SEMANTICS-BASED PROGRAM VERIFICATION

BY

DAEJUN PARK

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Grigore Roșu, Chair
Professor Vikram Adve
Assistant Professor Andrew Miller
Principal Researcher Nikolaj Bjørner, Microsoft Research

ABSTRACT

We present language-independent formal methods that are parameterized by the operational semantics of languages. We provide the theory, implementation, and extensive evaluation of the language-parametric formal methods. Specifically, we consider two formal analyses: program verification and program equivalence.

First, we propose a novel notion of bisimulation, which we call *cut-bisimulation*, allowing the two programs to semantically synchronize at relevant "cut" points, but to evolve independently otherwise. Employing the cut-bisimulation, we develop a language-independent equivalence checking algorithm, parameterized by the input and output language semantics, to prove equivalence of programs written in possibly different languages. We implement the algorithm in the K framework, yielding the first language-parametric program equivalence checker.

To demonstrate the practical feasibility of the language-parametric formal methods, we instantiate a language-independent deductive program verifier by plugging-in four real-world language semantics, C, Java, JavaScript, and Ethereum Virtual Machine (EVM), and use them to verify full functional correctness of challenging heap-manipulating programs and high-profile commercial smart contracts. In particular, to the best of our knowledge, the JavaScript and EVM verifiers are the first deductive program verifier for these languages.

To Joshua.

ACKNOWLEDGMENTS

Thanks to my advisor, Grigore Roșu for the continuous support of my Ph.D. study, for his patience, motivation, and enthusiasm. Thanks to the members of my thesis committee for their insightful comments, encouragement, and hard questions. Thanks to my fellow doctoral students over the course of my studies. And finally, thanks to my wife who assisted me through this long journey with advice, child care, and love.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Most of the existing formal program analysis tools (e.g., program verifiers) lack reusability. They were developed for a fixed target language, and the language semantics is usually hardcoded within the implementation of their formal analysis algorithm. Due to their monolithic design nature, retargeting them to another language requires significant effort. The retargeting requires either to replace the hardcoded language semantics with the new target language semantics, or to implement a translation from the new language to the existing language. The replacement of the hardcoded semantics almost amounts to rewriting the entire codebase, which is not viable in most cases. The language translation approach could be better than the semantics replacement, requiring affordable effort in case that the new language is similar to the existing one. However, if the new language is quite different from the existing language (for example, a register-based assembly language versus a stack-based one), translation would be at best ineffective if not infeasible. Moreover, the new language semantics (either defined in hardcoded form or indirectly defined via translation) may not be reusable for other formal tools. The lack of reusability leads to fragmentation in the formal tool community.

Language-independent formal methods [1, 2, 3, 4, 5] have been proposed to mitigate the reusability issue. The idea is to develop universal formal methods and tools that are *parameterized by language semantics*, and derive formal analysis tools for each target language by instantiating the universal ones with plugging-in the target language semantics. Moreover, theories [5, 4, 6] have been established that allow to plug-in *operational* language semantics. The underlying theories enable formal reasoning about programs directly over operational semantics without the burden of having to specify additional axiomatic semantics and prove its soundness with respect to the operational semantics. These operational-semantics-based formal methods benefit from the fact that operational semantics is easier to specify than axiomatic semantics. Indeed, specifying operational semantics amounts to writing an interpreter for the target language, which is doable even without extensive logical-theoretical background.

While the underlying theory of the proposed methodology has already been established, its practicality has not been comprehensively evaluated. Although a proof of concept implementation and its preliminary evaluation were provided [1], they are rather limited.

In this thesis, we demonstrate and improve the scalability and practicality of the language-independent formal methods parameterized by operational semantics, which falls into the following categories:

- Developing language-independent formal methods.
- Specifying real-world language semantics and measuring the specification effort.
- Instantiating the language-independent formal methods by plugging-in various real-world programming language semantics.
- Applying the derived formal analysis tools to real-world systems and applications, demonstrating their practical feasibility both in isolation and by comparison to state-of-the-art tools specifically crafted for the target languages.

Cross-Language Program Equivalence We have developed a language-independent equivalence checking algorithm, parameterized by the input and output language semantics, to prove equivalence of programs written in different languages. The algorithm employs a novel notion of bisimulation, which we call *cut-bisimulation*, allowing the two programs to semantically synchronize at relevant "cut" points, but to evolve independently otherwise. The program point pairs needed for the proof are provided as an input to the algorithm, and are called synchronization points. Intuitively, synchronization points are symbolic descriptions that capture the set of pairs of relevant (concrete) states of the input and output programs. For example, they include the pairs of (symbolic) input and output states of the two programs, and the beginnings of the same loop or cyclic structure of them.

Using the $\mathbb{K}$ semantic framework, we have implemented the first language-independent tool for proving program equivalence, parametric in the formal semantics of the two languages. The new tool, called KEQ, takes two language semantics as input and yields a checker that takes two programs as input, one in each language, and a (symbolic) synchronization relation, and checks whether the two programs are indeed equivalent with the synchronization relation as a witness. Intuitively, KEQ symbolically executes the two programs, each with its respective (symbolic) input state, until it reaches another pair of states presented in the synchronization points. Then, KEQ re-starts the symbolic execution from the newly reached states until reaching another pair of states. KEQ repeats this process until it reaches the pair of output states, which concludes the equivalence.

Complete Formal Semantics of JavaScript To evaluate the effectiveness of specifying operational language semantics, as well as to use it for instantiating the universal formal methods, we have completely specified, in $\mathbb{K}$, an operational semantics of JavaScript [7], one of the most popular client-side programming languages. The JavaScript semantics,

called KJS, is the most complete and thoroughly tested formal JavaScript semantics, specifically of ECMAScript 5.1, the latest language standard at the time of writing it. It has been tested against the ECMAScript conformance test suite, and passed all 2,782 test programs for the core language. KJS is far more complete than any other semantics, and even more standards-compliant than production JavaScript engines such as Safari WebKit and Firefox SpiderMonkey. Despite the complex nature of the language semantics, the development of KJS took only four months by a single person with no prior knowledge of JavaScript or of the semantic framework. This result supports the argument that specifying operational semantics is affordable, and the operational-semantics-based formal methods benefit from that.

Formal Verification of Ethereum Smart Contracts As a comprehensive evaluation, we have applied the universal program verifier to Ethereum smart contracts. The Ethereum smart contract is a safety-critical system whose failures have caused millions of dollars of lost funds [8], and rigorous formal methods are required to ensure the correctness and security of contract implementations [9, 10].

We chose the Ethereum Virtual Machine (EVM) bytecode as the verification target language so that we can directly verify what is actually executed without the need to trust the correctness of the compiler. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted KEVM [11], a complete formal semantics of the EVM, and instantiated our language-independent deductive program verifier [12] to generate a correct-by-construction deductive program verifier for the EVM. While it is sound, the initial out-of-box EVM verifier was relatively slow and failed to prove many correct programs. We further optimized the verifier by introducing custom abstractions and lemmas specific to EVM that expedite proof searching in the underlying theorem prover. Our EVM verifier has been used to verify the full functional correctness of high-profile smart contracts including various ERC20 token [13], Ethereum's Casper FFG [14], DappHub's MakerDAO [15], and Gnosis Safe [16] contracts.

Formal Verification of Heap Manipulating Programs For more thorough evaluation of the universal formal methods and performance of their derived formal analysis tools, we have instantiated our language-independent deductive program verifier [12] by plugging-in three real-world language semantics, C, Java, and JavaScript, and used them to verify full functional correctness of challenging heap-manipulating programs such as AVL trees and red-black trees, written in each different language. Performance of the derived verifiers is comparable to other state-of-the-art language-specific verifiers. For example,

VCDryad [17], a separation logic verifier for C built on top of VCC [18], takes 260 seconds to verify only the `balance` function in AVL, while it takes our derived C verifier 210 seconds to verify AVL `insert` (including `balance`). This result is supporting evidence for reusability of the universal formal methods.

End-to-End Verification In end-to-end verification, it is common to build a formal model of a system of interest at a high-level, reason about desirable properties of the model, generate (either automatically or manually) an implementation from the model (possibly in multiple steps), and prove that the implementation refines the model. Ideally, the refinement proof includes reasoning about low-level properties (e.g., memory safety) of the implementation, and preservation of the high-level properties of the model in the generated implementation. This verification methodology allows us to reason about each of the properties at the right level of abstraction.

We have demonstrated that our developed techniques and tools successfully realize the end-to-end verification methodology. We give an executable formal model of a system as a set of rewrite rules, and reason about properties of the model using the reachability logic theorem prover. Then we take an implementation of the system, and prove the refinement using cut-bisimulation, that is, that the implementation is consistent with the model. We can also ensure that the properties of the model are preserved in the implementation by showing that the cut-simulation relation is property-preserving.

Contributions Below are the primary contributions of this thesis:

- We have developed the first cross-language program equivalence checker, based on a novel notion of cut-bisimulation, and in the process we also significantly improved the existing language-parametric deductive program verifier.
- We have specified a complete semantics of a high-profile language, JavaScript, and showed that the specification effort is affordable.
- We have instantiated the universal program verifier with four real-world programming languages, C, Java, JavaScript, and EVM, where both JavaScript and EVM verifiers are the first deductive program verifiers for these languages, to the best of our knowledge.
- We have applied the derived program verifiers and program equivalence checkers to challenging heap manipulation programs and high-profile commercial smart contracts for end-to-end verification, demonstrating their scalability and practicality.

CHAPTER 2: BACKGROUND

In this chapter, we provide background on Matching logic reachability and its implementation in $\mathbb{K}$. We also provide background on Bisimulation and its variants. Much of the content in this chapter comes from Stefanescu *et al.* [12] and Sangiorgi [19].

2.1 MATCHING LOGIC REACHABILITY

We present our program verification foundation, which turns an operational semantics of a language into a sound and relatively complete procedure for proving reachability for that language. The idea is to treat both the operational semantics rules and the program correctness specifications as reachability rules between matching logic patterns, and to use a fixed and language-independent proof system to derive the specifications.

2.1.1 Matching Logic

Matching logic [20] is a logic for specifying and reasoning about structure by means of patterns and pattern matching. Its sentences, the *patterns*, are constructed using *variables*, *symbols*, *connectives* and *quantifiers*, but no difference is made between function and predicate symbols. In models, a pattern evaluates into a power-set domain (the set of values that *match* it), in contrast to FOL where functions and predicates map into a regular domain. Matching logic generalizes several logical frameworks important for program analysis, such as FOL with equality and separation logic. An early variant of matching logic was presented in [21]; here we use the latest variant in [20].

For a set of sorts S, assume Var is an S-sorted set of variables. We write $x:s$ for $x \in Var_s$; when s is irrelevant, we write $x \in Var$. Let $\mathcal{P}(M)$ denote the powerset of M.

