq e1 e2 => Eq (sym_beval env e1) (sym_beval env e2)
  | Lt e1 e2 => Lt (sym_beval env e1) (sym_beval env e2)
  | Bool _ => e
  | And b1 b2 => And (sym_beval env b1) (sym_beval b2)
  | Or b1 b2 => Or (sym_beval env b1) (sym_beval env b2)
  | Neg b => Neg (sym_beval env b)
end.
```

The transition rules from symbolic states to symbolic states closely resembles the rules of the concrete semantics. We simply replace all operations on concrete memories by operations on symbolic memories and collect constraints when branching instructions are executed. However, contrary to the concrete semantics, the symbolic semantics is *non-deterministic*: when a branching instruction is reached, multiple successor states are possible. For example, when executing an if-then-else statement, the symbolic execution can either continue with the *then* branch and assume that the condition holds, or pursue with the *else* branch and assume that the condition is violated. In both cases, the current path condition is extended with the corresponding assumption.

Definition 3.8 (Symbolic Operational Semantics of BUG)

SYM-ASSIGN

$$\frac{}{\langle M, \varphi, x = e\rangle \hookrightarrow_{\text{sym}} \langle M[x \leftarrow [\![e]\!]^{\text{sym}}_M], \varphi, \texttt{skip}\rangle}$$

SYM-SEQ-SKIP

$$\frac{}{\langle M, \varphi, \texttt{skip ; } s\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi, s\rangle}$$

SYM-SEQ-STEP

$$\frac{\langle M, \varphi_1, s_1\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi_2, s_2\rangle}{\langle M, \varphi_1, s_1 \texttt{ ; } s_3\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi_2, s_2 \texttt{ ; } s_3\rangle}$$

SYM-IF-TRUE

$$\frac{}{\langle M, \varphi, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi \texttt{ and } [\![b]\!]^{\text{sym}}_M, s_1\rangle}$$

SYM-IF-FALSE

$$\frac{}{\langle M, \varphi, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi \texttt{ and } [\![\texttt{not } b]\!]^{\text{sym}}_M, s_2\rangle}$$

SYM-WHILE-TRUE

$$\frac{}{\langle M, \varphi, \texttt{while } b \texttt{ do } s\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi \texttt{ and } [\![b]\!]^{\text{sym}}_M, s \texttt{ ; while } b \texttt{ do } s\rangle}$$

SYM-WHILE-FALSE

$$\frac{}{\langle M, \varphi, \texttt{while } b \texttt{ do } s\rangle \hookrightarrow_{\text{sym}} \langle M, \varphi \texttt{ and } [\![\texttt{not } b]\!]^{\text{sym}}_M, \texttt{skip}\rangle}$$

To symbolically execute a program p, we generally start the execution with the symbolic state $\langle \texttt{true}, x \mapsto x, p\rangle$ where the path condition is `true` (we make no assumption on the inputs), and the symbolic memory maps every program variable to its own name (initially, the inputs are undetermined and are treated as opaque symbols).

3.2.2 Relating concrete and symbolic states

Intuitively, a sequence of symbolic execution steps is nothing but a sequence of concrete execution steps where the value of the input variables are left undetermined. To recover a concrete execution from a symbolic one, we just need to fix concrete initial values for all undetermined variables. As an example we consider the symbolic execution of the simple program $p = $ `if x > 0 then x := x - 1 else x := x + 1`. One possible symbolic sequence of steps follows the *then* branch:

$$\langle \texttt{true}, [\texttt{x} \mapsto x], p\rangle \xhookrightarrow{\text{sym}} \langle x > 0, [\texttt{x} \mapsto x], \texttt{x} = \texttt{x} + 1\rangle \xhookrightarrow{\text{sym}} \langle x > 0, [\texttt{x} \mapsto x + 1], \texttt{skip}\rangle$$

If we ignore the path conditions and we replace x with 1 in this sequence of symbolic execution steps, we obtain the following valid concrete execution

$$\langle [\texttt{x} \mapsto 1], p\rangle \hookrightarrow \langle [\texttt{x} \mapsto 1], \texttt{x} = \texttt{x} + 1\rangle \hookrightarrow \langle [\texttt{x} \mapsto 2], \texttt{skip}\rangle$$

This concrete execution corresponds exactly to the execution of p with initial memory $[\texttt{x} \mapsto 1]$. Note that picking $x = 0$ to *concretize* would not lead to a valid execution according to the concrete semantics. Indeed, the execution depicted above follows a branch where the initial value of the variable `x` is supposed to be strictly greater than 0. To concretize a symbolic execution with initial values for the program variables, we also need to make sure that the choice of values is consistent with the path conditions.

To capture the notion of *concretization* of symbolic states more formally, we start by introducing a composition operator $\circ : \mathbb{M} \times \mathbb{M}^{\text{sym}} \to \mathbb{M}$. Given any concrete memory $M \in \mathbb{M}$ representing a choice of initial values, and a symbolic memory $\hat{M} \in \mathbb{M}^{\text{sym}}$, the concrete memory $M \circ \hat{M}$ is obtained from $\hat{M}$ by replacing free variables with their associated values in M.

Definition 3.9 (Composition)

$M \circ \hat{M} \triangleq x \mapsto [\![\hat{M}(x)]\!]_M$

Using the composition operator, we can define what it means for a concrete state to be a valid concretization of a symbolic state. We introduce a relation $s \sqsubseteq_{M_0} \hat{s}$ (read "concrete state s concretizes the symbolic state $\hat{s}$ with choice of initial values M_0" or "symbolic state $\hat{s}$ symbolizes the concrete state s through initial values M_0") to denote that the memory M_0 is a consistent choice of values to concretize the symbolic state $\hat{s}$ into s. In particular, we make sure that the initial memory M_0 satisfies the path condition.

Definition 3.10 (Conretization relation)

$$\frac{[\![\varphi]\!]_{M_0} = \texttt{true}}{\langle M_0 \circ \hat{M}, p\rangle \sqsubseteq_{M_0} \langle \hat{M}, \varphi, p\rangle}$$

The concretization relation precisely describes how to interpret a sequence of symbolic states as a sequence of concrete ones. Going back to our example, we have the following valid relations:

$$\begin{tikzcd}
\langle \texttt{true}, [\texttt{x} \mapsto x], \texttt{p} \rangle \arrow[r, hook, "\text{sym}"] \arrow[d, dotted, "\sqsupseteq_{[x \mapsto 1]}"] & \langle x > 0, [\texttt{x} \mapsto x], \texttt{x = x + 1} \rangle \arrow[r, hook, "\text{sym}"] \arrow[d, dotted, "\sqsupseteq_{[x \mapsto 1]}"] & \langle x > 0, [\texttt{x} \mapsto x + 1], \texttt{skip} \rangle \arrow[d, dotted, "\sqsupseteq_{[x \mapsto 1]}"] \\
\langle [\texttt{x} \mapsto 1], \texttt{p} \rangle \arrow[r, hook] & \langle [\texttt{x} \mapsto 1], \texttt{x = x + 1} \rangle \arrow[r, hook] & \langle [\texttt{x} \mapsto 2], \texttt{skip} \rangle
\end{tikzcd}$$

Using the concretization relation, we can naturally define which concrete states are *virtually* reached during symbolic execution. It suffices to consider the concretization of all reachable symbolic states. We note $\mathit{Reach}_{\text{sym}}(\texttt{p})$ the set of concrete states that are explored by symbolically executing the program $\texttt{p}$.

Definition 3.11

$\mathit{Reach}_{\text{sym}}(\texttt{p}) = \{s \mid \exists M \exists \hat{s}, \langle \texttt{true}, x \mapsto x, \texttt{p} \rangle \hookrightarrow^{*}_{\text{sym}} \hat{s} \wedge s \sqsubseteq_M \hat{s}\}$

In the following sections, we show that the states reached by concrete and symbolic execution coincide. In other words we have, $\mathit{Reach}_{\text{sym}}(\texttt{p}) = \mathit{Reach}(\texttt{p})$. The fact that the symbolic semantics explores only states that are concretely reachable is referred to as **soundness** with respect to the concrete semantics. Dually, the fact that all states that are concretely reachable can be reached by the symbolic semantics is referred to as **completeness**. Soundness and completeness of the symbolic semantics will play a central role in proving the precision and the exhaustiveness of a bug finder.

3.2.3 Relating concrete and symbolic evaluation

In order to prove soundness and completeness of the symbolic operational semantics, we first need to establish a relation between symbolic and concrete evaluation of expressions. In particular, we show that evaluating (concretely) an expression in a composed memory $M \circ \hat{M}$, is the same as first evaluating symbolically in $\hat{M}$ and then evaluate the resulting expression concretely in M.

Theorem 2 (Composition-Evaluation)**.** *For any boolean or arithmetic expression* e*, memory* M *and symbolic memory* $\hat{M}$ *we have* $[\![e]\!]_{M \circ \hat{M}} = [\![[\![e]\!]^{\mathit{sym}}_{\hat{M}}]\!]_M$

Proof. We need to cover the cases where e is an arithmetic expression or a boolean expression. In both cases, the proof goes by induction on the structure of e. We cover only the arithmetic case. The the boolean case follows the exact same principles.

Case of a constant:
Suppose $c \in \mathbb{Z}$, by definitions we have $[\![c]\!]_{M \circ \hat{M}} = c = [\![[\![c]\!]^{sym}_{\hat{M}}]\!]_M$.

Case of a variable:
Suppose $x \in \mathbb{V}$, by definitions we have $[\![x]\!]_{M \circ \hat{M}} = [\![\hat{M}(x)]\!]_M = [\![[\![x]\!]^{sym}_{\hat{M}}]\!]_M$.

Case of a binary operator:
Suppose $\diamond \in \{+, -\}$, by definition we have $[\![e_1 \diamond e_2]\!]_{M \circ \hat{M}} = [\![e_1]\!]_{M \circ \hat{M}} \diamond [\![e_2]\!]_{M \circ \hat{M}}$.
Also, $[\![[\![e_1 \diamond e_2]\!]^{sym}_{\hat{M}}]\!]_M = [\![[\![e_1]\!]^{sym}_{\hat{M}} \diamond [\![e_2]\!]^{sym}_{\hat{M}}]\!]_M = [\![[\![e_1]\!]^{sym}_{\hat{M}}]\!]_M \diamond [\![[\![e_2]\!]^{sym}_{\hat{M}}]\!]_M$.
By induction hypothesis, we have $[\![[\![e_i]\!]^{sym}_{\hat{M}}]\!]_M = [\![e_i]\!]_{M \circ \hat{M}}$ and therefore $[\![e_1 \diamond e_2]\!]_{M \circ \hat{M}} = [\![[\![e_1 \diamond e_2]\!]^{sym}_{\hat{M}}]\!]_M$. □

In our language BUG, we evaluate expressions (symbolically or concretely) in two situations: when we perform a variable assignment, or when we check boolean conditions. Theorem 2 already establish a connection between symbolic and concrete evaluation of boolean expression. We can also relate concrete and symbolic variable assignments. Given a variable x, an expression e, and a concrete memory M, we can perform the concrete variable assignment to update M into $M[x \leftarrow [\![e]\!]_M]$. Similarly, given a symbolic memory $\hat{M}$, we can do a symbolic assignment to update $\hat{M}$ into $\hat{M}[x \leftarrow [\![e]\!]^{sym}_{\hat{M}}]$. Building on theorem 2, we can prove that performing a concrete assignment in a composed memory $M \circ \hat{M}$ is the same as first performing a symbolic assignment in $\hat{M}$, and then composing.

Theorem 3 (Composition-Updates). *For any expression* e*, memory* M *and symbolic memory* $\hat{M}$ *we have* $M \circ (\hat{M}[x := [\![e]\!]^{sym}_{\hat{M}}]) = (M \circ \hat{M})[x := [\![e]\!]_{M \circ \hat{M}}]$

Proof. Let $M_{left} = M \circ (\hat{M}[x := [\![e]\!]^{sym}_{\hat{M}}])$ and $M_{right} = (M \circ \hat{M})[x := [\![e]\!]_{M \circ \hat{M}}]$. We need to prove that for all y, $M_{left}(y) = M_{right}(y)$. We consider two cases:

If $x = y$
On one hand, $M_{left}(y) = M_{left}(x) = [\![[\![e]\!]^{sym}_{\hat{M}}]\!]_M$.
On the other hand, $M_{right}(y) = M_{right}(x) = [\![e]\!]_{M \circ \hat{M}}$.
By theorem 2, $[\![[\![e]\!]^{sym}_{\hat{M}}]\!]_M = (M \circ \hat{M})[x := [\![e]\!]_{M \circ \hat{M}}]$ end hence, $M_{left}(y) = M_{right}(y)$.

If $x \neq y$
In that case, we have $M_{left}(y) = (M \circ \hat{M})(y) = M_{right}(y)$. □

3.2.4 Soundness of the symbolic semantics

The soundness theorem we want to prove simply states that every state (virtually) reached by symbolic execution is also reached by concrete execution. Formally, we prove that $\mathrm{Reach}_{\mathrm{sym}}(p) \subseteq \mathrm{Reach}(p)$. This ensures that the symbolic semantics is exploring only real executions of programs and will be the key ingredient to prove the *precision* of a bug-finder.

To establish the soundness of the symbolic semantics, we use a simulation diagram technique. We start by proving that one step of symbolic execution is simulating one step of concrete execution. More precisely, we prove any symbolic execution step can be concretized by replacing free variables with values that satisfy the path condition generated during the symbolic execution. Then, we show that this result is preserved over multiple symbolic execution steps.

First, we prove that one step of symbolic execution is simulating one step of concrete execution in the sense of the following theorem:

Lemma 4 (Soundness over one step)**.**

$$\frac{\langle \varphi_1, \hat{M}_1, p_1\rangle \hookrightarrow_{sym} \langle \varphi_2, \hat{M}_2, p_2\rangle \qquad [\![\varphi_2]\!]_M = \mathit{true}}{\langle M \circ \hat{M}_1, p_1\rangle \hookrightarrow \langle M \circ \hat{M}, p_2\rangle}$$

Proof. By induction on the derivation $\langle \varphi_1, \hat{M}_1, p_1\rangle \hookrightarrow_{\mathrm{sym}} \langle \varphi_2, \hat{M}_2, p_2\rangle$. Each case of the induction corresponds to a rule of the symbolic semantics. We prove each case by applying the corresponding rule of the concrete semantics.

Case of an assignment:
We have $\langle \varphi, \hat{M}, x = e\rangle \hookrightarrow_{\mathrm{sym}} \langle \varphi, \hat{M}[x := [\![e]\!]^{\mathrm{sym}}_{M}], \texttt{skip}\rangle$ and $[\![\varphi]\!]_M = \texttt{true}$.
We need to show $\langle M \circ \hat{M}, x = e\rangle \hookrightarrow \langle M \circ (\hat{M}[x := [\![e]\!]^{\mathrm{sym}}_{M}]), \texttt{skip}\rangle$.
By lemma 3, it is enough to show $\langle M \circ \hat{M}, x = e\rangle \hookrightarrow \langle (M \circ \hat{M})(x := [\![e]\!]^{\mathrm{sym}}_{M\circ\hat{M}}), \texttt{skip}\rangle$.
This exactly matches with the concrete semantic rule for assignment.

Case of a sequence prefixed with skip:
We have $\langle \varphi, \hat{M}, \texttt{skip ; } p\rangle \hookrightarrow_{\mathrm{sym}} \langle \varphi, \hat{M}, p\rangle$ and $[\![M]\!]_{\varphi} = \texttt{true}$. We need to show $\langle M \circ \hat{M}, \texttt{skip ; } p\rangle \hookrightarrow \langle M \circ \hat{M}, p\rangle$. This exactly matches with the concrete semantic rule for sequences prefixed by `skip`.

Case of a sequence not prefixed with skip:
We have $\langle \varphi_1, \hat{M}_1, p_1 \texttt{ ; } p_3\rangle \hookrightarrow_{\mathrm{sym}} \langle \varphi_2, \hat{M}_2, p_2 \texttt{ ; } p_3\rangle$ and $\langle \varphi_1, \hat{M}_1, p_1\rangle \hookrightarrow_{\mathrm{sym}} \langle \varphi_2, \hat{M}_2, p_2\rangle$ and $[\![\varphi_2]\!]_M = \texttt{true}$.
By induction hypothesis we have $\langle M \circ \hat{M}, p_1\rangle \hookrightarrow \langle M \circ \hat{M}, p_2\rangle$.

Using the semantic rule for the sequence, we can conclude $\langle M \circ \hat{M}, p_1 \texttt{ ; } p_3\rangle \hookrightarrow \langle M \circ \hat{M}, p_2 \texttt{ ;} p_3\rangle$.

Case of an "if" when the condition is supposed to be satisfied:
We have $\langle \varphi, \hat{M}, \texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2\rangle \hookrightarrow_{\text{sym}} \langle \varphi \texttt{ and } [\![c]\!]^{\text{sym}}_{\hat{M}}, \hat{M}, p_1\rangle$
and $[\![\varphi \texttt{ and } [\![c]\!]^{\text{sym}}_{\hat{M}}]\!]_M = \texttt{true}$.
We need to show that $\langle M \circ \hat{M}, \texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2\rangle \hookrightarrow \langle M \circ \hat{M}, p_1\rangle$.
Using the first semantic rule for conditionals, it suffices to show that $[\![c]\!]_{M \circ \hat{M}} = \texttt{true}$. Since $[\![\varphi \texttt{ and } [\![c]\!]^{\text{sym}}_{\hat{M}}]\!]_M = \texttt{true}$, in particular $[\![[\![c]\!]^{\text{sym}}_{\hat{M}}]\!]_M = \texttt{true}$. From lemma 2 it immediately follows that $[\![c]\!]_{M \circ \hat{M}} = \texttt{true}$.

Case of an "if" when the condition is supposed to be violated:
We have $\langle \varphi, \hat{M}, \texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2\rangle \hookrightarrow_{\text{sym}} \langle \varphi \texttt{ and } [\![c]\!]^{\text{sym}}_{\hat{M}}, \hat{M}, p_2\rangle$
and $[\![\varphi \texttt{ and not } [\![c]\!]^{\text{sym}}_{\hat{M}}]\!]_M = \texttt{true}$.
We need to show that $\langle M \circ \hat{M}, \texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2\rangle \hookrightarrow \langle M \circ \hat{M}, p_2\rangle$.