Definition 2.1. *Let (S, Σ) be a many-sorted signature of* symbols*. Matching logic (S, Σ)-*formulae*, or (S, Σ)-*patterns*, are inductively defined for all sorts $s \in S$ as follows:*

$$\begin{array}{rcl} \varphi_s & ::= & x \in Var_s \quad | \quad \sigma(\varphi_{s_1}, ..., \varphi_{s_n}) \ \textit{with} \ \sigma \in \Sigma_{s_1 ... s_n, s} \\ & | & \neg\varphi_s \quad | \quad \varphi_s \wedge \varphi_s \quad | \quad \exists x.\varphi_s \ \textit{with} \ x \in Var \end{array} \tag{2.1}$$

Derived constructs can also be used, e.g., $\bot_s$ for $x:s \wedge \neg x:s$, $\varphi_1 \rightarrow \varphi_2$ for $\neg(\varphi_1 \wedge \neg\varphi_2)$, etc. Compared to FOL, matching logic thus collapses all the operation and predicate

symbols into just symbols, used to build patterns, which generalize the usual FOL terms by allowing logical connectives over them.

Definition 2.2. *A matching logic (S, Σ)-model M is an S-sorted set $\{M_s\}_{s \in S}$ and together with interpretation maps $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ for all symbols $\sigma \in \Sigma_{s_1 \ldots s_n, s}$.*

Usual FOL (S, Σ)-models/algebras are a special case, where $|\sigma_M(m_1, ..., m_n)| = 1$ for any $m_1 \in M_{s_1}, \ldots, m_n \in M_{s_n}$. Similarly, partial (S, Σ)-algebras also fall as special case, where $|\sigma_M(m_1, ..., m_n)| \leq 1$, since we can capture the undefinedness of σ_M on $m_1, \ldots, m_n$ with $\sigma_M(m_1, ..., m_n) = \emptyset$.

We tacitly use the same notation σ_M for its extension $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$ to argument sets, i.e., $\sigma_M(A_1, ..., A_n) = \bigcup\{\sigma_M(a_1, ..., a_n) \mid a_1 \in A_1, ..., a_n \in A_n\}$, where $A_1 \subseteq M_{s_1}, ..., A_n \subseteq M_{s_n}$.

Definition 2.3. *Given a model M and a map $\rho : Var \to M$, called an M-valuation, let its extension $\overline{\rho} : \text{PATTERN} \to \mathcal{P}(M)$ be inductively defined as follows:*

- $\overline{\rho}(x) = \{\rho(x)\}$, *for all* $x \in Var_s$
- $\overline{\rho}(\sigma(\varphi_{s_1}, ..., \varphi_{s_n})) = \sigma_M(\overline{\rho}(\varphi_1), ...\overline{\rho}(\varphi_n))$
- $\overline{\rho}(\neg\varphi_s) = M_s \setminus \overline{\rho}(\varphi_s)$ *("$\setminus$" is set difference)*
- $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : Var \to M,\ \rho'|_{Var \setminus \{x\}} = \rho|_{Var \setminus \{x\}}\}$ *("$\rho|_A$" is ρ restricted to A)*

The intuition for the elements in $\overline{\rho}(\varphi_s)$ is that they *match* the pattern φ_s, with witness ρ. For example, suppose that Σ is the signature of Peano natural numbers and M is the model of natural numbers with 0 and *succ* interpreted accordingly. Then $\overline{\rho}(succ(x))$ is interpreted as the singleton set containing only the successor of $\rho(x)$ in M; that is, given ρ, the pattern $succ(x)$ is only matched by the successor of $\rho(x)$. Further, the pattern $\exists x \,.\, succ(x)$ is matched by all positive numbers, and $0 \vee \exists x \,.\, succ(x)$ by all numbers, that is, it is *satisfied* by M:

Definition 2.4. *We write $(\gamma, \rho) \models \varphi_s$ when the particular matching element $\gamma \in \overline{\rho}(\varphi_s)$ needs to be emphasized. M* satisfies *φ_s, written $M \models \varphi_s$, iff $\overline{\rho}(\varphi_s) = M_s$ for all $\rho : Var \to M$, iff $(\gamma, \rho) \models \varphi_s$ for all ρ and γ. Pattern φ is* valid, *written $\models \varphi$, iff $M \models \varphi$ for all M. A* matching logic theory *is a triple (S, Σ, F) with F a set of patterns.*

An interesting aspect of matching logic explained in detail in [20] is that, unlike FOL, it can define equality. With it, we can state that a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is interpreted as a function with the pattern $\exists y \,.\, \sigma(x_1, ..., x_n) = y$ (free variables are assumed universally quantified over the entire pattern). Similar patterns can define partial/injective/surjective functions, total relations, and so on. When a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is to be interpreted as a function, the functional notation $\sigma : s_1 \times \cdots s_n \to s$ can be used instead of the equation above; similarly we use $\sigma : s_1 \times \cdots s_n \rightharpoonup s$ for partial functions. With this, algebraic specifications and FOL with equality, partial or not, fall as syntactic sugar in matching logic. For practical reasons and notational convenience, from here on we assume a pre-defined sort *Bool* and take the freedom to write *Bool* patterns in any sort context as a shorthand for their equality to *true*. With our Peano numbers above, for example, the pattern $\exists x.(succ(x) \wedge (x > 0))$ is a shorthand for $\exists x.(succ(x) \wedge (x > 0 = true))$ and thus specifies all the natural numbers strictly larger than 1.

Separation logic (see, e.g., [22]) can be framed as a matching logic theory over a map model [20]. Indeed, let $S = \{Nat, Bool, Map\}$ and Σ contain the map symbols $emp : \to Map$, $\_ \mapsto \_ : Nat \times Nat \rightharpoonup Map$, and $\_ * \_ : Map \times Map \rightharpoonup Map$. Consider the canonical model of finite-domain partial maps M, where: $M_{Nat} = \{0, 1, 2, ...\}$; $M_{Map} =$ partial maps from natural numbers to natural numbers with finite domains and undefined in 0, with *emp* interpreted as the map undefined everywhere, with $\_ \mapsto \_$ interpreted as the corresponding one-element partial map except when the first argument is 0 in which case it is undefined, and with $\_ * \_$ interpreted as map merge when the two maps have disjoint domains, or undefined otherwise. One may want to add pattern axioms stating that $*$ is associative, commutative and has *emp* as unit, that $0 \mapsto a = \bot_{Map}$, that $x \mapsto a * x \mapsto b = \bot_{Map}$, and so on. With the above, any separation logic formula φ can be regarded, *as is*, as a matching logic pattern of sort *Map*, and φ is valid in separation logic if and only if $M \models \varphi$ [20].

Thanks to the result above, we can reuse the vast body of recent separation logic work on formalizing and reasoning about heap patterns. For example, here is our matching logic definition of binary trees used in our experiments: a sort *Tree* with symbols $leaf : \to Tree$ and $node : Nat \times Tree \times Tree \to Tree$ to be used as constructors, together with a symbol $tree \in \Sigma_{Nat \times Tree,\, Map}$ constrained by $tree(0, leaf) = emp$ and $tree(x, node(n,t_1,t_2)) = \exists yz.x \mapsto [n,y,z] * tree(y,t_1) * tree(z,t_2))$. The symbol $\_ \mapsto [\_] : Nat \times Seq \rightharpoonup Map$ allocating sequences of numbers (defined using binary associative operation $\_,\_$ with identity ϵ) at consecutive locations can be defined with pattern equations $x \mapsto [\epsilon] = emp$ and $x \mapsto [a, S] = x \mapsto a * (x + 1) \mapsto [S]$. Using the sound and complete matching logic proof

system [20], we can now prove:

$$\begin{array}{ll} & 1 \mapsto 3 * 2 \mapsto 0 * 3 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * 9 \mapsto 0 \\ \rightarrow & \mathit{tree}(7, \mathit{node}(9, \mathit{node}(3, \mathit{leaf}, \mathit{leaf}), \mathit{leaf})) \end{array} \tag{2.2}$$

We can embed such logical reasoning within any structural context, because in matching logic we can represent arbitrary structure using symbols, like we build terms, this way easily and naturally globalizing local reasoning. Consider, e.g., the semantics of C [23, 24], whose configuration has more than 100 semantic cells like the ones in Figure 4.5. The semantic cells, written using symbols $\langle ... \rangle_{\mathsf{cell}}$, can be nested and their grouping is associative and commutative. A top cell $\langle ... \rangle_{\mathsf{cfg}}$ holds a subcell $\langle ... \rangle_{\mathsf{mem}}$ among many others. We can globalize the local reasoning above to the entire C configuration [20]:

$$\begin{array}{ll} & \langle \langle 1 \mapsto 3 * 2 \mapsto 0 * 3 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * 9 \mapsto 0 * m \rangle_{\mathsf{mem}}\ c \rangle_{\mathsf{cfg}} \\ \rightarrow & \langle \langle \mathit{tree}(7, \mathit{node}(9, \mathit{node}(3, \mathit{leaf}, \mathit{leaf}), \mathit{leaf})) * m \rangle_{\mathsf{mem}}\ c \rangle_{\mathsf{cfg}} \end{array} \tag{2.3}$$

Free variables $c : \mathit{Cfg}$ and $m : \mathit{Map}$ are universally quantified and represent the *memory frame* and the *configuration frame*.

2.1.2 Specifying Reachability

We recall the two types of reachability statements that our proof system in Section 2.1.3 derives: the one-path reachability rule [1], and the all-path reachability rule [5]. These are pairs of matching logic patterns, in this thesis written $\varphi \Rightarrow^{\exists} \varphi'$ and, respectively, $\varphi \Rightarrow^{\forall} \varphi'$ to distinguish them, capturing the partial correctness intuition: for any program configuration γ that matches φ, one path ($\exists$), respectively each path ($\forall$), derived using the operational semantics from γ either diverges or otherwise reaches a configuration γ' that matches φ'.

Let us fix the following: (1) an algebraic signature Σ, associated to some desired configuration syntax, with a distinguished sort *Cfg*, (2) a sort-wise infinite set *Var* of variables, and (3) a Σ-algebra $\mathcal{T}$, the *configuration model*, which may but need not be a term algebra. As usual, $\mathcal{T}_{\mathit{Cfg}}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*,

Definition 2.5. *[1] A **one-path reachability rule** is a pair $\varphi \Rightarrow^{\exists} \varphi'$, with φ and φ' patterns (may have free variables). Rule $\varphi \Rightarrow^{\exists} \varphi'$ is **weakly well-defined** iff for any $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ and $\rho : \mathit{Var} \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$, there exists $\gamma' \in \mathcal{T}_{\mathit{Cfg}}$ with $(\gamma', \rho) \models \varphi'$. A **reachability system** $\mathcal{S}$ is a set of reachability rules. $\mathcal{S}$ is **weakly well-defined** iff each rule is weakly well-defined. $\mathcal{S}$*

induces a ***transition system*** $(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ *on the configuration model:* $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *for* $\gamma, \gamma' \in \mathcal{T}_{cfg}$ *iff there is some rule* $\varphi \Rightarrow^{\exists} \varphi'$ *in* $\mathcal{S}$ *and some valuation* $\rho : Var \to \mathcal{T}$ *with* $(\gamma, \rho) \models \varphi$ *and* $(\gamma', \rho) \models \varphi'$. *A* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$***-path*** *is a finite sequence* $\gamma_0 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} ... \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_n$ *with* $\gamma_0,...,\gamma_n \in \mathcal{T}_{cfg}$. *A* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*-path is* ***complete*** *iff it is not a strict prefix of any other* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*-path.*

We assume an operational semantics is a set of reduction rules "$l \Rightarrow r$ where b", with l and r configuration terms and b a boolean side condition constraining the variables of l, r. Operational semantics styles using only such rules include evaluation contexts [25], the CHAM [26], and $\mathbb{K}$ [27]. Several large languages have been given semantics in such styles, including the ones used in this thesis: C, Java, JavaScript. The reachability proof system below works with any set of rules of this form, being agnostic to the particular semantics style.

A rule "$l \Rightarrow r$ where b" states that a ground configuration γ which is an instance of l and satisfies condition b reduces to an instance γ' of r. Matching logic can express terms with constraints as particular patterns: $l \wedge b$ is satisfied by exactly such γ. Thus, such a semantics is a particular weakly well-defined reachability system $\mathcal{S}$ with rules of the form "$l \wedge b \Rightarrow^{\exists} r$". The weakly well-defined condition on $\mathcal{S}$ guarantees that if γ matches the left-hand-side of a rule in $\mathcal{S}$, then the respective rule induces an outgoing transition from γ. The transition system induced by $\mathcal{S}$ describes precisely the behavior of any program in any given state. See Section 2.2.1 (and particularly Figure 2.2) for a sample operational semantics based on evaluation contexts for the IMP language and an example of how we we view the semantics rules as one-path reachability rules.

Definition 2.6. *[1] A one-path reachability rule* $\varphi \Rightarrow^{\exists} \varphi'$ *is satisfied,* $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$*, iff for all* $\gamma \in \mathcal{T}_{cfg}$ *and* $\rho : Var \to \mathcal{T}$ *such that* $(\gamma, \rho) \models \varphi$*, there is either a* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*-path from* γ *to some* γ' *such that* $(\gamma', \rho) \models \varphi'$*, or there is a diverging execution* $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_2 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \cdots$ *from* γ.

We next recall the all-path variant from [5].

Definition 2.7. *[5] With the notation in Definition 2.5, an* ***all-path reachability rule*** *is a pair* $\varphi \Rightarrow^{\forall} \varphi'$*. Rule* $\varphi \Rightarrow^{\forall} \varphi'$ *is satisfied,* $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$*, iff for all complete* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*-paths* τ *starting with* $\gamma \in \mathcal{T}_{cfg}$ *and for all* $\rho : Var \to \mathcal{T}$ *such that* $(\gamma, \rho) \models \varphi$*, there exists some* $\gamma' \in \tau$ *such that* $(\gamma', \rho) \models \varphi'$.

The semantic validity of reachability rules captures the same intuition of partial correctness as Hoare logic, but in more general terms of reachability. If the language defined by $\mathcal{S}$ is deterministic, then the notions of one-path and all-path above coincide. A Hoare triple describes the resulting state after the execution finishes, so it corresponds

to a reachability rule where the right-hand-side contains no remaining code. However, reachability rules are strictly more expressive than Hoare triples, as they can also specify intermediate configurations (the code in the right-hand-side need not be empty). Like Hoare triples, reachability rules can only specify properties of complete paths (terminating execution paths). We do not discuss total correctness; however, one can use existing techniques to break reasoning about a non-terminating program into reasoning about its terminating components. Crucially, reachability rules provide a unified representation for both semantic rules and program specifications. This makes them perfectly suitable for our goal to obtain program verifiers from operational semantics.

The correctness property of a racing increment program in the context of a simple imperative language can be specified by

$$\begin{array}{l} \langle\langle \texttt{x = x + 1; || x = x + 1;} \rangle_{\mathsf{code}} \ \langle \mathsf{x} \mapsto m \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \\ \Rightarrow^{\forall} \exists n \ (\langle\langle\rangle_{\mathsf{code}} \ \langle \mathsf{x} \mapsto n \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \\ \qquad \wedge \ (n = m +_{Int} 1 \vee n = m +_{Int} 2) \end{array} \tag{2.4}$$

which states that every terminating execution reaches a state where execution of both threads is complete and the value of x has increased by 1 or 2 (this code has a race). As mentioned before, for deterministic programs, the one-path and the all-path reachability coincide. For example, the correctness property of a program computing the sum of all the natural numbers strictly less than n would be

$$\begin{array}{l} \langle\langle \texttt{s = 0; while(--n) s = s + n;} \rangle_{\mathsf{code}} \\ \ \langle \mathsf{n} \mapsto n, \ \mathsf{s} \mapsto s \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \ \wedge \ n \geq_{Int} 1 \\ \Rightarrow^{\exists} \ \langle\langle\rangle_{\mathsf{code}} \ \langle \mathsf{n} \mapsto 0, \ \mathsf{s} \mapsto n *_{Int} (n -_{Int} 1) \ /_{Int} 2 \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \end{array} \tag{2.5}$$

See Section 2.2.3 (especially Figure 2.5) for a full reachability logic proof of this rule.

2.1.3 Reachability Proof System

Figure 2.1 shows our proof system for both one-path and all-path reachability, which we refer to as *reachability logic*. It combines the one-path reachability proof system in [1] with the all-path one in [5], taking advantage of recent developments in matching logic in [20]. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Theorem 2.1) guarantees that $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ if $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$ is derivable, where $Q \in \{\forall, \exists\}$. The proof system derives more general sequents "$\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'$", where $\mathcal{A}$ and $\mathcal{C}$ are sets of reachability rules. The rules in $\mathcal{A}$ are called *axioms* and

STEP :

$$\frac{\models \varphi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l).\varphi_l \quad \models ((\varphi \wedge \varphi_l) \neq \bot_{Cfg}) \wedge \varphi_r \rightarrow \varphi' \qquad \text{for each } \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{\forall} \varphi'}$$

AXIOM :

$$\frac{\varphi \Rightarrow^{Q} \varphi' \in \mathcal{S} \cup \mathcal{A} \qquad \psi \text{ is FOL formula (logical frame)}}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \wedge \psi \Rightarrow^{Q} \varphi' \wedge \psi}$$

REFLEXIVITY :

$$\frac{\cdot}{\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi}$$

TRANSITIVITY :

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

CONSEQUENCE :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow^{Q} \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_2}$$

CASE ANALYSIS :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi \qquad \mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow^{Q} \varphi}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi}$$

ABSTRACTION :

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

CIRCULARITY :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'}$$

Figure 2.1: Proof system for reachability. We assume the free variables of $\varphi_l \Rightarrow^{\exists} \varphi_r$ in the STEP proof rule are fresh (e.g., disjoint from those of $\varphi \Rightarrow^{\forall} \varphi'$). Here $Q \in \{\forall, \exists\}$.

rules in $\mathcal{C}$ are called *circularities*. If $\mathcal{A}$ or $\mathcal{C}$ does not appear in a sequent, it is empty: $\mathcal{S} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'$ is shorthand for $\mathcal{S}, \emptyset \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'$, and $\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi'$ is shorthand for $\mathcal{S}, \mathcal{A} \vdash_{\emptyset} \varphi \Rightarrow^{Q} \varphi'$. Initially, $\mathcal{A}$ and $\mathcal{C}$ are empty. Note that "$\rightarrow$" in STEP and CONSEQUENCE denotes implication.

The intuition is that the reachability rules in $\mathcal{A}$ can be assumed valid, while those in $\mathcal{C}$ have been postulated but not yet justified. After making progress from φ (at least one derivation by STEP or by AXIOM), the rules in $\mathcal{C}$ become (coinductively) valid and can be used in derivations by AXIOM. During the proof, circularities can be added to $\mathcal{C}$ via CIRCULARITY, flushed into $\mathcal{A}$ by TRANSITIVITY, and used via AXIOM. The semantics of

sequent $\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'$ (read "$\mathcal{S}$ with axioms $\mathcal{A}$ and circularities $\mathcal{C}$ proves $\varphi \Rightarrow^{Q} \varphi'$") is: $\varphi \Rightarrow^{Q} \varphi'$ holds if the rules in $\mathcal{A}$ hold and those in $\mathcal{C}$ hold after taking at least one step from φ in the transition system $(\Rightarrow^{\mathcal{T}}_{\mathcal{S}}, \mathcal{T})$. Moreover, if $\mathcal{C} \neq \emptyset$ then φ reaches φ' after at least one step on all complete paths when $Q = \forall$ and on at least one path when $Q = \exists$. As a consequence of this definition, any rule $\varphi \Rightarrow^{Q} \varphi'$ derived by Circularity has the property that φ reaches φ' after at least one step, due to Circularity having a prerequisite $\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'$ (with a non-empty set of circularities). We next discuss the proof rules.

Step derives a sequent where φ reaches φ' in one step on all paths. The first premise ensures any configuration matching φ matches the left-hand-side φ_l of some rule in $\mathcal{S}$ and thus, as $\mathcal{S}$ is weakly well-defined, can take a step: if $(\gamma, \rho) \models \varphi$ then there is a $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and a valuation ρ' of the free variables of φ_l s.t. $(\gamma, \rho') \models \varphi_l$, and thus γ has at least one $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor generated by $\varphi_l \Rightarrow^{\exists} \varphi_r$. The second premise ensures that each $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor of a configuration matching φ matches φ': if $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ and γ matches φ then there is some rule $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and $\rho : \mathit{Var} \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi \wedge \varphi_l$ and $(\gamma', \rho) \models \varphi_r$; then the second part implies γ' matches φ'.

Axiom applies a trusted rule. Reflexivity and Transitivity capture the closure properties of the reachability relation. Reflexivity requires $\mathcal{C}$ empty to ensure that rules derived with non-empty $\mathcal{C}$ take at least one step. Transitivity enables the circularities as axioms for the second premise, since if $\mathcal{C}$ is not empty, the first premise is guaranteed to take a step. Consequence, Case Analysis and Abstraction are adapted from Hoare logic. Ignoring circularities, these seven proof rules are the formal infrastructure for symbolic execution.

Circularity has a coinductive nature, allowing us to make new circularity claims. We typically make such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If there is a derivation of the claim using itself as a circularity, then the claim holds. This would obviously be unsound if the new assumption was available immediately, but requiring progress (taking at least on step before circularities can be used) ensures that only diverging executions correspond to endless invocation of a circularity. Formally, we have the following result:

Theorem 2.1. *The proof system in Figure 2.1 is **sound**: if $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$ then $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ ($Q \in \{\exists, \forall\}$). Under some mild assumptions, it is **relatively complete**: given an oracle for $\mathcal{T}$, if $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ then $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$.*