Using the second semantic rule for conditionals, it suffices to show that $[\![c]\!]_{M \circ \hat{M}} = \texttt{false}$. Since $[\![\varphi \texttt{ and not } [\![c]\!]^{\text{sym}}_{\hat{M}}]\!]_M = \texttt{true}$, in particular $[\![[\![c]\!]^{\text{sym}}_{\hat{M}}]\!]_M = \texttt{false}$.
From lemma 2 it immediately follows that $[\![c]\!]_{M \circ \hat{M}} = \texttt{false}$.

Cases for loops:
The cases for loops are handled exactly like if-statements. □

It remains to prove that theorem 4 remains true over multiple symbolic execution step. To do this proof, we start by observing that the path conditions can only *grow* during the symbolic execution (i.e., one step of symbolic execution can only add new restrictions on the program variables). We refer to this observation as monotonicity of path constraints.

Lemma 5 (Monotonicity of path constraints). *Suppose* $\langle \varphi_1, \hat{M}_1, p_1\rangle \hookrightarrow_{sym} \langle \varphi_2, \hat{M}_2, p_2\rangle$. *If* $[\![\varphi_2]\!]_M = true$ *for some* M*, then also* $[\![\varphi_1]\!]_M = true$*.*

Proof. It suffices to look at all possible symbolic execution rules. The only rules that modify the path constraints are rules for conditionals and loops. In both cases, the current path constraint is only extended with a new conjunct. □

Building on lemma 5, we can now prove that multiple steps of symbolic execution simulate multiple steps of concrete execution.

Lemma 6 (Soundness over multiple steps).

$$\frac{\langle \varphi_1, \hat{M}_1, p_1\rangle \hookrightarrow^*_{sym} \langle \varphi_2, \hat{M}_2, p_2\rangle \qquad [\![\varphi_2]\!]_M = \texttt{true}}{\langle M \circ \hat{M}_1, p_1\rangle \hookrightarrow^* \langle M \circ \hat{M}, p_2\rangle}$$

Proof. The proof goes by induction on the number n of symbolic execution steps.

Base case (0 steps):
If $\langle \varphi_1, \hat{M}_1, p_1 \rangle \hookrightarrow^*_{\text{sym}} \langle \varphi_2, \hat{M}_2, p_2 \rangle$ in zero steps, then $p_1 = p_2$. By reflexivity, we always have $\langle M \circ \hat{M}, p_1 \rangle \hookrightarrow^* \langle M \circ \hat{M}, p_1 \rangle$.

Inductive case ($n + 1$ steps):
Suppose $\langle \varphi_1, \hat{M}_1, p_1 \rangle \hookrightarrow^*_{\text{sym}} \langle \varphi_2, \hat{M}_2, p_2 \rangle \hookrightarrow_{\text{sym}} \langle \varphi_3, \hat{M}_3, p_3 \rangle$ and $[\![\varphi_3]\!]_M = \texttt{true}$.
We have to show $\langle M \circ \hat{M}_1, p_1 \rangle \hookrightarrow^* \langle M \circ \hat{M}_3, p_3 \rangle$.
By induction, we have $[\![\varphi_2]\!] = \texttt{true} \implies \langle M \circ \hat{M}_1, p_1 \rangle \hookrightarrow^* \langle M \circ \hat{M}_2, p_2 \rangle$.

Since $\langle \varphi_2, \hat{M}_2, p_2 \rangle \hookrightarrow_{\text{sym}} \langle \varphi_3, \hat{M}_3, p_3 \rangle$ and $[\![\varphi_3]\!]_M = \texttt{true}$, by monotonicity (5) we have $[\![\varphi_2]\!] = \texttt{true}$ and therefore, $\langle M \circ \hat{M}_1, p_1 \rangle \hookrightarrow^* \langle M \circ \hat{M}_2, p_2 \rangle$. By soundness over one step (4) we also have $\langle M \circ \hat{M}_2, p_2 \rangle \hookrightarrow \langle M \circ \hat{M}_3, p_3 \rangle$. By transitivity, it follows $\langle M \circ \hat{M}_1, p_1 \rangle \hookrightarrow^* \langle M \circ \hat{M}_3, p_3 \rangle$. □

The intuition between theorem 6 is better pictured by drawing a simulation diagram. If we consider a symbolic execution from state $\hat{s}_1$ to state $\hat{s}_n$, and a memory M satisfying the path constraint of $\hat{s}_n$, we have the following relations (where concrete states s_i are the concretization with respect to M of symbolic states $\hat{s}_i$):

$$\begin{tikzcd}
\hat{s}_1 \arrow[r, hook, "\text{sym}"] \arrow[d, dotted, "\exists_M"] & \hat{s}_2 \arrow[r, hook, "\text{sym}"] \arrow[d, dotted, "\exists_M"] & \dots \arrow[r, hook, "\text{sym}"] & \hat{s}_n \arrow[d, "\exists_M"] \\
s_2 \arrow[r, hook, dotted] & s_2 \arrow[r, hook, dotted] & \dots \arrow[r, hook, dotted] & s_n
\end{tikzcd}$$

In such a simulation diagram, solid edges represent relations that we initially suppose to hold, and dotted edges represent relations that we can deduce. As a consequence of theorem 6, we can prove that symbolic reachability subsumes concrete reachability.

Theorem 7 (Soundness for reachability)**.** *For all* p, $\mathit{Reach}_{\mathit{sym}}(p) \subseteq \mathit{Reach}(p)$

Proof. Suppose $s \in \mathit{Reach}_{\text{sym}}(p)$. By definition, there exists a symbolic state $\hat{s}$ and a memory M, such that $\langle \texttt{true}, x \mapsto x, p \rangle \hookrightarrow^*_{\text{sym}} \hat{s}$ and $s \sqsubseteq_M \hat{s}$. By theorem 6 it follows $\langle M \circ (x \mapsto x), p_1 \rangle = \langle M, p_1 \rangle \hookrightarrow s$. Therefore $s \in \mathit{Reach}(p)$. □

It is important to notice that the inclusion $\mathit{Reach}_{\text{sym}}(p) \subseteq \mathit{Reach}(p)$ only imply that the **concretization** of any reachable symbolic state is concretely reachable. There exists a valid concretization of a symbolic state only if its path condition is satisfiable. Nonetheless, if the satisfiability of paths-conditions is checked *a posteriori*, the soundness of the symbolic semantics ensures that only true bugs can be detected by symbolic execution.

This fact follows immediately from the soundness theorem and can be formalized as follows.

Corollary 8. *For any program* p*, if a symbolic state of the form* $\langle \varphi, \hat{M}, \texttt{fail ; } \ldots \rangle$ *is in* $Reach_{sym}(p)$ *and* φ *is satisfiable, then* p *has a bug.*

3.2.5 Completeness

While soundness guarantees that a symbolic execution simulates a set of concrete ones, we would also like to have a completeness results ensuring that *all* concrete executions can be simulated. Our goal is to show that for every sequence of concrete execution steps, there exists a *similar* sequence of symbolic steps. Formally, we demand that for any sequence of concrete states $s_1 \hookrightarrow s_2 \ldots \hookrightarrow s_n$, there should exist a sequence of symbolic states $\hat{s}_1 \hookrightarrow_{\text{sym}} \hat{s}_2 \ldots \hookrightarrow_{\text{sym}} \hat{s}_n$ and an initial memory M such that $s_i \sqsubseteq_M \hat{s}_i$ for all $1 \leqslant i \leqslant n$.

Again, we use a *simulation diagram* technique. We start by showing that for every concrete transition $s_1 \hookrightarrow s_2$, if s_1 concretizes $\hat{s}_1$ via a memory M, then we can always take a symbolic execution step from $\hat{s}_1$ to some symbolic state $\hat{s}_2$ while preserving the concretization relation.

Lemma 9 (Completeness over one step)**.**

$$\frac{s_1 \sqsubseteq_M \hat{s}_1 \wedge s_1 \hookrightarrow s_2}{\exists \hat{s}_2, s_2 \sqsubseteq_M \hat{s}_2 \wedge \hat{s}_1 \hookrightarrow_{sym} \hat{s}_2}$$

Proof. The proof goes by induction on the derivation $s_1 \hookrightarrow s_2$. Supposing $s_1 = \langle M_1, p_1 \rangle$, we consider all possible derivations.

Case of `skip` or `fail` : If $p_1 = \texttt{skip}$ or $p_1 = \texttt{fail}$, s_1 does not have a successor which contradicts the assumption $s_1 \hookrightarrow s_2$.

Case of an assignement: If $p_1 = x = e$, then $s_2 = \langle M_1[x \leftarrow [\![e]\!]_{M_1}], \texttt{skip} \rangle$. Now, $\hat{s}_1$ is necessarily of the form $\langle \varphi_1, \hat{M}_1, p_1 \rangle$ for some φ_1 and $\hat{M}_1$. We choose $\hat{s}_2 = \langle \varphi_1, \hat{M}_1[x \leftarrow [\![e]\!]^{\text{sym}}_{\hat{M}_1}], \texttt{skip} \rangle$.
Clearly, we have $\hat{s}_1 \hookrightarrow_{\text{sym}} \hat{s}_2$. It remains to prove that $s_2 \sqsubseteq_M \hat{s}_2$. By definition, it suffices to show that $[\![\varphi_1]\!]_M = \mathit{true}$ and $M_1[x \leftarrow [\![e]\!]] = M \circ (\hat{M}_1[x \leftarrow [\![e]\!]^{\text{sym}}_{\hat{M}_1}])$.
From our assumptions, we already know that $s_1 \sqsubseteq \hat{s}_1$ so in particular $[\![\varphi_1]\!]_M = \mathit{true}$, and $M_1 = M \circ \hat{M}_1$. We just need to prove that $M_1[x \leftarrow [\![e]\!]] = M \circ (\hat{M}_1[x \leftarrow [\![e]\!]^{\text{sym}}_{\hat{M}_1}])$. This equality is established by first replacing M_1 with its equivalent

formulation as a function of $\hat{M}_1$ and then applying lemma 2:

$$M_1[x \leftarrow [\![e]\!]_{M_1}] = (M \circ \hat{M}_1)[x \leftarrow [\![e]\!]_{M \circ M_1}] = M \circ (\hat{M}_1[x \leftarrow [\![e]\!]^{\text{sym}}_{\hat{M}_1}])$$

Case of an "if" when the condition is supposed to be satisfied:
Supposing $p_1 = \texttt{if}\ c\ \texttt{then}\ p_2\ \texttt{else}\ p_3$, $[\![c]\!]_{M_1} = \mathit{true}$, we have $s_2 = \langle M_1, p_2 \rangle$. Further, $\hat{s}_1$ is of the form $\langle \varphi_1, \hat{M}_1, p_1 \rangle$. We choose $\hat{s}_2 = \langle \varphi_1\ \texttt{and}\ [\![c]\!]^{\text{sym}}_{\hat{M}_1}, \hat{M}_1, p_2 \rangle$. Clearly, $\hat{s}_1 \hookrightarrow_{\text{sym}} \hat{s}_2$. It remains to prove that $s_2 \sqsubseteq_M \hat{s}_2$. It is enough to show that $[\![\varphi_1\ \texttt{and}\ [\![c]\!]^{\text{sym}}_{\hat{M}_1}]\!]_M = \mathit{true}$.

$$[\![\varphi_1\ \texttt{and}\ [\![c]\!]^{\text{sym}}_{\hat{M}_1}]\!]_M = [\![\varphi_1]\!]_M \wedge_{\text{bool}} [\![[\![c]\!]_{\hat{M}_1}]\!]_M = [\![\varphi_1]\!]_M \wedge_{\text{bool}} [\![c]\!]_{M \circ \hat{M}_1}$$

Since $s_1 \sqsubseteq_M \hat{s}_1$, we have $[\![\varphi_1]\!]_M = \mathit{true}$ and $M_1 = M \circ \hat{M}_1$. It follows that $[\![c]\!]_{M \circ \hat{M}_1} = [\![c]\!]_M = \mathit{true}$.

Other cases The other cases follow exactly the same principles as the case for "if" when the condition is supposed to be satisfied. $\square$

We then prove that the concretization relation can also be preserved over multiple concrete steps by *iterating* the diagram established in lemma 9.

Lemma 10 (Completeness over multiple steps)**.**

$$\frac{s_1 \sqsubseteq_M \hat{s}_1 \wedge s_1 \hookrightarrow^* s_2}{\exists \hat{s}_2, s_2 \sqsubseteq_M \hat{s}_2 \wedge \hat{s}_1 \hookrightarrow^*_{sym} \hat{s}_2}$$

Proof. The proof goes by induction on the number of concrete steps.

Base case (0 steps): Suppose $s_1 = s_2$. Then it suffices to pick $\hat{s}_2 := \hat{s}_1$ as $\hat{s}_1 \hookrightarrow^*_{\text{sym}} \hat{s}_1$.

Inductive case ($n+1$ steps): Suppose there exists a state s_3 such that $s_1 \hookrightarrow^* s_3 \hookrightarrow s_2$ and $s_1 \sqsubseteq_M \hat{s}_1$. By induction hypothesis, there exists a symbolic state $\hat{s}_3 \sqsupseteq_M s_3$ such that $\hat{s}_1 \hookrightarrow^*_{\text{sym}} \hat{s}_3$. Applying lemma 9 starting from $\hat{s}_3$ we obtain a symbolic state $\hat{s}_2 \sqsupseteq_M s_2$ such that $\hat{s}_3 \hookrightarrow_{\text{sym}} \hat{s}_2$. We obtained an appropriate symbolic execution $\hat{s}_1 \hookrightarrow^*_{\text{sym}} \hat{s}_3 \hookrightarrow_{\text{sym}} \hat{s}_2$ $\square$

Similar to theorem 6, the completeness theorem over multiple steps can be better visualized using the following simulation diagram:

$$\begin{tikzcd}
s_1 \arrow[r, hook] \arrow[d, "\sqsubseteq_M"] & s_2 \arrow[r, hook] \arrow[d, dotted, "\sqsubseteq_M"] & \dots \arrow[r, hook] & s_n \arrow[d, dotted, "\sqsubseteq_M"] \\
\hat{s}_1 \arrow[r, hook, dotted, "\text{sym}"] & \hat{s}_2 \arrow[r, hook, dotted, "\text{sym}"] & \dots \arrow[r, hook, dotted, "\text{sym}"] & \hat{s}_n
\end{tikzcd}$$

Given a concrete execution $s_1 \hookrightarrow \dots \hookleftarrow s_n$ and knowing that $s_1 \sqsubseteq_M \hat{s}_1$, we can always reconstruct an *equivalent* symbolic execution $\hat{s}_1 \hookrightarrow_{\text{sym}} \dots \hookrightarrow \hat{s}_n$. From this fact, we can easily derive that all concretely reachable states are also symbolically reachable.

Theorem 11 (Completeness for reachability)**.** *For any program* p*,* $\mathit{Reach}(p) \subseteq \mathit{Reach}_{\mathit{sym}}(p)$

Proof. Let $s \in \mathit{Reach}(p)$. By definition, there exists a memory M such that $\langle M, p \rangle \hookrightarrow^* s$. Further, it is clear that $\langle M, p \rangle \sqsubseteq_M \langle \texttt{true}, x \mapsto x, p \rangle$. Therefore, by theorem 10, we know that there exists a symbolic state $\hat{s}$ such that $\langle \texttt{true}, x \mapsto x, p \rangle \hookrightarrow^*_{\text{sym}} \hat{s}$ and $s \sqsubseteq_M \hat{s}$. By definition of symbolic reachability, this implies $s \in \mathit{Reach}_{\text{sym}}(p)$. □

Chapter 4

Verified Implementation of a Symbolic Interpreter

4.1 Challenges

So far, we formalized a symbolic execution model describing how to symbolically execute programs. We proved that this symbolic execution model can predict the concrete behavior of programs. However, to symbolic execute programs, we still need to implement the symbolic semantics in the form of an executable interpreter. In particular, we want to implement this interpreter in Coq and prove that it is sound and complete with respect to the symbolic semantics (and therefore, by transitivity, also with respect to the concrete semantics). This means that the interpreter should cover **only** valid executions of the program (soundness) but also **all** possible executions (completeness). In this section, we discuss some challenges that arise when implementing such a symbolic interpreter in Coq.

4.1.1 Loops VS termination

The first main challenge in implementing a verified symbolic interpreter is termination. As explained before, symbolically executing programs with loops is not guaranteed to terminate. When implementing symbolic execution engines in traditional programming languages, this is not really an issue. Symbolic execution engines are typically implemented using a collection of mutually recursive functions or a never-ending execution loop, and the execution is interrupted manually by users. However, in Coq, all functions are required to terminate. This requirement is enforced by a collection of syntactic constraints on recursive definitions. If these syntactic requirements are not met, it is also possible to provide an explicit proof of termination. Multiple tools and libraries such as PROGRAM [Soz] and EQUATION [Soz10] simplify the process of defining recursive functions with non-trivial termination proofs. Unfortunately, in the context of symbolic execution, these tools are not really useful as it is impossible to prove termination. Nonetheless, we show in section 4.2 that lazy-execution techniques can be used to overcome this problem and still be able to implement and verify a symbolic interpreter in Coq.

4.1.2 Non-determinism VS exhaustiveness

Another major obstacle in the implementation of a symbolic interpreter is non-determinism. Indeed, when a conditional instruction is reached, the execution can continue in two ways depending on whether or not the condition is assumed to hold. To be exhaustive, we need to make sure that the symbolic interpreter follows both branches. Because symbolic execution can be non-terminating, one cannot simply decide to first fully execute the first branch and then fully execute the second branch. If the execution of the first branch never ends, the second branch will never be explored and some behavior of the program might remain unexplored. A solution to this problem is to use a breadth-first execution strategy.

4.2 A coinductive symbolic interpreter

To answer the challenges with termination and exhaustiveness, we use a lazy approach. Our symbolic interpreter produces a lazy-evaluated, potentially endless stream of reachable symbolic states. We then prove that all states in the stream are indeed reachable according to the symbolic semantics. Further, we prove that all reachable states eventually appear in the stream.

To implement the symbolic interpreter, we start by defining an expansion function `expand` : $\mathbb{S}_{\text{sym}} \rightarrow$ `list` $\mathbb{S}_{\text{sym}}$ that computes the list of all possible successors of a symbolic state. The expansion function simply performs a case analysis on the next program statement that needs to be executed and compute the successors accordingly by following the rules of the symbolic semantics.