The proof for the all-path case is available in [5], and for the one-path case in [3]. When considering the completeness of program verification logics, notice that if the logic for

$$
\begin{array}{rl}
\mathit{Int} ::= & \text{Arbitrarily large integers} \\
\mathit{Var} ::= & \text{Arbitrarily variables (identifiers)} \\
\mathit{Exp} ::= & \mathit{Int} \mid \mathit{Exp} \texttt{ + } \mathit{Exp} \mid \mathit{Exp} \texttt{ - } \mathit{Exp} \mid \texttt{--}\mathit{Var} \\
\mathit{Stmt} ::= & \texttt{\{\}} \mid \texttt{\{}\mathit{Stmt}\texttt{\}} \mid \mathit{Var} \texttt{ = } \mathit{Exp}\texttt{;} \mid \mathit{Stmt}\ \mathit{Stmt} \\
& \mid \texttt{if(}\mathit{Exp}\texttt{)}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\
& \mid \texttt{while(}\mathit{Exp}\texttt{)}\ \mathit{Stmt} \\
C ::= & \Box \mid C\ \mathit{Stmt} \mid \mathit{Var} \texttt{ = } C\texttt{;} \mid C \texttt{ + } \mathit{Exp} \mid \mathit{Exp} \texttt{ + } C \\
& \mid \texttt{if(}C\texttt{)}\mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}
\end{array}
$$

$$
\begin{array}{l}
\langle\langle C[X \Rightarrow I]\rangle_{\mathsf{code}}\ \langle X \mapsto I, \sigma\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}} \\
I_1 \texttt{ + } I_2 \Rightarrow I_1 +_{\mathit{Int}} I_2 \\
I_1 \texttt{ - } I_2 \Rightarrow I_1 -_{\mathit{Int}} I_2 \\
\langle\langle C[\texttt{--}X \Rightarrow I -_{\mathit{Int}} 1]\rangle_{\mathsf{code}}\ \langle X \mapsto (I \Rightarrow I -_{\mathit{Int}} 1), \sigma\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}} \\
\texttt{\{\}}\ S \Rightarrow S \\
\texttt{\{}S\texttt{\}} \Rightarrow S \\
\langle\langle C[X \texttt{ = } I\texttt{;} \Rightarrow \texttt{\{\}}]\rangle_{\mathsf{code}}\ \langle X \mapsto (I' \Rightarrow I), \sigma\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}} \\
\texttt{if(0)}\ S_1\ \texttt{else}\ S_2 \Rightarrow S_2 \\
\texttt{if(}I\texttt{)}\ S_1\ \texttt{else}\ S_2 \Rightarrow S_1 \quad \text{where } I \neq_{\mathit{Int}} 0 \\
\texttt{while(}E\texttt{)}\ S \Rightarrow \texttt{if(}E\texttt{)\{}S\ \texttt{while(}E\texttt{)}\ S\texttt{\}}\ \texttt{else\{\}}
\end{array}
$$

Figure 2.2: Reduction semantics of a simple imperative language with auto-decrement. Configurations have the form $\langle\langle ... \rangle_{\mathsf{code}}\ \langle ... \rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}$. C ranges over evaluation contexts; X over variables; I, I', I_1, I_2 over integers; σ over states; S, S_1, S_2 over statements; and E over expressions. $\mathit{Context}[t_1 \Rightarrow t_1', ..., t_n \Rightarrow t_n']$ is shorthand for $\mathit{Context}[t_1, ..., t_n] \Rightarrow \mathit{Context}[t_1', ..., t_n']$, and $t \Rightarrow t'$ is shorthand for $\langle\langle C[t \Rightarrow t']\rangle_{\mathsf{code}}\ \langle\sigma\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}$.

specifying state properties (in this case, matching logic) is undecidable, then the entire program verification logic (in this case, reachability logic) is undecidable. By relative completeness, we prove the completeness of the proof system in Figure 2.1 assuming we can decide any matching logic formula in $\mathcal{T}$, which means that any undecidability comes from $\mathcal{T}$ and is unavoidable. This theorem generalizes similar results from Hoare logic, but in a language-independent setting.

2.2 REACHABILITY LOGIC VS. HOARE LOGIC

Here we briefly compare reachability logic (Section 2.1.3) with Hoare logic by means of a simple example, aiming to convey the message that verification using reachability logic is not harder than using Hoare logic, even when done manually.

$$\{\psi[E/X]\}\ \ X = E;\ \ \{\psi\} \qquad \text{(HL-Asgn)}$$

$$\frac{\{\psi_1\}\ \ S_1\ \ \{\psi_2\} \qquad \{\psi_2\}\ \ S_2\ \ \{\psi_3\}}{\{\psi_1\}\ \ S_1\ S_2\ \ \{\psi_3\}} \qquad \text{(HL-Seq)}$$

$$\frac{\models \psi_1' \to \psi_1 \qquad \{\psi_1\}\ \ S\ \ \{\psi_2\} \qquad \models \psi_2 \to \psi_2'}{\{\psi_1'\}\ \ S\ \ \{\psi_2'\}} \qquad \text{(HL-Cnsq)}$$

$$\frac{\{\psi_{inv} \wedge E \neq_{Int} 0\}\ \ S\ \ \{\psi_{inv}\}}{\{\psi_{inv}\}\ \ \texttt{while(}E\texttt{)}S\ \ \{\psi_{inv} \wedge E =_{Int} 0\}} \qquad \text{(HL-While)}$$

Figure 2.3: Part of Hoare logic proof system

2.2.1 The Program and the Language

Consider the following snippet, say `SUM`, part of a C-like program summing up the natural numbers smaller than `n`:

```
s = 0;
while(--n) s = s + n;
```

Assume a simplified language whose loops cannot break/return/jump, whose integers are arbitrarily large, and without local variables (so blocks are used for grouping only). Figure 2.2 shows a reduction-style executable semantics of the needed language fragment; with the notation explained in the caption of Figure 2.2, the semantics consists of ten reduction rules between configuration terms. Each of these rules can be regarded as a one-path reachability rule, with side conditions as constraints on the left-hand-side pattern of the rule. For example, the second rule for the conditional statement becomes the following one-path reachability rule:

$$\begin{array}{ll} & \langle\langle C[\texttt{if(}I\texttt{)}\ S_1\ \texttt{else}\ S_2]\rangle_{\mathsf{code}}\ \langle\sigma\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}\ \wedge\ I \neq_{Int} 0 \\ \Rightarrow^{\exists} & \langle\langle C[S_1]\rangle_{\mathsf{code}}\ \langle\sigma\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}} \end{array} \qquad (2.6)$$

Mathematical domain operations ($+_{Int}$, etc.) are subscripted with *Int* to distinguish them from the language constructs.

2.2.2 Hoare Logic Proof

The Hoare logic precondition ψ_{pre} is $\mathsf{n} =_{Int} n \wedge n \geq_{Int} 1$, and the postcondition ψ_{post} is $\mathsf{n} =_{Int} 0 \wedge \mathsf{s} =_{Int} n *_{Int} (n -_{Int} 1)\ /_{Int} 2$. The variable n using italic font is introduced to capture the original value of the program variable `n`, so that we can use it to express the

$$\text{HL-While(7), HL-Cnsq}\ \dfrac{\text{HL-Seq(8,9)}\ \dfrac{\text{HL-Asgn}\ \dfrac{}{\{\psi_2\}\ \texttt{s=s+n;}\ \{\psi_3\}}\ (8) \quad \dfrac{}{\{\psi_3\}\ \texttt{n=n-1;}\ \{\psi_{inv}\}}\ (9)\ \text{HL-Asgn, HL-Cnsq}}{\{\psi_2\}\ \texttt{s=s+n;n=n-1;}\ \{\psi_{inv}\}}\ (7)}{\{\psi_{inv}\}\ \texttt{LOOP'}\ \{\psi_{post}\}}\ (6)$$

$$\text{HL-Seq(4,5)}\ \dfrac{\text{HL-Asgn}\ \dfrac{}{\{\psi_{pre}\}\ \texttt{s=0;}\ \{\psi_{pre} \wedge \texttt{s} =_{Int} 0\}}\ (4) \quad \dfrac{}{\{\psi_{pre} \wedge \texttt{s} =_{Int} 0\}\ \texttt{n=n-1;}\ \{\psi_1\}}\ (5)\ \text{HL-Asgn}}{\{\psi_{pre}\}\ \texttt{s=0;n=n-1;}\ \{\psi_1\}}\ (3)$$

$$\text{HL-Cnsq(3)}\ \dfrac{\{\psi_{pre}\}\ \texttt{s=0;n=n-1;}\ \{\psi_1\}}{\{\psi_{pre}\}\ \texttt{s=0;n=n-1;}\ \{\psi_{inv}\}}\ (2)$$

$$\text{HL-Seq(2,6)}\ \dfrac{\{\psi_{pre}\}\ \texttt{s=0;n=n-1;}\ \{\psi_{inv}\} \qquad \{\psi_{inv}\}\ \texttt{LOOP'}\ \{\psi_{post}\}}{\{\psi_{pre}\}\ \texttt{SUM'}\ \{\psi_{post}\}}\ (1)$$

(a) Hoare logic proof of SUM'

$$\begin{array}{lll}
\psi_{pre} & \equiv \texttt{n} =_{Int} n \ \wedge\ n \geq_{Int} 1 & \psi_{inv} \equiv \texttt{n} \geq_{Int} 0 \ \wedge\ s =_{Int} \Sigma_{\texttt{n}+_{Int}1}^{n-_{Int}1} \\
\psi_{post} & \equiv \texttt{n} =_{Int} 0 \ \wedge\ \texttt{s} =_{Int} n *_{Int} (n -_{Int} 1) \ /_{Int} 2 & \texttt{LOOP'} \equiv \texttt{while(n)\{s = s + n; n = n - 1;\}} \\
\psi_1 & \equiv \texttt{n} =_{Int} n -_{Int} 1 \ \wedge\ n \geq_{Int} 1 \ \wedge\ \texttt{s} =_{Int} 0 & \psi_2 \equiv \texttt{n} >_{Int} 0 \ \wedge\ \texttt{s} =_{Int} \Sigma_{\texttt{n}+_{Int}1}^{n-_{Int}1} \\
\Sigma_i^j & \equiv (j +_{Int} i) *_{Int} (j -_{Int} i +_{Int} 1) \ /_{Int} 2 & \psi_3 \equiv \texttt{n} >_{Int} 0 \ \wedge\ \texttt{s} =_{Int} \Sigma_{\texttt{n}}^{n-_{Int}1}
\end{array}$$

(b) Notations for Hoare logic proof

Figure 2.4: Hoare logic proof of SUM. The numbers appearing in the side of each proof steps are not part of the proofs, but only references to be used in the explanation of the proofs in Section 2.2.2.

value of s in the post-condition (the loop changes the value of n). A typical simplification in hand proofs using Hoare logic is to collapse expression constructs in the language with operations in the underlying domain, e.g., + with $+_{Int}$. Tools, however, distinguish the two and implement translations from the former to the latter; e.g., + may be 32-bit while $+_{Int}$ may be arbitrary precision, or + may have a concurrent semantics allowing all the interleavings of its arguments' behaviors, etc. Since our language is simple, we do this translation by hand on the fly, but for clarity we use mathematical operations in formulae.

To derive the Hoare triple $\{\psi_{pre}\}$ SUM $\{\psi_{post}\}$, we need to find a loop invariant ψ_{inv} and then use the invariant proof rule as shown in Figure 2.3. The loop condition is inserted within formulae. Thus, when verifying programs using Hoare logic, expressions cannot have side effects; programs need to be modified to isolate side effects from computed values of expressions, which is inherently language-specific.

$$\textsc{Transitivity}\ (11,12)\ \frac{\textsc{Axiom}^{+}\ \dfrac{\vdots}{\mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_2 \wedge n' >_{Int} 1 \Rightarrow^{\exists} \varphi_3}\ (11) \qquad \dfrac{}{\mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_3 \Rightarrow^{\exists} \varphi_{post}}\ (12)\ \textsc{Axiom}(\mu)}{\mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_2 \wedge n' >_{Int} 1 \Rightarrow^{\exists} \varphi_{post}}\ (9)$$

$$\textsc{Case}(9,10)\ \frac{\mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_2 \wedge n' >_{Int} 1 \Rightarrow^{\exists} \varphi_{post} \qquad \textsc{Axiom}^{+}\ \dfrac{\vdots}{\mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_2 \wedge n' \leq_{Int} 1 \Rightarrow^{\exists} \varphi_{post}}\ (10)}{\mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_2 \Rightarrow^{\exists} \varphi_{post}}\ (8)$$

$$\textsc{Transitivity}\ (7,8)\ \frac{\textsc{Axiom}\ \dfrac{}{\mathcal{S}, \emptyset \vdash_{\{\mu\}} \varphi_{inv} \Rightarrow^{\exists} \varphi_2}\ (7) \qquad \mathcal{S}, \{\mu\} \vdash_{\emptyset} \varphi_2 \Rightarrow^{\exists} \varphi_{post}}{\mathcal{S}, \emptyset \vdash_{\{\mu\}} \varphi_{inv} \Rightarrow^{\exists} \varphi_{post}}\ (6)$$

$$\textsc{Consequence}\ (4)\ \frac{\textsc{Axiom}\ \dfrac{}{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \varphi_1}\ (4)}{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \exists n'.\varphi_{inv}}\ (2) \qquad \frac{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{inv} \Rightarrow^{\exists} \varphi_{post}}{\mathcal{S}, \emptyset \vdash_{\emptyset} \exists n'.\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}}\ (3)\ \textsc{Abstraction}\ (5)$$

$$\textsc{Transitivity}\ (2,3)\ \frac{\textsc{Consequence}\ (4)\ \dfrac{\textsc{Axiom}\ \dfrac{}{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \varphi_1}\ (4)}{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \exists n'.\varphi_{inv}}\ (2) \qquad \dfrac{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{inv} \Rightarrow^{\exists} \varphi_{post}}{\mathcal{S}, \emptyset \vdash_{\emptyset} \exists n'.\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}}\ (3)\ \textsc{Abstraction}\ (5)}{\mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \varphi_{post}}\ (1)$$

(a) Reachability logic proof of SUM

$$\begin{array}{lcl}
\varphi_{pre} & \equiv & \langle\langle \mathsf{SUM} \rangle_{\mathsf{code}}\ \langle \mathsf{n} \mapsto n,\ \mathsf{s} \mapsto s \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}}\ \wedge\ n \geq_{Int} 1 \\
\varphi_{post} & \equiv & \langle\langle \rangle_{\mathsf{code}}\ \langle \mathsf{n} \mapsto 0,\ \mathsf{s} \mapsto n *_{Int} (n -_{Int} 1)\ /_{Int} 2 \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \\
\varphi_{inv} & \equiv & \langle\langle \mathsf{LOOP} \rangle_{\mathsf{code}}\ \langle \mathsf{n} \mapsto n',\ \mathsf{s} \mapsto \Sigma_{n'}^{n -_{Int} 1} \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}}\ \wedge\ n' \geq_{Int} 1 \\
\mu & \equiv & \varphi_{inv} \Rightarrow^{\exists} \varphi_{post} \\
\varphi_1 & \equiv & \langle\langle \mathsf{LOOP} \rangle_{\mathsf{code}}\ \langle \mathsf{n} \mapsto n,\ \mathsf{s} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}}\ \wedge\ n \geq_{Int} 1 \\
\varphi_2 & \equiv & \langle\langle \mathsf{IF} \rangle_{\mathsf{code}}\ \langle \mathsf{n} \mapsto n',\ \mathsf{s} \mapsto \Sigma_{n'}^{n -_{Int} 1} \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}}\ \wedge\ n' \geq_{Int} 1 \\
\varphi_3 & \equiv & \langle\langle \mathsf{LOOP} \rangle_{\mathsf{code}}\ \langle \mathsf{n} \mapsto n' -_{Int} 1, \mathsf{s} \mapsto \Sigma_{n' -_{Int} 1}^{n -_{Int} 1} \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \wedge\ n' >_{Int} 1 \\
\mathtt{LOOP} & \equiv & \mathtt{while(--n)\{\ s = s + n;\ \}} \\
\mathtt{IF} & \equiv & \mathtt{if(--n)\{s = s + n;\ LOOP\}\ else\ \{\}}
\end{array}$$

(b) Notations for reachability logic proof

Figure 2.5: Reachability logic proofs of SUM. The numbers appearing in the side of each proof steps are not part of the proofs, but only references to be used in the explanation of the proofs in Section 2.2.3.

For example, VCC [18] expands the loop above into one having more than a dozen statements in its translation to Boogie [28]. To keep it human readable, we manually modify SUM in a minimal (but adhoc) way to the equivalent `SUM', s = 0; n = n - 1; while(n) { s = s + n; n = n - 1; }`, which can be verified using conventional Hoare logic. The proof can be derived as shown in Figure 2.4. Step (1) factors the proof using the loop invariant ψ_{inv}. First we show using HL-Asgn twice (4,5) followed by HL-Seq (3)

that ψ_1 is reachable before the loop (3), which implies the invariant holds when the loop is reached (2). To prove the invariant, we use HL-While at (6), which generates the proof obligation (7) for the loop body, noticing that ψ_2 is logically equivalent to $\psi_{inv} \wedge \mathtt{n} \neq_{Int} 0$. The rest follows by two applications of HL-Asgn at (8,9), followed by an HL-Seq which concludes.

2.2.3 Reachability Logic Proof

Let us now verify the original program `SUM` (with `--n` in the `while` condition) using the generic reachability logic instantiated with the executable semantics of the language. Notice that we only transformed the code in Section 2.2.2 because the Hoare logic proof rule for `while` assumes there are no side-effects in the condition.

Let $\mathcal{S}$ be the reachability logic system in Figure 2.2, where each rule is regarded as a one-path rule as explained in Section 2.2.1. The reachability logic rule stating the correctness of `SUM` is $\varphi_{pre} \Rightarrow^{\exists} \varphi_{post}$, which can be derived as shown in Figure 2.5. Step (1) factors the proof using the loop invariant existentially quantified in all its new (mathematical) variables. To show that the invariant holds when the loop is reached (2), we "execute" the initial pattern φ_{pre} with the operational semantics rule of assignment (4), reaching pattern φ_1, which implies (in matching logic) the existentially quantified invariant. To prove the existentially quantified invariant, thanks to Abstraction we first eliminate the existential quantifier (3) and then, expecting a circular behavior of the loop, we add the proof obligation as a circularity (5). The rest is just symbolic execution of the loop body using the executable semantics and giving priority to the circularity when it matches. Specifically, the loop is unrolled using the executable semantics of `while` (7), then a case analysis is initiated on whether the value held by `n` is larger than 1 or not (8), and φ_{post} is indeed reached on both paths (9,10). The circularity is used on the positive branch only (12), as expected. In this proof we do not mention the Consequence steps that change a formula into an equivalent formula (i.e. φ_2 into $\varphi_2 \wedge (n' \leq_{Int} 1 \vee n' >_{Int} 1)$).

2.2.4 Discussion

Forty-five years of Hoare logic cannot be taken lightly. We do not expect the reader to immediately agree with us that the reachability logic proof above is more intuitive than the Hoare logic proof. We do, however, urge the reader to consider the main practical benefits of the reachability logic proof: it used the executable semantics of the programming language *unchanged* and only a fixed set of language-independent proof

rules, without requiring any other semantics to be crafted or the program to be modified in order to be verifiable.

These benefits cannot be taken lightly either, especially when certifiable verification is a concern. The current state of the art in certifiable verification is to define an alternative Hoare logic of the language (or a corresponding VC generator) and prove its soundness w.r.t. the trusted operational semantics; similarly, the transformed program needs to be in the correct relationship with the original program (the transformed program may lose behaviors) also using the operational semantics. These tasks are quite tedious when real-world languages are concerned. Besides, they need to be maintained as the language evolves, or as bugs are found and fixed in the operational semantics, or even as the operational semantics is refactored. For example, the semantics of C [24] has over 2,500 rules and according to the repository history it has been updated at a rate of two commits per day over the last 3 years. In this light, one can regard the reachability logic proof system as an effective mechanism to turn an operational semantics into a corresponding axiomatic semantics.

2.3 REACHABILITY LOGIC THEOREM PROVER IN K

We discuss our novel implementation of the $\mathbb{K}$ verification infrastructure, depicted in Figure 4.1, based on the language-independent proof system in Figure 2.1. Our framework takes an operational semantics defined in $\mathbb{K}$ [27] as a parameter and uses it to automatically derive program correctness properties. In other words, our verification infrastructure *automatically* generates a program verifier from the semantics, which is *correct-by-construction* w.r.t. the semantics. As discussed in Section 2.1.2, we view a semantics as a set of reachability rules $l \wedge b \Rightarrow^{\exists} r$. A major difficulty in a language-independent setting is that standard language features relevant to verification, like control flow or memory access, are not explicit, but rather implicit (defined through the semantics).

The generated program verifier proves a set of user provided reachability rules, representing the program correctness specifications of the code being verified, typically one for each recursive function and loop. For the sake of automation, the rules have the more restrictive form $\pi \wedge \psi \Rightarrow^{\forall} \pi' \wedge \psi'$, with $\pi \wedge \psi$ and $\pi' \wedge \psi'$ conjunctive patterns. A *conjunctive pattern* is a formula $\pi \wedge \psi$ with π a program configuration term with variables, and ψ a formula without any configuration terms. We use all-path rules for specifications to capture some of the local non-determinism (e.g. the non-deterministic C expression

```
1  Q := successors(φ)
2  if Q is empty and  ⊭ φ → φ′ then fail
3  while Q not empty
4    pop φ_c from Q
5    if  ⊨ φ_c → φ′ continue
6    else if ∃σ with ⊨ φ_c → σ(φ_l) for φ_l ⇒^∀ φ_r ∈ C
7      add σ(φ_r) ∧ frame(φ_c) to Q
8    else
9      Q′ := successors(φ_c)
10     if Q′ is empty then fail
11     add all Q′ to Q
```

Figure 2.6: Reachability logic theorem prover algorithm

evaluation order). Section 4.1.1 shows examples of specifications. As discussed there, we use conventions already supported by $\mathbb{K}$ to have more compact specifications.

The generated program verifier is fully automated. The user only provides the program correctness specifications. The verifier uses the operational semantics for symbolic execution and performs matching logic reasoning automatically. Specifically, to prove a set $\mathcal{C}$ of rules between conjunctive patterns, it uses the algorithm in Figure 2.6 to derive

$$\mathcal{S}, \emptyset \ \vdash_{\mathcal{C}} \varphi \Rightarrow^{\forall} \varphi' \tag{2.7}$$

for each $\varphi \Rightarrow^{\forall} \varphi' \in \mathcal{C}$, where $\mathit{successors}(\varphi)$ returns, as a set, the disjunction of conjunctive patterns representing the one-step successors of φ (see Section 2.3.1), σ is a substitution, and $\mathit{frame}(\pi \wedge \psi)$ returns ψ. The algorithm uses a queue Q of conjunctive patterns, which is initialized with the one-step successors of φ (lines 1-2). At each step the main loop (lines 3-11) processes a conjunctive pattern φ_c from Q. If φ_c implies the postcondition φ' then verification succeeds on this execution path (line 5). If φ_c matches the left-hand-side of a specification rule in $\mathcal{C}$ then the respective rule is used to summarize its corresponding code (lines 6-7). Finally, if none of the cases above hold, add all one-step successors of φ_c to Q (lines 9-11). Using a specification is preferred over the operational semantics. If there are no *successors* (lines 2 and 10), the verification fails, as some concrete configurations satisfying the formula may not have a successor (e.g. a dereferenced pointer may be `NULL` in C). Our algorithm is incomplete, i.e., `fail` means that the specification cannot be verified successfully, not that it is violated by the code. Each pattern is simplified using function/abstraction definitions and lemmas before being added to Q.