```
Fixpoint expand path mem prog :=
  match prog with
  | Skip => []
  | Error => []
  | Seq Skip p => [ (path, mem, prog) ]
  | Seq p1 p2 =>
    map
```

```
      (fun '(path, mem, p1') => (path, mem, Seq p1' p2))
      (expand path mem p1)
  | Ite c p1 p2 => [
      (And path (beval_sym mem c), mem, p1);
      (And path (Neg (beval_sym mem c)), mem, p2)
    ]
  | Loop c p => [
      (And path (beval_sym mem c), mem, Seq p (Loop c p));
      (And path (Neg (beval_sym mem c), mem, Skip)
    ]
  end.
```

It is worth noting that the expansion function needs to be recursive because of the sequence operator. Indeed, when a statement of the form $p_1 ; p_2$ is executed, we need to first recursively compute all direct successors p_1, and then extend each of them with the *continuation* $(. ; p_2)$. In Coq, recursive functions must be structurally decreasing in one of their arguments. This forces us to split the 3 components of the input symbolic state (path condition, symbolic memory, and program statement) as 3 separate parameters in order for the recursion to be structural on the program statement. As this is just an artefact of Coq's termination-checking mechanism, we will sometimes note `expand` s the application of `expand` to a triple $s \in \mathbb{S}_{\text{sym}}$ for convenience.

It is relatively straightforward to see that the expansion function computes exactly the $\hookrightarrow_{\text{sym}}$-successor of any symbolic state. We say that expand is a sound and complete functional implementation of the relation $\hookrightarrow_{\text{sym}}$.

Theorem 12 (Sound and complete expansion). *For all symbolic states* $s_1, s_2 \in \mathbb{S}_{sym}$, $s_2 \in$ `expand` s_1 *if and only if* $s_1 \hookrightarrow_{sym} s_2$.

Proof. Let $s_1 = \langle \varphi_1, \hat{M}_1, p_1 \rangle$. The proof goes by immediate induction on the structure of p_1. For each case, we observe that all (and only) matching rules of $\hookrightarrow_{\text{sym}}$ are considered. □

Starting from a state s, the function `expand` correctly computes its direct successors. To compute all symbolic states reachable from s, we need to repeatedly computes direct successors. However, this would generate a potentially infinite tree of symbolic states. To enumerate all the reachable symbolic states, we compute a traversal of the tree in the form of a infinite stream of states. To ensure that all nodes of the tree are explored, we choose to traverse it in breadth.

We define a cofixpoint `reachable` that takes a list l of symbolic states as an input, and use a breadth-first strategy to enumerate all states reachable from l.

```
CoFixpoint reachable l :=
  match l with
  | [] => snil
  | s::l => scons s (reachable (l ++ expand s))
  end.
```

Note that from one recursive call to the other, the input list is expanded on the right to prioritize visits to states that have been generated first. This expansion strategy realize a breadth-first traversal of the state space an plays a key role in the proof that `reachable` exhaustively enumerate all reachable states.

Theorem 13 (Soundness)**.** *The stream reachable l only contains states that are $\hookrightarrow_{sym}$-reachable from l.*

Proof. We prove that for any state s_2 occurring at some position i of the stream, there exists a state s_1 of l that reaches s_2:

$$\forall i, \forall s_2, \forall l, (\texttt{reachable}\ l)[i] = s_2 \implies \exists s_1 \in l, s_1 \hookrightarrow^*_{\text{sym}} s_2$$

The proof goes by induction on the position i of s_2 in the stream (note that we generalize over l).

Base case ($i = 0$): Suppose $(\texttt{reachable}\ l)[0] = s_2$. If l is empty, the stream is empty so there cannot be an element at position 0. If l is not-empty, the first element of the stream is the first element of l so in particular it is reachable from l.

Inductive case: Suppose $(\texttt{reachable}\ l)[i+1] = s_2$ for some i. Again, l cannot be empty so $l = s :: l'$ for some state s and some list l'. By definition $\texttt{reachable}\ (s :: l')[i+1] = \texttt{reachable}(l'\ \texttt{++}\ \texttt{expand}\ s)[i]$. By induction hypothesis which know there exists a state $s_1 \in l'\ \texttt{++}\ \texttt{expand}\ s$ such that $s_1 \hookrightarrow^*_{\text{sym}} s_2$. If $s_1 \in l'$ then it is also in l and therefore s_2 is reachable from l. If $s_1 \in \texttt{expand} s$, by 12 we know that $s \hookrightarrow_{\text{sym}} s_1$ and therefore $s \hookrightarrow^*_{\text{sym}} s_2$ which proves that s_2 is reachable from l. □

The proof of completeness is more technical. We would like to prove that for any state s reachable from l, s will eventually appear at some position in the stream `reachable` l. Formally, this means we need to *guess* the exact position of s. To ease the proof, we start by introducing a collection of auxiliary lemmas.

First of all, if we consider a state s in a list l, it is possible to predict exactly when it is going to be visited. More precisely, we can prove that the element at position i in the list l will occur at the i-th position of the stream `reachable` l.

Lemma 14. $(\mathtt{reachable}\ (l_1 +\!\!+ [s] +\!\!+ l_2)[\mathtt{length}\ l_1] = s$

Proof. By immediate induction on the list l_1. □

An immediate corollary of 14 is that every state in l occurs in $\mathtt{reachable}\ l$.

Corollary 15. *For all state $s \in l$, there exists a position i such that $(\mathtt{reachable}\ l)[i] = s$*

Using this fact, we prove that for every state in the stream, its direct successors are also in the stream.

Lemma 16. *Let s_1, s_2 such that s_1 occurs in $\mathtt{reachable}\ l$ and $s_1 \hookrightarrow_{sym} s_2$. Then s_2 also occurs in $\mathtt{reachable}\ l$.*

Proof. Formally, we prove the following stronger result:

$$\forall i, \forall l, \forall s_1, \forall s_2, ((\mathtt{reachable}\ l)[i] = s_1 \wedge s_1 \hookrightarrow_{\mathrm{sym}} s_2) \implies \exists j, (\mathtt{reachable}\ l)[j] = s_2$$

The proof is by induction on the position i of s_1 in the stream.
Base case $(i = 0)$: Suppose $(\mathtt{reachable}\ l)[0] = s_1$ and $s_1 \hookrightarrow_{\mathrm{sym}} s_2$. First, because s_1 is in the stream, l cannot be empty so $l = s :: l'$ for some l'. Second, by completeness of $\mathtt{expand}$, we know that $s_2 \in \mathtt{expand}\ s_1$. and therefore $\mathtt{expand}\ s_1 = l_1 +\!\!+ [s_2] +\!\!+ l_2$ for some lists l_1, l_2. We prove that s_2 occurs in the stream exactly at position $1 + \mathtt{length}\ l' + \mathtt{length}\ l_1$:

$$\begin{aligned}
& (\mathtt{reachable}\ l)[1 + \mathtt{length}\ l' + \mathtt{length}\ l_1] \\
= {} & (\mathtt{reachable}\ (s :: l'))[1 + \mathtt{length}\ l' + \mathtt{length}\ l_1] \\
= {} & (\mathtt{reachable}\ (l' +\!\!+ \mathtt{expand}\ s)[\mathtt{length}\ l' + \mathtt{length}\ l_1] \\
= {} & (\mathtt{reachable}\ (l' +\!\!+ (l_1 +\!\!+ [s_2] +\!\!+ l_2))[\mathtt{length}\ l' + \mathtt{length}\ l_1] \\
= {} & (\mathtt{reachable}\ ((l' +\!\!+ l_1) +\!\!+ [s_2] +\!\!+ l_2))[\mathtt{length}\ (l' +\!\!+ l_1)] = s_2
\end{aligned}$$

Inductive case: Suppose that $(\mathtt{reachable}\ l)[i+1] = s_1$ for some i, and $s_1 \hookrightarrow_{\mathrm{sym}} s_2$. l can again not be empty so $l = s :: l'$ and by definition

$$s_1 = (\mathtt{reachable}\ (s :: l'))[i + 1] = (\mathtt{reachable}\ (l' +\!\!+ \mathtt{expand}\ s))[i]$$

Then, by induction hypothesis, there exists a position j such that

$$s_2 = (\mathtt{reachable}\ (l' +\!\!+ \mathtt{expand}\ s))[j]$$

By definition of $\mathtt{reachable}$, we also have

$$(\mathtt{reachable}\ (l' +\!\!+ \mathtt{expand}\ s) = (\mathtt{reachable}\ (s :: l'))[j + 1]$$

It follows that $j + 1$ is the position of s_2 in $\mathtt{reachable}\ l$. □

With the help of these lemmas, we can easily prove that any state reachable from l is in the stream `reachable` l. The intuition of the proof is to iterate lemma 16 along a path from l to s_2.

Theorem 17 (Completeness)**.** *The stream* `reachable` l *contains all states that are* $\hookrightarrow_{sym}$*-reachable from* l*.*

Proof. Suppose that s_2 is reachable from l. By definition, there exists $s_1 \in l$ with $s_1 \hookrightarrow^*_{\text{sym}} s_2$. The proof goes by induction on the number of steps from s_1 to s_2.

Base case (0 steps): Suppose $s_1 = s_2$. Then $s_2 \in l$ and by corollary 15, it has to occur in `reachable` l.

Inductive case: Suppose $s_1 \hookrightarrow^*_{\text{sym}} s_3 \hookrightarrow_{\text{sym}} s_2$ for some s_3. We have to show that s_2 occurs at some point in the stream. By induction hypothesis, we know that s_3 eventually occurs. Since s_2 is a direct successor of s_3, lemma 16 ensures that s_2 also appears in the stream. □

4.3 Faithfulness to the Reference Semantics

The soundness and completeness of the symbolic semantics, combined with the soundness and completeness of the `reachable` function, gives us a way to precisely and exhaustively traverse the reachable states of a program. Let p be a program and s a reachable state. We have the following chain of equivalences:

$$\begin{tikzcd}
s \in \text{Reach}(p) \arrow[d, leftrightarrow, "\text{Soundness + Completeness of the symbolic semantics (7, 11)}"] \\
s \in \text{Reach}_{\text{sym}}(p) \arrow[d, leftrightarrow, "\text{Definition of symbolic reachability}"] \\
\exists \hat{s}, \exists M, \langle \texttt{true}, x \mapsto x, p \rangle \hookrightarrow^*_{\text{sym}} s \wedge s \sqsubseteq_M \hat{s} \arrow[d, leftrightarrow, "\text{Soundness + Completeness of } \texttt{reachable} \text{ (13, 17)}"] \\
\hat{s} \in \texttt{reachable}\ [\langle \texttt{true}, x \mapsto x, p \rangle] \wedge s \sqsubseteq_M \hat{s}
\end{tikzcd}$$

Reading this diagram from bottom to top, we know that each symbolic states generated by the `reachable` function corresponds to a set of concretely reachable state. This makes our symbolic interpreter a suitable basis for precise bug finding. Conversely, from top to bottom, this diagram proves that asking whether a concrete state is reachable can be reduced to an effective symbolic computation. This indicates that our symbolic interpreter is suitable for exhaustive bug finding.

Chapter 5

Reducing The State Space

5.1 Symbolic semantics with pruning

The symbolic semantics we introduced in section 3 can be used as a foundation to implement a precise and exhaustive bug finder: there is a bug in a program if an only if an erroneous symbolic state with a satisfiable path-condition is eventually discovered. However, so far, we performed the satisfiability check *a posteriori*. This allowed us to implement and prove the correctness of the symbolic interpreter without referring to a constraint solver. While this greatly strengthen the reliability of the symbolic interpreter (its correctness does not depend on the assumption that we have a access to correct solver), it can also lead to significative performance losses. Indeed, first generating all reachable symbolic states and then filtering these with unsatisfiable path-conditions introduces a lot of useless states that we could avoid exploring without sacrificing exhaustiveness.

For example, when executing the piece of code below with the symbolic semantics described in chapter 3, the "complicated code" section will be fully symbolically executed, even though it is clear that it can never be reached. If the "complicated code" contains a non-terminating loop, the symbolic execution will never terminate.