The algorithm automates the proof system in Figure 2.1. Implementing the computation of multiple steps of symbolic execution across multiple paths with a queue corresponds

to Transitivity and Reflexivity. Computing *successors* (line 1 and line 9) corresponds to Step, and splitting the subsequent disjunction to Case Analysis. Finishing an execution path (line 5) corresponds to Consequence. Using a specification rule (lines 6-7) corresponds to Consequence, Abstraction, and Axiom. Since Q is initialized with the successors of φ, a step of Transitivity already moved $\mathcal{C}$ to $\mathcal{A}$. Consequence and Abstraction simplify a pattern before adding it to Q. We use Circularity on the set $\mathcal{C}$ before the beginning of the algorithm. This is sound because in line 1, we compute the successors of φ outside the `while` loop, which amounts to Step + Transitivity, and then we use the rules in $\mathcal{C}$ with Axiom in the body of the loop in line 6. Thus, we can conclude that all the rules in $\mathcal{C}$ hold.

Our verification infrastructure is implemented in Java, and uses Z3 [29]. It consists of approximately 30,000 non-blank lines of code, and it took about 2.5 man-years.

2.3.1 Symbolic Execution

Language-independent symbolic execution is complicated by the absence of explicit control flow statements, which are language specific. We handle control flow statements by noticing they are generally unifiable with the left-hand-sides of several semantics rules. Consider the C code "`if (`b`) x = 1; else x = 0;`". It does not match the left-hand-side of any of the two semantics rules of `if` (they require the condition to be either the constant true or the constant false [23]), but it is unifiable with the left-hand-sides of both rules. We achieve symbolic execution by performing *narrowing* [30] (i.e., rewriting with unification instead of matching). When using the semantics rules, taking steps of rewriting on a ground configuration yields concrete execution, while taking steps of narrowing yields symbolic execution.

We compute *successors*$(\pi \wedge \psi)$ using *unification modulo theories*. We distinguish several theories (e.g. booleans, integers, sequences, sets, maps, etc) that the underlying SMT solver can reason about. Specifically, we unify $\pi \wedge \psi$ with the left-hand-side of a semantics rule $\pi_l \wedge \psi_l$. We begin with the syntactic unification of π and π_l. Upon encountering corresponding subterms (π' in π and π'_l in π_l) which are both terms of one of the theories above, we record an equality $\pi' = \pi'_l$ rather than decomposing the subterms further (if one is in a theory, and the other one is in a different theory or is not in any theory, unification fails). If this stage is successful, we end up with a conjunction ψ_u of equalities, some having a variable in one side and some with both sides in one of the theories. Then we check the satisfiability of $\psi \wedge \psi_u \wedge \psi_l$ using the SMT solver. If it is satisfiable, then $\pi_r \wedge \psi \wedge \psi_u \wedge \psi_l \wedge \psi_r$ is a successor of $\pi \wedge \psi$, where $\pi_r \wedge \psi_r$ is the right-hand-side of the

semantics rule. Then *successors* is the disjunction of $\varphi_r \wedge \psi_u \wedge \psi \wedge \psi_l$ over all rules in $\mathcal{S}$ and all unification solutions ψ_u. While in general this disjunction may not be finite [5], in practice it is finite for the examples we considered. Intuitively, "collecting" the constraints $\psi_u \wedge \psi_l \wedge \psi_r$ is similar to collecting the path constraint in traditional symbolic execution (but is done in a language-generic manner). For instance, the `if` case above, results in collecting the constraints $b = \texttt{true}$ and $b = \texttt{false}$. Notice that $\models \varphi \wedge \varphi_l \neq \bot_{Cfg}$ is satisfiable iff φ and φ_l are unifiable. Thus, we are sound by Step.

Several optimizations improve performance; we mention two. First, as the semantics of a real-world language consists of thousands of rules, the verifier uses an indexing algorithm to determine which rules may apply. Second, the verifier caches partial unification results, e.g., for each semantics rule, the verifier caches pairs of terms (t_1, t_2) that fail to unify with t_2 a subterm of the left-hand-side of the rule.

2.3.2 Matching Logic Prover

Matching logic reasoning is used in three cases in our algorithm: (1) to finish the proof (line 5), (2) to use a specification rule to summarize a code fragment (line 6), and (3) to simplify a pattern (before adding it to Q).

As discussed in Section 2.1.1, we use recursively-defined heap abstractions to specify the correctness of programs manipulating lists and trees in the heap. Such definitions exploit the recursive nature of the data-structures , e.g.,

$$\begin{aligned} &\text{tree}(x, \text{node}(n, t_l, t_r)) = \exists yz.x \mapsto [n, y, z], \text{tree}(y, t_l), \text{tree}(z, t_r) \\ &\text{tree}(0, \text{leaf}) = \text{emp} \end{aligned} \tag{2.8}$$

There is an extensive literature on such recursive definitions, especially in the context of separation logic [31, 32, 33].

We employ two heuristics. The first is similar to natural proofs [32, 33]. We unfold a recursive definition during symbolic execution when we add conjunctive pattern $\pi \wedge \psi$ to Q if unfolding does not introduce a disjunction (i.e., ψ guarantee that only one of the cases in the definition holds). For example, in C, if ψ implies the head pointer p of a tree is `NULL`, then we conclude the tree is empty. If ψ implies p is not `NULL`, then we conclude p points to an object containing pointers to the left and right subtrees. Successful unfolding occurs at the start of symbolic execution, after a split (e.g. caused by `if`), or after using a specification rule (line 7). Unfolding makes a pattern more concrete, thus enabling operational semantics rules to apply. We similarly unfold recursive definitions

on the right-hand-side of an implication. Unfolding is language-independent, as it is not triggered by memory accesses or other language-specific features.

While the above heuristic works on tree manipulating programs, it fails on list segment manipulating programs, as a list segment can be unfolded at both ends. We solve this by adapting the folding axioms proposed in [34] to work with data, and using them as additional lemmas for list segments on the left-hand-side of an implication, e.g.,

$$\text{lseg}(x, y, \alpha),\ \text{lseg}(y, 0, \beta) = \text{lseg}(x, 0, \alpha \cdot \beta) \tag{2.9}$$

Folding and unfolding are implemented by rewriting using the same infrastructure used for symbolic execution. The recursive definitions and the lemmas are all given as $\mathbb{K}$ rules.

As shown in Section 4.1.1, we use equationally constrained function and predicate symbols (like bst and tree_keys); e.g.,

$$\begin{aligned} &\text{height}(\text{node}(\_, t_l, t_r)) = 1 + \max(\text{height}(t_l), \text{height}(t_r)) \\ &\text{height}(\text{leaf}) = 0 \\ &\text{height}(\_) \geq 0 = \text{true} \end{aligned} \tag{2.10}$$

The first two define the height of a tree, while the third is a lemma. These equations are given as $\mathbb{K}$ rules, and are used in two ways: to simplify a formula by rewriting (oriented from left to right), and to be added in Z3 (see Section 2.3.3).

2.3.3 Integration with Z3

We use Z3 [29] to discharge the formulae that arise during matching logic reasoning (required by Consequence and Step). These formulae involve the following theories: integer, bitvector, set, sequence, and floating-point. We chose Z3 because of its very good performance, and because it offers features that are not part of the SMT-LIB standard, including variables instantiation patterns for universally quantified axioms, and mapping functions over arrays. While some of the formulae are not in decidable theories, in practice Z3 successfully checks them.

As discussed in Section 2.3.2, the formulae contain equationally constrained symbols. We encode these in Z3 as uninterpreted functions combined with assertions of the form "$\forall X.\ t = t'$". Z3 handles such assertions efficiently using E-matching [35]. By default, we specify the left-hand-side of these equations as the variables instantiation pattern, which in effect makes the equations only apply from left to right. This heuristic is effective

in keeping the number of terms small. For a select few equations, like the ones for the sorted predicate for sequences, we wrote the patterns by hand.

Sets are one of the most important theories that we offer in our verifiers. We handle the set theory as proposed in [36]. We encode the sets themselves as arrays from the elements to true or false. Then, we encode the set operations as mapping of boolean functions over the arrays, and set membership as array lookup. The array map feature is only available in Z3, and is not part of the SMT-LIB standard. This results in a decidable theory for sets.

Unfortunately, this set encoding does not work well with the encoding of sequence theory symbols as equationally constrained uninterpreted functions. This case arises during the verification of the sorting examples. For this reason, we developed an encoding of sets using uninterpreted functions and universally quantified assertions. This encoding does not handle the set theory in a decidable way, but in practice it works with the sequence theory.

JavaScript verification generates floating-point constraints. Z3 has basic support for floating-point, but it does not integrate well with other theories. For this reason, we abstracted floating-point values to values in a partial-order relation, when the values only occur in comparisons and equality/inequality checks. This abstraction is used on the keys of the search trees or the values in the sorted lists.

For these reasons, we have different SMT encodings for the different programs we are verifying. We delegate to the user to choose which encodings are best suited for a given program.

Let us discuss how this algorithm derives the reachability logic proof of the correctness of the `SUM` program in Figure 2.5. Note that the algorithm derives the all-path version of the rule instead of the one-path version derived in Figure 2.5. The two rules specify the same property, since `IMP` is deterministic.

In this case, $\mathcal{C}$ is the set $\{\varphi_{pre} \Rightarrow^{\forall} \varphi_{post},\ \varphi_{inv} \Rightarrow^{\forall} \varphi_{post}\}$. First, let us run the algorithm on $\varphi_{pre} \Rightarrow^{\forall} \varphi_{post}$. In line 1, $successors(\varphi_{pre})$ returns φ_1 (corresponding to (4) in Figure 2.5). Since Q is non-empty, we enter the body of the `while` loop, and we set φ_c to be φ_1 (the single element in Q). The check $\models \varphi_1 \rightarrow \varphi_{post}$ in line 5 fails due to the syntactic differences in the `code` cell, without calling Z3. Then we continue to line 6, where the check $\models \varphi_1 \rightarrow \sigma(\varphi_{inv})$ succeeds for $\sigma = \{n' \mapsto n\}$ (proof step (2) in Figure 2.5), and $\varphi_{post} \wedge n \geq_{Int} 1$ is added to Q in line 7. In this step we use Z3 to check that $n' = n \rightarrow \Sigma_{n'}^{n-_{Int}1} = 0$ is valid; specifically, Z3 proves that the negation of the formula is unsatisfiable. We go through the `while` loop again, and this time the check in line 5 succeeds (without use of Z3), and the algorithm terminates successfully (corresponding to (1,3) in Figure 2.5).