```
if false then
  ... complicated code ...
else
  skip
```

This issue could easily be avoided if we check satisfiability during the execution and immediately prune symbolic states with unsatisfiable path-conditions to avoid exploring their successors. To support pruning of unfeasible paths, we need to slightly modify our symbolic semantics. We propose a semantics $\hookrightarrow_{\text{sym}\star}$ that require every generated state to have a satisfiable path conditions. This new semantics is defined on top of of $\hookrightarrow_{\text{sym}}$ with a single inference rule.

Definition 5.1 (Symbolic operational semantics with pruning)

$$\frac{\varphi \text{ is satisfiable} \qquad \langle M, \varphi, p\rangle \hookrightarrow_{\text{sym}} \langle M', \varphi', p'\rangle}{\langle M, \varphi, p\rangle \hookrightarrow_{\text{sym}\star} \langle M', \varphi', p'\rangle}$$

By definition, every symbolic execution with pruning $\hat{s}_1 \hookrightarrow_{\text{sym}\star} \ldots \hookrightarrow_{\text{sym}\star} \hat{s}_2$ is also a symbolic execution without pruning $\hat{s}_1 \hookrightarrow_{\text{sym}} \ldots \hookrightarrow_{\text{sym}} \hat{s}_2$ (i.e., pruning only removes possible executions). As a consequence, the soundness of $\hookrightarrow_{\text{sym}}$ is preserved by $\hookrightarrow_{\text{sym}\star}$.

Theorem 18 (Soundness)**.** $\mathit{Reach}_{\mathit{sym}\star}(p) \subseteq \mathit{Reach}_{\mathit{sym}}(p) \subseteq \mathit{Reach}(p)$

Proof. By definition, every $\hookrightarrow_{\text{sym}\star}$ step is also a valid $\hookrightarrow_{\text{sym}}$ step. □

The completeness is a little less immediate as we need to prove that we did not rule out too many executions. Because the reachability set $\mathit{Reach}_{\text{sym}}$ consider only valid concretization of reachable symbolic states, ruling out symbolic states with unsatisfiable path conditions is not removing any concrete states from $\mathit{Reach}_{\text{sym}}$.

Theorem 19 (Completeness)**.** $\mathit{Reach}(p) \subseteq \mathit{Reach}_{\mathit{sym}\star}(p)$

Proof. The proof is exactly the same as 10. It suffices to see that whenever the path condition is extended to simulate the concrete execution of a guarded statement, the current concrete memory is a solution of the path condition. This fact was simply ignored in the previous completeness proof. □

It is interesting to see that, semantically, the two symbolic execution models $\hookrightarrow_{\text{sym}}$ and $\hookrightarrow_{\text{sym}\star}$ are equivalent: they simulate exactly the same sets of concrete executions. Nonetheless, $\hookrightarrow_{\text{sym}\star}$ is computationally more efficient: it encodes the same amount of information but generates less "noise". While this does not make much of a difference in theory, this has important consequences if one is to implement an efficient symbolic interpreter. However, this efficiency gain comes at a cost. Indeed, for each execution step, a constraint solver has to be called to decide the satisfiability of the current path conditions. Relying on constraint solvers can be expansive. Further, constraint solvers are not always conclusive and can fail to decide the satisfiability of a given path condition. In the next section, we discuss these issues in more details and propose a verified implementation of the symbolic semantics with pruning.

5.2 Implementing the interpreter with pruning

To implement the semantics with pruning, we build on the interpreter without pruning. We simply remove the unfeasible states produced by the expansion function by calling a constraint solver. Verifying a fully featured constraint solver with a proof assistant is a research topic in itself and goes far beyond the scope of this thesis [Arm+11]. Instead, we use the module system of Coq to parametrize the implementation of our bug finder by a solver. The bug finder itself is then a functor that takes a solver as an argument, and returns a bug-finding function with its proof of correctness.

To ensure that our symbolic interpreter is correct, we need to make assumptions on the correctness of the solver. One could for example make the assumption that the solver is sound and complete: for every formula, it decides in finite time whether it is satisfiable or not. Such an assumption is not realistic as, in practice, solvers can timeout and give up on a query. Instead, we drop completeness and only require the solver to be *sound*: when it produces a result, it correctly classifies the input formula as satisfiable or unsatisfiable. However, it is still allowed to not provide a conclusive answer. We formalize this intuition by providing an interface SOUND_SOLVER. A module implementing this interface should expose a function check_sat that takes a boolean expression and can return either SAT, UNSAT or TIMEOUT. Further, we require that check_sat returns SAT (resp. UNSAT) only if the formula is satisfiable (resp. unsatisfiable).

```
Inductive solver_result :=
  | SAT
  | UNSAT
  | TIMEOUT.

Module Type SOUND_SOLVER.
  Parameter check_sat : bexpr -> solver_result.
  Hypothesis check_sat_SAT:
    forall (f : bexpr), check_sat f = SAT -> sat f.
  Hypothesis check_sat_UNSAT:
    forall (f : bexpr), check_sat f = UNSAT -> unsat f.
End SOUND_SOLVER.
```

To develop a symbolic interpreter parametrized by an arbitrary solver respecting the SOUND_SOLVER interface, we use a functor MakeInterpreter. For any module Solver implementing the interface, MakeInterpreter(Solver) is a module exposing a correct implementation of the symbolic semantics with pruning.

```
Module MakeInterpreter(Solver : SOUND_SOLVER).
  (* source code of the symbolic interpreter *)
End MakeInterpreter.
```

The implementation of the interpreter itself builds on the implementation of the symbolic semantics without pruning. To compute all the successors of a symbolic state, we start by applying the `expand` function, and then we filter out the successors that have unsatisfiable path conditions. If the solver fails to determine whether a path condition is satisfiable or not, we conservatively keep the state. Otherwise, we risk to delete a successor with a satisfiable path condition and therefore loose the completeness of the symbolic interpreter. This filtering mechanism is implemented by the following `expand'` function. The function `filter` comes from Coq standard library and performs a list filtering: it keeps all elements of a list that satisfy a given boolean predicate.

```
Definition expand' s :=
  filter (fun '(path, _, _) =>
    match Solver.check_sat path with
    | UNSAT => false
    | SAT | TIMEOUT => true
    end
  ) (expand s).
```

As for the symbolic interpreter without pruning, we generate a stream of all reachable symbolic states by iterating the `expand'` function.

```
CoFixpoint reachable' l :=
  match l with
  | [] => snil
  | s::l => scons s (reachable' (l ++ expand' s))
  end.
```

5.3 Correctness of the interpreter with pruning

We can easily prove that the new interpreter computes all $\hookrightarrow_{\text{sym}\star}$-successors (i.e., it is complete w.r.t. $\hookrightarrow_{\text{sym}\star}$).

Theorem 20 (Complete expansion). *if* $s_1 \hookrightarrow_{sym\star} s_2$ *then* $s_2 \in$ `expand'` s_1.

Proof. Suppose $s_1 \hookrightarrow_{\text{sym}\star} s_2$. By definition s_2 has a satisfiable path condition and $s_1 \hookrightarrow_{\text{sym}} s_2$. By completeness of `expand` (12), $s_2 \in$ `expand` s_1. Since the solver is supposed to be sound, calling the solver on the satisfiable path condition of s_2 returns either `SAT` or `TIMEOUT`. In both cases, s_2 is preserved by the filtering. $\square$

Proving that the new interpreter computes **only** valid $\hookrightarrow_{\text{sym}\star}$-successors is however impossible (i.e., it is not sound w.r.t. $\hookrightarrow_{\text{sym}\star}$). Indeed, because of the potential incompleteness of solvers, some symbolic states with unsatisfiable path conditions might survive the filtering mechanism of the `expand'` function. However, $\hookrightarrow_{\text{sym}\star}$ forbids to generate states with unsatisfiable path conditions. Nonetheless, because `expand'` only removes states from these computed by `expand`, it is still sound with respect to $\hookrightarrow_{\text{sym}}$. We therefore introduce a slight dissymmetry in the correctness proof of the new interpreter: completeness is proven with respect to the ideal symbolic execution model with perfect pruning, while soundness is proven with respect to the semantics without any pruning. Both symbolic executions models are sound and complete with respect to the concrete semantics, so this dissymmetry is not a problem in practice.

Theorem 21 (Sound expansion). *if* $s_2 \in$ `expand'` s_1 *then* $s_1 \hookrightarrow_{sym} s_2$.

Proof. If $s_2 \in$ `expand'` s_1 then in particular, $s_2 \in$ `expand` s_1. The result follows by soundness of `expand` with respect to $\hookrightarrow_{\text{sym}}$ (12). ☐

The soundness and completeness proofs of the `reachable` functions are independent from the implementation details of the expansion function. The only assumption that is used in the soundness (resp. completeness) proof is that `expand` is a sound (resp. complete) implementation of $\hookrightarrow_{\text{sym}}$. By replacing the expansion function with `expand'`, we immediately obtain the following theorem:

Theorem 22 (Soundness and completeness of `reachable'`). *The stream* `reachable'` l *contains* <u>*all*</u> *states that are* $\hookrightarrow_{sym\star}$*-reachable from* l*, and* <u>*only*</u> *states that are* $\hookrightarrow_{sym}$*-reachable from* l*.*

As discussed, soundness (the <u>only</u> part of the statement) is proven with respect to the symbolic semantics **without** pruning. This means that in practice, some states with unsatisfiable path conditions could survive the pruning mechanism. How much can be pruned depends on the underlying constraint solver.

Chapter 6

Deriving a Verified Bug Finder

6.1 Turning the symbolic interpreter into a bug finder

In this section, we transform the verified symbolic interpreter with pruning (presented in chapter 5) into an automated bug finder. As the interpreter is parametrized by a solver, the bug finder itself is a implemented as functor `MakeBugFinder` that takes a solver supposed to be correct as an argument:

```
Module MakeBugFinder(Solver : SOUND_SOLVER).
  Module Interpreter := MakeInterpreter(Solver).
  ...
End MakeBugFinder.
```

To convert the symbolic interpreter into a bug finder, the first step is to be able to detect runtime errors during the symbolic execution. As shown in lemma 1, erroneous states can be identified just by looking at the structure of the next instruction to be executed. In particular, erroneous states are shown to be exactly these where the next instruction to execute is `fail`. This check can be performed by traversing the structure of a program statement recursively. The function `next_is_fail` takes a program statement `p` as an argument and checks whether the next instruction to execute in `p` is `fail`.

```
Fixpoint next_is_fail p :=
  match p with
  | Fail => true
  | Seq p _ => next_is_fail p
  | _ => false
  end.
```

The function `next_is_fail` can naturally be applied to a symbolic state $\hat{s} = \langle \varphi, \hat{M}, p \rangle$ by calling it on the component p. Clearly, a symbolic state $\hat{s}$ such that `next_is_fail` $\hat{s} = \mathit{true}$ represents a set of erroneous concrete states. This intuition is summarized in the following theorem.

Theorem 23 (Symbolic detection of runtime errors)**.** *Let $\hat{s}$ be a symbolic state, and s a concrete state such that $s \sqsubseteq_M \hat{s}$. Then,* `next_is_fail` *$\hat{s} = \mathit{true}$ if and only if s is stuck.*

Proof. Let $\hat{s} = \langle \varphi, \hat{M}, p \rangle$. Since $s \sqsubseteq_M \hat{s}$, by definition we have $s = \langle M \circ \hat{M}, p \rangle$. By theorem 1, s is stuck if and only if p is of the form `fail ; ...` which is exactly what `next_is_fail` $\hat{s}$ = `next_is_fail` p is testing. $\square$

Theorem 23 suggests that to detect concrete bugs, it suffices to look for erroneous symbolic states $\hat{s}$ using the function `next_is_fail` and then check whether there exists at least one concrete state $s \sqsubseteq_M \hat{s}$ (for some memory M). Testing the existence of such a concretization reduces to checking the satisfiability of the path condition of $\hat{s}$. If the path condition is satisfied by some memory M then we found a concrete state s such that $s \sqsubseteq_M \hat{s}$. If the path condition is unsatisfiable, then the erroneous symbolic state $\hat{s}$ represents an empty set of concrete states and it can be ignored. This gives an effective way to detect bugs by traversing the set of all reachable symbolic states and, for each of them, use `next_is_fail` and a constraint solver to detect whether they represent a bug.

Unfortunately, if we find an erroneous symbolic state but the solver fails to decide whether its path condition is satisfiable or not, we cannot really conclude whether there is a bug or not. If we consider this a bug but the path condition is unsatisfiable, then it's a false alarm. If we ignore the error but the path condition turns out to be satisfiable, then we missed a bug. To solve this problem, we make an explicit distinction between two kinds of bugs reports: *sure bugs* and *potential bugs*. We define a type `bug_report` to classify the symbolic states depending on whether or not they represent a bug (and with which level of confidence).

```
Inductive bug_report :=
  | SureBug (b : bexpr)
  | PotentialBug (b : bexpr)
  | NoBug.
```

When a bug is found, the path-condition of the corresponding bad state is also returned. Therefore, bug reports also carry a boolean expression. We provide a function `report` that takes a symbolic state $\langle \varphi, \hat{M}, p \rangle$ as an input and produces a bug report.

```
Definition report (path, mem, p) :=
  if next_is_error p then
    match Solver.check_sat path with
    | SAT => SureBug path
    | UNSAT => NoBug
    | TIMEOUT => PotentialBug path
    end
  else NoBug
```

Turning our symbolic interpreter into a bug finder is as easy as applying `report` on every element of the stream of reachable symbolic states.

```
CoFixpoint map (f : A -> B) (s : stream A) : stream B :=
  match s with
  | snil => snil
  | scons x xs => scons (f x) (map f xs)
  end.

Definition find_bugs p :=
  map report (reachable' [(Bool true, (fun x => Var x), p)]).
```

The function `find_bugs` takes a program p, generate all symbolic states reachable from the initial state $\langle \texttt{true}, x \mapsto x, p\rangle$ (i.e., the initial path condition is set to be true and the symbolic memory is binding each program variable to its own name) and transform this stream of states into a stream of bug reports. Using the correctness of our symbolic interpreter, we can prove that the function `report_bugs` reports all and only bugs: it is a precise and exhaustive bug finder.

Theorem 24 (Precision of the bug finder)**.** *Let p be a program, and r be a bug report in the stream `find_bugs`(p). Then we have the following two properties:*

1. *if $r = \texttt{SureBug}(\varphi)$, then p has a bug*
2. *if $r = \texttt{PotentialBug}(\varphi)$ and φ is satisfiable, then p has a bug*

Additionally, in both cases, executing p from any memory satisfying φ triggers the bug.

Proof. `SureBug`(φ) or `PotentialBug`(φ) occurs in the stream `find_bugs` p only if an erroneous symbolic state $\hat{s}$ occurs in the stream `reachable` $[\langle \text{true}, x \mapsto x, p\rangle]$. By soundness of `reachable'` (22), $\hat{s}$ is reachable according to the symbolic semantics. By soundness of the symbolic semantics and by theorem 23 it follows that every memory M satisfying φ leads to a runtime error in p. Additionally, in the case of `SureBug`(φ), we know that φ is satisfiable (because of the successfull satisfiability check) so there is at least one initial memory leading p to failure. $\square$

Theorem 25 (Exhaustiveness of the bug finder)**.** *Let p be a program, and suppose that p has a bug triggered by the input memory M. Then there exists a bug report r in the stream* `find_bugs`(p) *satisfying one of the two following properties:*

1. *Either $r =$ `SureBug`(φ) and M satisfies φ*
2. *Or $r =$ `PotentialBug`(φ) and M satisfies φ*

Proof. Suppose that executing p from initial memory M leads to a runtime error. By completeness of the symbolic semantics and by 23, there exists a symbolic execution leading to a erroneous symbolic states $\hat{s}$ such that M satisfies the path condition φ of $\hat{s}$. By completeness of `reachable` (17), $\hat{s}$ is necessarily in the stream `reachable` $[\langle \text{true}, x \mapsto x, p\rangle]$. Either the solver successfully detects that φ is satisfiable, in which case `find_bugs` will contain `SureBug`(φ). Or, the solver fails to determine the satisfiability of φ. In this case `find_bugs` will contain `PotentialBug`(φ). $\square$

An immediate corollary of the exhaustiveness is that our bug finder can sometimes be used to prove the absence of bugs! Indeed, if the stream of bug reports is finite (typically, when the analyzed program has no loops) and no bug is reported (i.e., the stream contains only `NoBug` reports), then the program is necessarily free of bugs (otherwise, we would have found the bug before reaching the end of the stream).

Corollary 26 (Freedom of runtime errors)**.** *Suppose that the stream* `find_bugs`(p) *contains only* `NoBug`*, then p is free of runtime errors.*

6.2 Bug finding under assumptions

If a program makes assumptions on the inputs (for example, by requiring a pre-condition to be satisfied in order to execute a function), crashing the program by running it with inputs that violate the assumptions should not be considered a bug. In practice, we are more interested in finding bugs under a certain set of assumptions ψ. To denote the set of states that can be reached from an initial memory that satisfies a precondition ψ, we introduce the a conditional reachability set $\mathit{Reach}^{\psi}(p)$:

Definition 6.1 (Conditional reachability)

$$\text{Reach}^{\psi}(p) = \{s \mid \exists M, [\![\psi]\!]_M = \text{true} \land \langle M, p\rangle \hookrightarrow^* s\}$$

Similar to *Reach* and $\mathit{Reach}_{\text{sym}}$, we can define a symbolic variant of conditional reachability. To model conditional symbolic reachability, we look at symbolic states that are reachable when starting with φ as an initial path condition (instead of *true*).

Definition 6.2 (Conditional symbolic reachability)

$$\mathrm{Reach}^{\psi}_{\mathrm{sym}}(p) = \{s \mid \exists \hat{s}, \exists M, \langle \psi, x \mapsto x, p\rangle \hookrightarrow^{*}_{\mathrm{sym}} \hat{s} \wedge s \sqsubseteq_M \hat{s}\}$$

Without much surprise, conditional reachability coincides with symbolic conditional reachability as stated in the following theorem:

Theorem 27. $\mathit{Reach}^{\psi}(p) = \mathit{Reach}^{\psi}_{\mathit{sym}}(p)$

Proof. Immediate corollary of the soundness and completeness of the symbolic semantics (6, 10). □

To compute the symbolic states that are reachable under assumptions ψ it suffices to call the function `reachable'` from the initial state $\langle \psi, x \mapsto x, p\rangle$ instead of $\langle \texttt{true}, x \mapsto x, p\rangle$. We can therefore easily extend the bug finder `find_bugs` to support assumptions as follows:

```
Definition find_bugs assumptions p :=
  map report (reachable' [(assumptions, fun x => Var x, p)]).
```

By correctness of the function `reachable'` (22) and thanks to theorem 27, the precision and exhaustiveness results established for the bug finder without assumptions (theorems 24 and 25) immediately transfer to the bug finder with assumptions.

Theorem 28 (Precision of the bug finder with assumptions). *Let p a program, ψ a formula representing a set of assumptions, and r a bug report in the stream `bug_reports'` ψ p. Then we have the following two properties:*

1. *if $r = \texttt{SureBug}(\varphi)$, then p has a bug triggered for some inputs satisfying the assumptions ψ*

2. *if $r = \texttt{PotentialBug}(\varphi)$ and φ is satisfiable, then p has a bug triggered for some inputs satisfying the assumptions ψ*

Additionally, in both cases, executing p from any memory satisfying φ triggers the bug.

Theorem 29 (Exhaustiveness of the bug finder with assumptions). *Let p be a program, ψ be a set of assumptions, and suppose M is a bug-triggering input memory satisfying ψ. Then there exists a bug report r in the stream `find_bugs'` ψ p satisfying one of the two following properties:*

1. *Either $r = \texttt{SureBug}(\varphi)$ and M satisfies φ*

2. *Or $r = \texttt{PotentialBug}(\varphi)$ and M satisfies φ*

Corollary 30 (Freedom of runtime errors under assumptions)**.** *Suppose that the stream* `find_bugs'` ψ p *is finite and contains only* `NoBug`*, then* p *is free of runtime errors if executed with inputs satisfying* ψ*.*

6.3 Extraction from Coq to OCaml

6.3.1 Choosing a constraint solver

In the previous section, we implemented and verified in Coq a bug finder based on symbolic execution. This bug finder is presented as a functor parametrized by a constraint solver supposed to be sound. To execute the bug finder, we need to first instantiate the functor with a constraint solver of our choice. To do so, a first option would be to develop a constraint solver in Coq, prove its soundness formally, and instantiate the bug finder in Coq. Then, we can either execute the bug finder directly inside of Coq using the Coq interpreter, or rely on the extraction mechanism to compile the sources of the bug finder to a standalone OCaml executable (see figure 6.1).

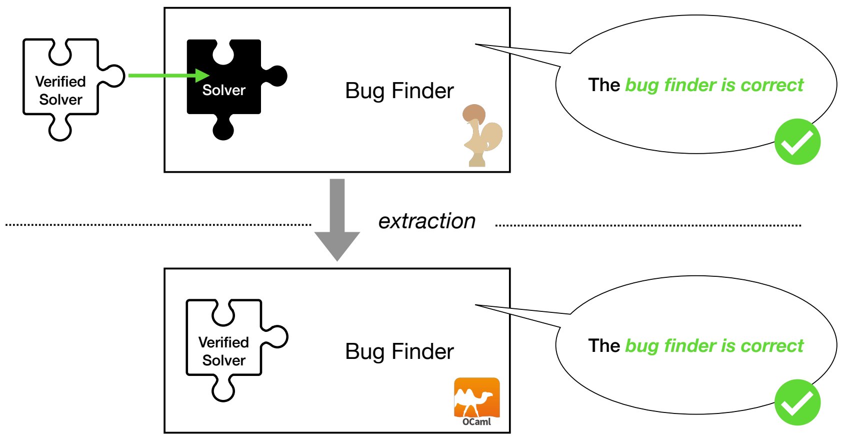


Figure 6.1: Extraction of a fully verified bug finder

Another option is to rely on the extraction mechanism of Coq to compile the sources of the bug finder to a standalone OCaml library. In that case, we can extract the entire `MakeBugFinder` functor to an equivalent OCaml functor. The soundness hypothesis on the solver is erased during the extraction because the type system of OCaml is not powerful enough to express formal specifications and formal requirements on modules.Once the functor is extracted, we can therefore choose an arbitrary (unverified) solver (see figure 6.2). This second method has the drawback that it sacrifices a little bit of reliability. Nonetheless, provided the chosen solver is correct, the core algorithm of the bug finder is still guaranteed to be correct. This is a major improvement in comparisons with existing unverified tools. Further, our design still allows to recover full correctness if some time is dedicated to verifying a constraint solver in Coq.

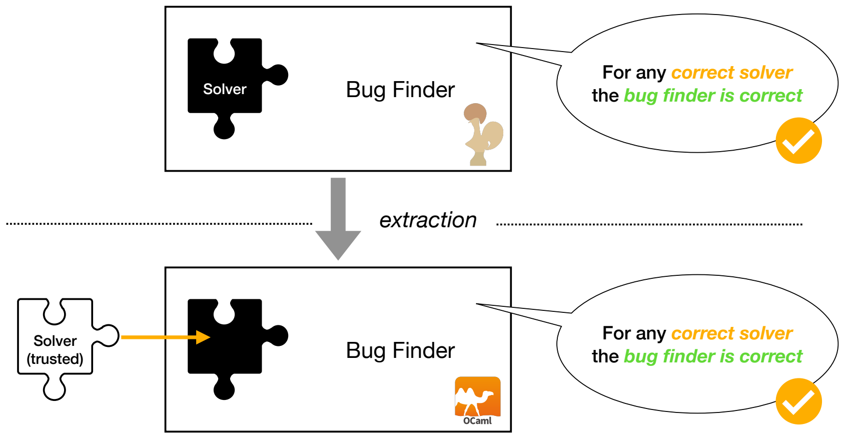


Figure 6.2: Extraction of a verified bug finder parametrized by a trusted solver

Implementing and verifying a fully featured constraint solver in Coq is a research topic in itself and goes far beyond the scope of a master's thesis. Therefore, we chose to only extract the functor `MakeBugFinder` using the following command:

```
Extraction "bugfinder.ml" MakeBugFinder.
```

This generates an OCaml file `bugfinder.ml` containing the OCaml translation of the functor `MakeBugFinder`.

6.3.2 The OCaml Front-end of the bug finder

Once the `MakeBugFinder` is extracted to OCaml, we still need to develop a user interface to communicate with the verified bug-finding algorithm. More specifically, we need to pick a constraint solver, develop a parser for programs, and a pretty printer to display the results of the bug finder. We develop these features in plain OCaml.

Constraint solver. We choose Z3 as a constraint solver [DB08]. To instantiate the functor `MakeBugFinder` we start by wrapping the OCaml API of Z3 into a module `Solver` that implements the module signature `SOUND_SOLVER`. We can then apply the functor `MakeBugFinder` to the module `Solver` to obtain a module `BugFinder`. The module `BugFinder` exposes the function `find_bugs'` that can be called on any BUG program to generate a stream of bug reports.

```
open Z3
module Solver : SOUND_SOLVER = struct
  ...
end
module BugFinder = MakeBugFinder(Solver)
```

To obtain better performance, our implementation of the module `Solver` uses a cache to store the results of satisfiability queries. If the same query needs to be solved twice, Z3 will therefore be called only once.

Parsing. To read programs from files, we implemented a simple parser that accepts programs expressed in a concrete syntax close to the abstract syntax of BUG presented in figure 3.1. We extend the syntax of BUG with the macros `assume` and `assert` to instrument programs. The command `assume` can only be used at the beginning of a file and is used to indicate under which assumptions the symbolic execution should be performed (see 6.2). The command `assert cond` is used to check a boolean condition at runtime and it is just an alias for the snippet `if cond then skip else fail`. Given a source file, our parser returns a pair `(assumptions, program)` of the assumption (as a boolean expression) and the abstract syntax tree of the program.

Pretty printing. Once a program is parsed, we use the function

```
BugFinder.find_bugs assumptions program
```

to produce a lazy stream of bug reports. Then, we traverse the stream and display appropriate error messages if negative bug reports are found. For every displayed error message, the precision theorem 28 ensures that it corresponds to a real bug. Because the stream can be infinite, traversing the stream might never terminates. Nonetheless, if there is a bug in the program, the exhaustiveness theorem 29 ensures that an error message will eventually be displayed if we let the bug finder run long enough. Further, if the stream is exhausted in finite time and no bugs were displayed, the corollary 30 ensures that the program under test is free of bugs (under the given assumptions).

6.3.3 Comments on the Coq development

In the end, the complete development of our verified bug finder is composed of approximately 1200 lines of Coq code and proofs and 400 lines of OCaml code.

Component	Lines	Language
Source code and proofs	1219	Coq
Extracted code	200	OCaml
Front-end (parser + Z3 + printer)	189	OCaml

As described in the above table, once extracted, the code of the bug finder only represents 200 of the 1200 lines of Coq. The remaining 1000 lines contains the formalization of the target programming language, its 3 formal semantics (concrete, symbolic, symbolic + pruning), and all theorems and proofs discussed in this thesis. The list of formally proven theorems also includes many technical lemmas we did not not discuss in this thesis but were required to prove the correctness of the bug finder (most notably, lemmas on stream functions).

6.4 Evaluation

We evaluate our verified bug finder on example programs to demonstrate its capabilities and its current limitations.

6.4.1 Sample programs

GCD. The first program we test is an algorithm to compute the GCD of two positive integers a and b using iterative subtraction. At every iteration of the algorithm, we check that the variant $a + b$ is strictly decreasing to ensure termination. We give two versions of the algorithm: a correct one (on the left), and an incorrect one (on the right). The incorrect version has a bug leading to potential non-termination (indicated in red). The purpose of this sample is to demonstrate the ability for our bug finder to find bugs in arithmetic programs.

```
assume a > 0
assume b > 0
while (a != b) {
  old_a = a
  old_b = b
  if (a > b) {
    a = a - b
  } else {
    b = b - a
  }
  assert (a + b < old_a + old_b)
}
```

```
assume a > 0
assume b > 0
while (a != b) {
  old_a = a
  old_b = b
  if (a > b) {
    a = a - b
  } else {
    b = b + a
  }
  assert (a + b < old_a + old_b)
}
```

Bounded loops. The second sample program tests the ability of our bug finder to prove the absence of bugs for programs with bounded loops. The program on the left can crash when the bound k is at least 100. If we restrict k to be smaller, the program is bug free.

```
assume 0 <= k
assume 0 <= x
while x < k {
  x = x + 1
  assert (x <= 100)
}
```

```
assume 0 <= k <= 100
assume 0 <= x
while x < k {
  x = x + 1
  assert (x <= 100)
}
```

Deep bugs. Finally, the last sample program we use exercises the ability for our bug finder to find deep bugs. The program simply iterate a variable x from 0 to k and then crashes. In the evaluation, we run the bug finder with $k \in \{100, 500, 1000\}$.