Next, we run the algorithm on $\varphi_{inv} \Rightarrow^{\forall} \varphi_{post}$ (corresponding to finding the sub-proof tree rooted at (5) in Figure 2.5). Like before, we compute $\mathit{successors}(\varphi_{inv})$ in line 1, which returns φ_2 (corresponding to (7) in Figure 2.5). We enter the `while` loop, and this time both the checks in lines 5 and 6 fail, so we proceed to line 9. Here, we compute the $\mathit{successors}(\varphi_2)$, which is

$$\begin{array}{l} \langle\langle \texttt{if(}n' -_{Int} 1\texttt{)\{s = s + n; LOOP\} else \{\}} \rangle_{\mathsf{code}} \\ \quad \langle \mathsf{n} \mapsto n',\ \mathsf{s} \mapsto \Sigma_{n'}^{n -_{Int} 1} \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \ \wedge\ n' \geq_{Int} 1 \end{array} \tag{2.11}$$

At the next iteration of the loop, and we reach line 9 again. This time, a proper step of narrowing is performed, and we have the following two successors added to Q (roughly speaking, corresponding to finding the sub-proof trees rooted at (9) and (10), respectively, in Figure 2.5):

$$\begin{array}{l} \langle\langle \texttt{\{s = s + n; LOOP\}} \rangle_{\mathsf{code}} \\ \quad \langle \mathsf{n} \mapsto n',\ \mathsf{s} \mapsto \Sigma_{n'}^{n -_{Int} 1} \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \ \wedge\ n' \geq_{Int} 1 \wedge n' -_{Int} 1 \neq 0 \end{array} \tag{2.12}$$

and

$$\begin{array}{l} \langle\langle \texttt{\{\}} \rangle_{\mathsf{code}} \\ \quad \langle \mathsf{n} \mapsto n',\ \mathsf{s} \mapsto \Sigma_{n'}^{n -_{Int} 1} \rangle_{\mathsf{state}} \rangle_{\mathsf{cfg}} \ \wedge\ n' \geq_{Int} 1 \wedge n' -_{Int} 1 = 0 \end{array} \tag{2.13}$$

We continue iterating through the loop in a similar way going through lines 9-11, where each formula has exactly one successor. Eventually, we reach φ_3 (corresponding to (11) in Figure 2.5), at which point we go through lines 6-7 (corresponding to (12) in Figure 2.5). Finally, we reach twice formulae for which the check in line 5 succeeds, and the algorithm terminates successfully. The successful checks in lines 5 and 6 make calls to Z3 with similar formulae as the one shown above.

2.4 BISIMULATION

Here we provide background on bisimulation and its variants. Most of the contents in this section come from Sangiorgi [19].

There are several notions of behavioral equivalence between transition systems, depending on what kind of behaviors are considered. Notably, we have graph isomorphism, bisimulation, and trace equivalence. Graph isomorphism is the strongest, since it requires two transition systems be isomorphic in their graph structures. This requirement is sometimes too strong, especially in the case that only observable behaviors are considered.

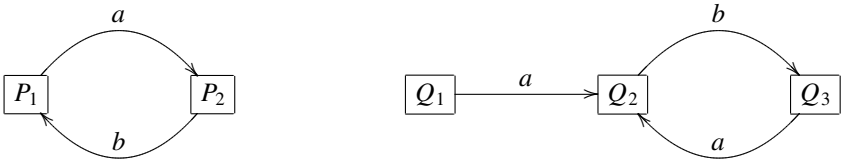


Figure 2.7: Bisimilar, but not graph-isomorphic systems [19]

Figure 2.8: Trace-equivalent, but not bisimilar systems [19]

Figure 2.7 shows as an example two transition systems that are observably equivalent but not graph-isomorphic. On the other hand, trace equivalence is the weakest, since it requires only input-output behaviors be the same. This requirement, however, is sometimes too weak, when intermediate behaviors are considered. Figure 2.8 shows as an example two transition systems that are trace-equivalent but do not have the same intermediate behaviors. Bisimulation is located in the middle. It allows to consider behavioral equivalence rather than structural equivalence, and also takes into account intermediate behaviors as well as input-output behaviors.

Bisimulation also has variants, namely, strong bisimulation and weak bisimulations. Strong bisimulation equally considers every behavior (i.e., transition), while weak bisimulation considers only external (i.e., observable) behaviors, ignoring internal behaviors.

2.4.1 Preliminaries

Definition 2.8 (Labelled transition system)**.** *A* labelled transition system *is a triple* $(\mathcal{S}, \mathcal{L}, \rightarrow)$ *where* $\mathcal{S}$ *is a set of states,* $\mathcal{L}$ *is a set of labels, and* $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ *is a set of labelled transitions. We write* $P \xrightarrow{\mu} Q$ *when* $(P, \mu, Q) \in \rightarrow$*. A labelled transition system is* deterministic *when for all* $P \in \mathcal{S}$ *and* $\mu \in \mathcal{L}$*,* $P \xrightarrow{\mu} P'$ *and* $P \xrightarrow{\mu} P''$ *implies* $P' = P''$*.*

Process Calculus The process calculus CCS [19], inspired by Calculus of Communicating Systems [37], can be used to algebraically represent a labelled transition system.

$$P ::= P_1 | P_2 \mid P_1 + P_2 \mid \mu.P \mid 0$$

$$\textsc{ParL}\ \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2} \qquad \textsc{ParR}\ \frac{P_2 \xrightarrow{\mu} P_2'}{P_1 \mid P_2 \xrightarrow{\mu} P_1 \mid P_2'}$$

$$\textsc{SumL}\ \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'} \qquad \textsc{SumR}\ \frac{P_2 \xrightarrow{\mu} P_2'}{P_1 + P_2 \xrightarrow{\mu} P_2'}$$

$$\textsc{Com}\ \frac{P_1 \xrightarrow{\mu} P_1' \qquad P_2 \xrightarrow{\overline{\mu}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'} \qquad \textsc{Pre}\ \frac{}{\mu.P \xrightarrow{\mu} P}$$

Figure 2.9: CCS syntax and semantics

Syntax and semantics of CCS are given in Figure 2.9. CCS supports parallel compositions, choice (i.e., branching), and prefixing (i.e., sequencing) operators. The restriction operator is omitted for the sake of simplicity. For more details, refer to [19].

2.4.2 Strong Bisimulation

Definition 2.9 (Strong bisimilarity)**.** *A relation $\mathcal{R}$ is a* strong bisimulation *if, whenever $P\ \mathcal{R}\ Q$:*

1. *for all $P \xrightarrow{\mu} P'$, there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\ \mathcal{R}\ Q'$*
2. *for all $Q \xrightarrow{\mu} Q'$, there exists P' such that $P \xrightarrow{\mu} P'$ and $P'\ \mathcal{R}\ Q'$*

Strong bisimilarity*, written $\sim$, is the largest bisimulation (i.e., the union of all bisimulations); thus $P \sim Q$ holds if there is a bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$.*

Strong bisimulation enjoys a nice property; it is preserved by operators such as parallel compositions, choice (i.e., branching), and prefixing (i.e., sequencing).

Theorem 2.2 (Congruence)**.** *$\sim$ is an equivalence relation, i.e.,*

- *$P \sim P$ (reflexivity)*
- *$P \sim Q$ implies $Q \sim P$ (symmetry)*
- *$P \sim Q$ and $Q \sim R$ imply $P \sim R$ (transitivity)*

Furthermore, it is a congruence relation under CCS, i.e., $P \sim Q$ implies $C[P] \sim C[Q]$ for all contexts C, more specifically,

- $P \mid R \sim Q \mid R$
- $P + R \sim Q + R$
- $\mu.P \sim \mu.Q$

Definition 2.10 (Similarity)**.** *A relation $\mathcal{R}$ is a* simulation *if, whenever $P\ \mathcal{R}\ Q$:*

1. *for all $P \xrightarrow{\mu} P'$, there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\ \mathcal{R}\ Q'$;*

Similarity*, written $\leq$, is the largest simulation (i.e., the union of all simulations). We say that Q* simulates *P if $P \leq Q$.*

We give another definition that is useful to prove bisimilarity.

Definition 2.11 (Bisimulation up-to $\sim$)**.** *A relation $\mathcal{R}$ is a* bisimulation up-to $\sim$ *if, whenever $P\ \mathcal{R}\ Q$:*

1. *for all $P \xrightarrow{\mu} P'$, there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim \mathcal{R} \sim Q'$*
2. *for all $Q \xrightarrow{\mu} Q'$, there exists P' such that $P \xrightarrow{\mu} P'$ and $P' \sim \mathcal{R} \sim Q'$*

where $P' \sim \mathcal{R} \sim Q'$ if there exist P'' and Q'' such that $P' \sim P''$, $P''\ \mathcal{R}\ Q''$, and $Q'' \sim Q'$.

Lemma 2.1. *If $\mathcal{R}$ is a bisimulation up-to $\sim$, then $\sim \mathcal{R} \sim$ is a bisimulation and $\mathcal{R} \subseteq \sim$.*

2.4.3 Weak Bisimulation

Weaker notions of behavior equivalence are required when non-observable or internal transitions in the systems should not be counted in deciding similarity. Suppose the label τ represents such internal transitions. We first define new relations to specify transitions involving τ-moves.

Definition 2.12 (Weak transitions)**.** *Several weak transitions are defined as follows.*

- *Relation $\Longrightarrow$ is the reflexive and transitive closure of internal transitions, $\xrightarrow{\tau}$. That is, $P \Longrightarrow P'$ holds if there exist $n \geq 0$ and $P_1, \cdots, P_n$ such that $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n = P'$.*
- *Relation $\stackrel{\mu}{\Longrightarrow}$ is the composition of $\Longrightarrow$, $\xrightarrow{\mu}$, and $\Longrightarrow$. That is, $P \stackrel{\mu}{\Longrightarrow} P'$ holds if there exist P_1 and P_2 such that $P \Longrightarrow P_1 \xrightarrow{\mu} P_2 \Longrightarrow P'$.*

- *Relation* $\stackrel{\widehat{\mu}}{\Longrightarrow}$ *is defined as* $\begin{cases} \stackrel{\mu}{\Longrightarrow} & \text{if } \mu \neq \tau \\ \Longrightarrow & \text{otherwise} \end{cases}$

A weaker notion of bisimilarity can be defined by using the weak transition $\stackrel{\widehat{\mu}}{\Longrightarrow}$ instead of $\stackrel{\mu}{\longrightarrow}$ in Definition 2.9.

Definition 2.13 (Weak bisimilarity)**.** *A relation $\mathcal{R}$ is a* weak bisimulation *if, whenever $P\ \mathcal{R}\ Q$:*

1. *for all $P \stackrel{\mu}{\longrightarrow} P'$, there exists Q' such that $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P'\ \mathcal{R}\ Q'$*
2. *for all $Q \stackrel{\mu}{\longrightarrow} Q'$, there exists P' such that $P \stackrel{\widehat{\mu}}{\Longrightarrow} P'$ and $P'\ \mathcal{R}\ Q'$*

Weak bisimilarity*, written $\approx$, is the largest weak bisimulation (i.e., the union of all weak bisimulations); thus $P \approx Q$ holds if there is a weak bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$.*

Unlike strong bisimilarity, weak bisimilarity is not a congruence relation; it is not preserved by choice operators. For example, $\tau.a \approx a$ but $\tau.a + b \not\approx a + b$. However, it is preserved by guarded forms of choice, i.e., $\mu_1.P_1 + \cdots + \mu_n.P_n$, which is most of the use case in practice.