```
x = 0
while true {
  x = x + 1
  assert (x < 500)
}
```

6.4.2 Results

For each program, we run two versions of the bug finder: one with on-the-fly pruning of infeasible paths (this corresponds to the symbolic semantics presented in 5) and one which only check the feasibility of paths leading to errors (this corresponds to the initial semantics 3). In each case, we measure the runtime in seconds as well as the number of SMT queries sent to Z3. Because the stream of bug reports produced by the bug finder can be infinite, we set a timeout after processing the first 5000 bug reports (we note TO in the table). The evaluation is performed on a MacBook PRO with an Apple M1 Pro processor and 16GB of ram. The results are gathered in table 6.3.

For the GCD sample program, both versions of the bug finder find the bug instantly. For the bounded loop sample, the bug finder without pruning fails to prove the absence of bugs. This is because it cannot detect that the program exits the bounded loop after 100 iterations. However, the bug finder with on-the-fly pruning proves the absence of bug in less than a second.

The last sample gives more surprising results. Indeed, for $k = 100$ and $k = 500$, the version without pruning finds the bug significantly faster than the version with on-the-fly pruning. This can be explained because finding the bug essentially boils down to concretely executing the program for the first k iterations until the bug is hit. As a

Program	Expected	Pruning	Runtime	Queries	Result
GCD	bug	no	< 0.01	2	bug found
		yes	< 0.01	10	bug found
Bounded loops	no bugs	no	32	714	TO
		yes	0.7	402	proved
Deep bugs $k = 100$	bug	no	0.05	100	bug found
		yes	0.2	401	bug found
Deep bugs $k = 500$	bug	no	4	500	bug found
		yes	17	2001	bug found
Deep bugs $k = 1000$	bug	no	12	714	TO
		yes	135	4001	TO

Figure 6.3: Evaluation of the verified bug finder on sample programs

result, the bug finder without pruning only sends k queries to the solver (once at every iteration of the loop, to check whether the assertion holds). In contrast, the bug finder with pruning sends a query to the solver for every intermediate execution step. This results in a significant time overhead. For $k = 1000$, both versions of the bug finders fail to see the bug after processing the first 5000 states generated by the symbolic interpreter. This is due to the fact that we use a naive breadth-first strategy and hence, the number of states to explore before reaching deep states is high. To overcome these limitations, we could explore two solutions:

1. A first solution would be to develop a symbolic semantics with *selective* pruning. Instead of testing the satisfiability of the path condition for every generated state, we could develop heuristics to decide when to call the solver. This would result in a symbolic semantics that is still sound and complete but with an implementation that generates less SMT queries.

2. Another solution is to replace the breadth-first search strategy with a more efficient strategy that still guarantees exhaustiveness. For example, depth-first search with iterative deepening. This requires more work and in particular more proof engineering as one would need to verify the underlying datastructures used in the search algorithm.

Chapter 7

Related Work

7.1 Verified program provers

Proof assistants have been successfully applied to verify the correctness of automated program provers. For example, Verasco is a static analyzer for C programs based on abstract interpretation [Jou+15]. It can prove freedom of runtime-errors and it is entirely verified using the Coq proof assistant.

Other verification methods such as model-checking for temporal properties have been verified using a proof assistant. For example, the CAVA [Esp+13; BL18] model checker is verified using the proof assistant Isabelle/HOL and extracted to efficient ML code using the refinement framework of Isabelle/HOL. CAVA accepts specifications expressed in Linear Temporal Logic and systems expressed in the modeling language Promela, and can automatically verify that a system respect its specification.

Our work use the same technique than CAVA and Verasco: we implement a verification tool in a proof assistant and develop a formal proof of its correctness. However, while Verasco and CAVA focus on proving correctness of programs using abstract interpretation and LTL model-checking, we focus on automatically showing the presence of bugs using symbolic execution. Additionally, CAVA can only verify finite-state models of programs.

7.2 Formal foundations of bug-finding

Until recently, formal foundations of bug-finding tools received less attention than automated provers. In this section, we present recent work going in the direction of proving the correctness of bug-finding tools and compare our work to these other approaches.

Incorrectness Logic In the paper *Incorrectness Logic* [OHe19], Peter W. O'Hearn was the first to propose a program logic to reason about incorrect behavior of programs rather than proving their correctness using the more traditional Hoare logic. This work layed the foundations to design formal systems dedicated to proving the presence of bugs rather than proving their absence.

Formal foundations of symbolic execution The semantic foundations of symbolic execution have already been studied. For example, Boer and Bonsangue proposed the first formal framework to formally reason about symbolic execution as just another semantics to execute programs [BB19; BB21; Boe+20]. In a sequence of papers, they proposed different symbolic semantics to justify the correctness of symbolic execution for various programming languages with different features (objects, recursion, parallelism, etc). However, to the best of our knowledge, their theoretical framework is not mechanized in a proof assistant.

In [Fra+20], Fragoso et al propose to develop a generic platform for symbolic execution of programs based on solid semantic foundations. They proved, manually, that their symbolic execution engine is correct with respect to the semantics of a core programming language. Their platform, called Gillian offers bug-finding features. However, their tool is not mechanized in a proof assistant and its correctness proof is not machine-checked.

Going on step further, in [Por+22], Pornchroenwase et al propose to formally verify an executable symbolic evaluator for a subset of the Scheme language. Their symbolic engine called Leanette, is mechanized using the Lean proof assistant[1] and is proven to be correct. In their approach, they propose a big-step symbolic semantics that execute programs until termination and collect all possible program paths. This restrict their work to the analysis of terminating programs. Instead, we propose to formally verify in Coq a symbolic interpreter based on a small-step operational semantics. Our interpreter can therefore provide insightful information on non-terminating programs which makes it more suitable for bug-finding tasks.

In [Keu+22a], Keuchel et al mechanized a symbolic execution engine in Coq to automatically discharge obligations in Coq proofs involving reasoning about programs. Their approach focuses on symbolic execution-based program verifiers that are embedded within a proof assistant. In contrast our approach focus on the verification of a stan-

[1] `https://lean-lang.org`

dalone symbolic execution engine dedicated to bug finding, and it can be used outside of the proof assistant.

Verified property-based testing Property Based Testing (PBT) [CH00] is another automated testing method. It automatically generates inputs to test whether a program is conforming to the a given specification. In [Par+15], Paraskevopoulou et al propose to implement a PBT framework offering strong formal guarantees. Their framework, called QuickChick, is implemented as a library in the Coq proof assistant. Given a specification that a Coq function should satisfy, their library provides trustworthy inputs generators to automatically test whether the specification is satisfied. This approach allows to reliably test code written in Coq but it cannot be applied outside of the context of the proof assistant.

Chapter 8

Conclusion

Automated testing is an important method to quickly evaluate the robustness or the correctness of large software packages. Nonetheless, without formal foundations, testing only provides relatively weak guarantees. If an automated testing tool finds a bug, it might be a false alarm. If it does not find any bug, it does not necessarily mean that the analyzed program is free of bugs. In this thesis, we proposed to formally verify the correctness of an automated bug finder based on symbolic execution. We developed a symbolic interpreter for a small programming language and derived an automatic bug finder. Using the Coq proof assistant, we then carried the complete proof of correctness of this automated bug finding tool. In particular, we proved that our symbolic interpreter is faithful to the formal semantics of our target programming language. This later allowed us to prove that our bug finder is precise and exhaustive: it finds all and only real bugs of the analyzed programs. Further, since exhaustiveness has been formally proven, an interesting side effect is that it can be used to prove the absence of bugs on programs without loops (or programs with bounded loops with a static bound).

Future work. Our verified bug finder is a prototype and our target programming language is not realistic (it does not support functions, arrays, data-structures, etc). Nonetheless, our prototype is functional and it can be used to find bugs (or prove the absence of them) in sample programs. In future work, we hope to extend the principles developed in this thesis to more realistic programming languages whose semantics have already been formalized in the Coq proof assistant (for example C or JavaScript [Ler09; Bod+14]). For our approach to scale to the analysis of realistic programming languages, we anticipate multiple challenges. First, we will need to design new types of symbolic semantics to capture the features of modern programming languages. Further, these new symbolic semantics should be easily implementable in the form of an efficient symbolic interpreter. In particular, it is known that simplifying path conditions on-the-fly before sending them to the constraint solver can greatly reduce the runtime of symbolic verification tools [Cor14]. Verifying symbolic interpreters that supports such optimizations is a interesting research direction. Another important challenge is the design and the verification of efficient data-structures and algorithms for exhaustively traversing the search space generated by symbolic interpreters. The automated testing literature abounds with search heuristics empirically known to be efficient for bug-finding by symbolic execution [GKS05; GLM08; He+21]. However, relying on heuristics only sometimes ensures the exhaustiveness of the search for bugs. It could be interesting to study what can be done to turn existing search heuristics into provably exhaustive search algorithms and implement these new algorithms in Coq.

Bibliography

[Arm+11] Mickaël Armand et al. "Verifying SAT and SMT in Coq for a fully automated decision procedure". In: (Aug. 2011).

[Bar+11] Clark W. Barrett et al. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_14.

[Bau+21] Patrick Baudin et al. "The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform". In: *Commun. ACM* 64.8 (July 2021), pp. 56–68. ISSN: 0001-0782. DOI: 10.1145/3470569. URL: https://doi.org/10.1145/3470569.

[BB19] Frank S. de Boer and Marcello M. Bonsangue. "On the Nature of Symbolic Execution". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Proceedings*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 64–80. DOI: 10.1007/978-3-030-30942-8_6. URL: https://doi.org/10.1007/978-3-030-30942-8%5C_6.

[BB21] Frank S. de Boer and Marcello Bonsangue. "Symbolic execution formally explained". In: *Formal Aspects of Computing* 33.4 (2021), pp. 617–636. DOI: 10.1007/s00165-020-00527-y. URL: https://doi.org/10.1007/s00165-020-00527-y.

[Bie+09] Armin Biere et al. "Bounded model checking." In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481.

[BL18] Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction". In: *J. Autom. Reason.* 60.1 (Jan. 2018), pp. 3–21. ISSN: 0168-7433. DOI: 10.1007/s10817-017-9418-4. URL: https://doi.org/10.1007/s10817-017-9418-4.

[Bla+03] B. Blanchet et al. "A Static Analyzer for Large Safety-Critical Software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. San Diego, California, USA: ACM Press, June 2003, pp. 196–207.

[Bod+14] Martin Bodin et al. "A Trusted Mechanised JavaScript Specification". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 87–100. ISSN: 0362-1340. DOI: 10.1145/2578855.2535876. URL: https://doi.org/10.1145/2578855.2535876.

[Boe+20] Frank S de Boer et al. "SymPaths: Symbolic execution meets partial order reduction". In: *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY* (2020), pp. 313–338.

[Bou+09] Thomas Bouton et al. "veriT: An Open, Trustable and Efficient SMT-Solver". In: *Automated Deduction – CADE-22*. Ed. by Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 151–156.

[Bun+21] Joshua Bundt et al. "Evaluating Synthetic Bugs". In: *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security*. Ed. by Jiannong Cao et al. ACM, 2021. DOI: 10.1145/3433210.3453096. URL: https://doi.org/10.1145/3433210.3453096.

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1977, pp. 238–252.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. "Model Checking: Algorithmic Verification and Debugging". In: *Commun. ACM* 52.11 (Nov. 2009), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/1592761.1592781. URL: https://doi.org/10.1145/1592761.1592781.

[CH00] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266. URL: https://doi.org/10.1145/357766.351266.

[Cor] Arthur Correnson. *A la recherche de tous les vrais bugs – Vérification formalle d'un détecteur de bugs automatique*. To appear at JFLA 2024.

[Cor14] Loic Correnson. "Qed. Computing What Remains to Be Proved". In: *NASA Formal Methods*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Springer International Publishing, 2014, pp. 215–229. ISBN: 978-3-319-06200-6.

[CS23] Arthur Correnson and Dominic Steinhöfel. "Engineering a Formally Verified Automated Bug Finder". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 1165–1176. ISBN: 9798400703270. DOI:

10.1145/3611643.3616290. URL: https://doi.org/10.1145/3611643.3616290.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[Dij75] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://doi.org/10.1145/360933.360975.

[Esp+13] Javier Esparza et al. "A Fully Verified Executable LTL Model Checker". In: *Computer Aided Verification (CAV 2013)*. Ed. by N. Sharygina and H. Veith. Vol. 8044. 2013, pp. 463–478.

[Fio+20] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies*. Ed. by Yuval Yarom and Sarah Zennou. USENIX Association, 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[Fra+20] José Fragoso Santos et al. "Gillian, Part i: A Multi-Language Platform for Symbolic Execution". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 927–942. ISBN: 9781450376136. DOI: 10.1145/3385412.3386014. URL: https://doi.org/10.1145/3385412.3386014.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David Molnar. "Automated Whitebox Fuzz Testing". In: Nov. 2008.

[God05] Patrice Godefroid. "The Soundness of Bugs is What Matters (Position Statement)". In: *PLDI'05 Workshop on the Evaluation of Software Defect Detection Tools (BUGS'05), Proceedings*. 2005. URL: https://patricegodefroid.github.io/public_psfiles/bugs2005.pdf.

[He+21] Jingxuan He et al. "Learning to Explore Paths for Symbolic Execution". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2526–2540. ISBN: 9781450384544. DOI: 10.1145/3460120.3484813. URL: https://doi.org/10.1145/3460120.3484813.

[Jou+15] Jacques-Henri Jourdan et al. "A Formally-Verified C Static Analyzer". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 247–259. DOI: 10.1145/2676726.2676966. URL: https://doi.org/10.1145/2676726.2676966.

[Keu+22a] Steven Keuchel et al. "Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming)". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547628. URL: https://doi.org/10.1145/3547628.

[Keu+22b] Steven Keuchel et al. "Verified Symbolic Execution with Kripke Specification Monads (and no Meta-Programming)". In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 194–224. DOI: 10.1145/3547628. URL: https://doi.org/10.1145/3547628.

[Kin76] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976). DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[Kne91] Ralf Kneuper. "Symbolic Execution: A Semantic Approach". In: *Sci. Comput. Program.* 16.3 (1991), pp. 207–249. DOI: 10.1016/0167-6423(91)90008-L.

[Ler09] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (2009). DOI: 10.1007/s10817-009-9155-4. URL: https://doi.org/10.1007/s10817-009-9155-4.

[Ler11] Xavier Leroy. "Verified squared: does critical software deserve verified tools?" In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 1–2. DOI: 10.1145/1926385.1926387. URL: https://doi.org/10.1145/1926385.1926387.

[Let08] Pierre Letouzey. "Extraction in Coq: An Overview". In: *Logic and Theory of Algorithms*. Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 359–369. ISBN: 978-3-540-69407-6.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[OHe19] Peter W. O'Hearn. "Incorrectness Logic". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371078. URL: https://doi.org/10.1145/3371078.

[Par+15] Zoe Paraskevopoulou et al. "Foundational Property-Based Testing". In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang.

Cham: Springer International Publishing, 2015, pp. 325–343. ISBN: 978-3-319-22102-1.

[Por+22] Sorawee Porncharoenwase et al. "A Formal Foundation for Symbolic Evaluation with Merging". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: `10.1145/3498709`. URL: `https://doi.org/10.1145/3498709`.

[Soz] Matthieu Sozeau. *The Coq documentation (version 8.18.0) - Language Extensions - Program*. URL: `https://coq.inria.fr/refman/addendum/program.html`.

[Soz10] Matthieu Sozeau. "Equations: A Dependent Pattern-Matching Compiler". In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 419–434.

[Ste20] Dominic Steinhöfel. "Abstract Execution: Automatically Proving Infinitely Many Programs". PhD thesis. Darmstadt University of Technology, Germany, 2020. URL: `http://tuprints.ulb.tu-darmstadt.de/8540/`.

[WZS20] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: `10.1145/3428261`. URL: `https://doi.org/10.1145/3428261`.