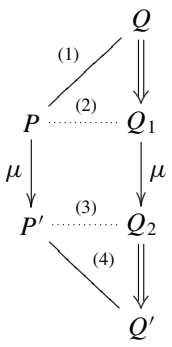
Lemma 2.2. *If $P \approx Q$, then for all R and μ, we have:*

- $P \mid R \approx Q \mid R$
- $\mu.P \approx \mu.Q$
- $\mu.P + R \approx \mu.Q + R$

but $P + R \not\approx Q + R$ in general; thus $\approx$ is not *a congruence under CCS.*

There is a variant of weak bisimilarity, called *rooted weak bisimilarity*, written $\approx^c$, that has additional condition, so called 'being rooted', so as to be a congruence under CCS. Essentially, $\approx^c$ is the same with $\approx$ except what are not preserved by CCS operators. That is, $P \approx^c Q$ iff $C[P] \approx C[Q]$.

There is another variant of weak bisimilarity, called *dynamic bisimilarity*, written $\approx_{\text{dyn}}$, which is a bit stronger than weak bisimilarity, but a congruence under CCS without any 'rooted' conditions. Section 2.4.5 highlights differences between these variants. For more details, refer to [19].



		(1)	(2)	(3)	(4)
Weak bisimulation	$\approx$	○	×	×	○
Delay bisimulation	$\approx_{\mathrm{d}}$	○	×	○	○
η-bisimulation	$\approx_{\eta}$	○	○	×	○
Branching bisimulation	$\approx_{\mathrm{br}}$	○	○	○	○

Figure 2.10: Bisimulation game in $\approx$, $\approx_{\mathrm{d}}$, $\approx_{\eta}$, and $\approx_{\mathrm{br}}$ [19]

2.4.4 Branching Bisimulation

Ignoring τ-transitions may lead to failure in distinguishing branch structures. For example, two transition systems shown in Figure 2.11 have different branch structures but they are still weakly bisimilar. In order to distinguish branch structures in the presence of τ-transitions, we need to relate nodes over the path through $\Longrightarrow$. Figure 2.10 shows a local condition of weak bisimulation from P. Weak bisimulation considers the relations (1) and (4), but not (2) and (3). That means that Q_1 and Q_2 may be not related to others, which may lead to branch structures to be indistinguishable. By additionally considering (2) and/or (3), we have variants of weak bisimulation as follows.

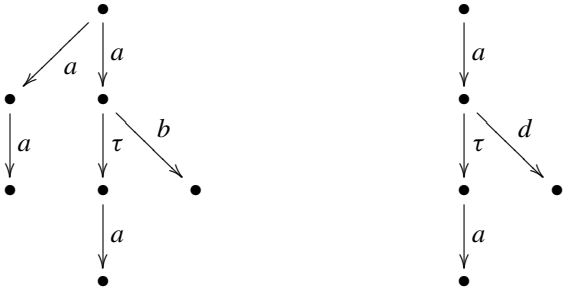
Definition 2.14 (Branching, η-, and delay bisimilarities)**.** *A relation $\mathcal{R}$ is a* branching bisimulation *if, whenever $P\ \mathcal{R}\ Q$:*

1. *for all $P \xrightarrow{\mu} P'$,*
   - *(a) $\mu = \tau$ and $P'\ \mathcal{R}\ Q$*
   - *(b) or, there exists Q_1, Q_2, Q' such that $Q \Longrightarrow Q_1 \xrightarrow{\mu} Q_2 \Longrightarrow Q'$ and*
     - *i. $P\ \mathcal{R}\ Q_1$*
     - *ii. $P'\ \mathcal{R}\ Q_2$*
     - *iii. $P'\ \mathcal{R}\ Q'$*
2. *converse of (1) from Q*

*Furthermore, η-*bisimulation *(resp.* delay bisimulation*) is defined in the same way above, except the requirement 1.b.ii (resp. 1.b.i).*

Branching *(resp. η- and* delay*)* bisimilarity*, written $\approx_{br}$ (resp. $\approx_{\eta}$ and $\approx_{d}$), is the largest branching (resp. η- and delay) bisimulation.*

(i) Systems in $\approx$ and $\approx_{\mathrm{d}}$, but not in $\approx_{\eta}$ and $\approx_{\mathrm{br}}$



(ii) Systems in $\approx$ and $\approx_{\eta}$, but not in $\approx_{\mathrm{d}}$ and $\approx_{\mathrm{br}}$

Figure 2.11: Systems in $\approx$, but not in $\approx_{\mathrm{br}}$ [19]

Branching, η-, and delay bisimilarities are not preserved by choice operators, as weak bisimilarity. They are needed to be 'rooted', in the similar way with weak bisimilarity, so as to be congruence relations under CCS.

Delay bisimilarity is natural in the sense that it can be defined in the same way with weak bisimilarity, except that τ-moves are not allowed after μ-transition, as shown in Section 2.4.5.

2.4.5 Comparison of Bisimulation Variants

Strong, weak, dynamic, and delay bisimulations can be defined in the same structure. They can be presented by instantiating the place-holder $\star$ in the following definition, as shown in Table 2.1.

Definition 2.15 ($\star$-bisimilarity)**.** *A relation $\mathcal{R}$ is a* $\star$-bisimulation *if, whenever $P\ \mathcal{R}\ Q$:*

1. *for all $P \xrightarrow{\mu} P'$, there exists Q' such that $Q \xrightarrow{\star} Q'$ and $P'\ \mathcal{R}\ Q'$*
2. *for all $Q \xrightarrow{\mu} Q'$, there exists P' such that $P \xrightarrow{\star} P'$ and $P'\ \mathcal{R}\ Q'$*

$\star$-bisimilarity, *written $\overset{\star}{\sim}$, is the largest $\star$-bisimulation (i.e., the union of all $\star$-bisimulations); thus $P \overset{\star}{\sim} Q$ holds if there is a $\star$-bisimulation $\mathcal{R}$ such that $P\ \mathcal{R}\ Q$.*

$\star$-bisimulation	$\xrightarrow{\star}$	$\overset{\star}{\sim}$
Strong bisimulation	$\xrightarrow{\mu}$	$\sim$
Dynamic bisimulation	$\Longrightarrow\xrightarrow{\mu}\Longrightarrow$	$\approx_{\mathrm{dyn}}$
Weak bisimulation	$\overset{\widehat{\mu}}{\Longrightarrow} \overset{\mathtt{def}}{=} \begin{cases} \Longrightarrow\xrightarrow{\mu}\Longrightarrow & \text{if } \mu \neq \tau \\ \Longrightarrow & \text{otherwise} \end{cases}$	$\approx$
Delay bisimulation	$\begin{cases} \Longrightarrow\xrightarrow{\mu} & \text{if } \mu \neq \tau \\ \Longrightarrow & \text{otherwise} \end{cases}$	$\approx_{\mathrm{d}}$

Table 2.1: Instantiations of Definition 2.15

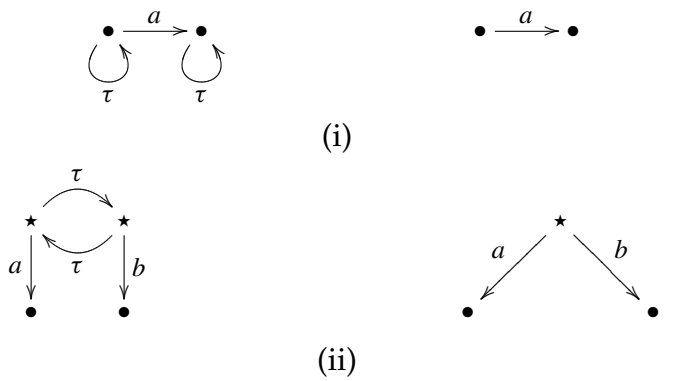


Figure 2.12: Two pairs of systems that are weakly bisimilar in the presence of τ-cycles [19]

Divergence Weak bisimilarity is insensitive to τ-cycles, that is, a system with τ-cycles can be weakly bisimilar one without τ-cycles. For example, suppose Ω with a transition $\Omega \xrightarrow{\tau} \Omega$, then we have $P \mid \Omega \approx P$, as shown in Figure 2.12 (i). For another example, suppose K with transitions, $K \xrightarrow{\tau} K$ and $K \xrightarrow{\tau} P$, then we have $K \approx \tau.P$, as similarly shown in Figure 2.12 (ii). This property is useful when one wants to abstract out internal cycles so that the abstracted system be internal-cycle-free, allowing to focus on only important transitions.

There is a variant of weak bisimilarity, called *prebisimilarity with divergence*, written $\leq_{\Uparrow}$, to distinguish two systems where one has τ-cycles and the other does not. That is, $P \leq_{\Uparrow} Q$ means that P may diverge while Q is not. In this case, such bisimilarity becomes a preorder (i.e., not necessarily symmetric, but only reflexive and transitive), rather than an equivalence relation. For more details, refer to [19].

2.5 THE K FRAMEWORK

$\mathbb{K}$ [38] (`http://kframework.org`) is a framework for defining language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, as well as formal analysis tools such as model checkers and deductive program verifiers, at no additional cost. Using the interpreter, one can test their semantics immediately, which significantly increases the efficiency of semantics developments. Furthermore, the formal analysis tools facilitate formal reasoning about the given language semantics. This helps both in terms of applicability of the semantics and in terms of engineering the semantics itself; for example, the state-space exploration capability helps the language designer cover all the non-deterministic behaviors of certain constructs or combinations of them in the language definition.

We briefly describe $\mathbb{K}$ here and refer the reader to [38, 27] for more details. In $\mathbb{K}$, a language syntax is given using conventional Backus-Naur Form (BNF). A language semantics is given as a transition system, specifically a set of reduction rules over configurations. A configuration is an algebraic representation of the program code and state. Intuitively, it is a tuple whose elements (called cells) are labeled and possibly nested. Each cell represents a semantic component such as stores, environments, and threads that are used in defining semantics. A special cell, named k, contains a list of computations to be executed. A computation is essentially a program fragment, while the original program is flattened into a sequence of computations. A rule describes a one-step transition relation between configurations, thus giving semantics to language constructs. Rules are modular; they mention only relevant cells that are needed in each rule. For example, an object field lookup semantics can be defined as the following $\mathbb{K}$ rule:

$$\langle \frac{O[X]}{V} \ \cdots \rangle_{\mathsf{k}} \ \langle \langle O \rangle_{\mathsf{oid}} \ \langle \cdots X \mapsto V \cdots \rangle_{\mathsf{fields}} \cdots \rangle_{\mathsf{obj}} \tag{2.14}$$

The cells are represented with angle brackets notation. The horizontal line represents a reduction (i.e., a transition relation). A cell with no horizontal line means that it is read but not changed by the rule. The rule above mentions two cells: k, and obj. The k cell contains a list of computations to be executed, and the obj cell represents an object. The obj cell contains several sub-cells: e.g., the oid cell contains the object identifier and the fields cell stores a map from field names to values. This rule is applied when the current computation (top of the k cell) is a field lookup and there exists an obj cell whose oid is matched with O and fields contains the field name X. This rule resolves the object field

lookup $O[X]$ to the value V. The "..." is a structural frame, that is, it matches the portions of a cell that are neither read nor written by the rule.

One of the most appealing aspects of $\mathbb{K}$ is its modularity. It is very rarely the case that one needs to touch existing rules in order to add a new feature to the language. This is achieved by structuring the configuration as nested cells and by requiring the language designer to mention only the cells that are needed in each rule, and only the needed portions of those cells. For example, the above rule only refers to the k and obj cells, while the entire configuration may contain many more cells. This modularity makes for compact and human readable semantics, and also helps with the overall effectiveness of the semantics development. For example, even if new cells are later added to configuration, to support new features, the above rule does not change.

Another appealing aspect of $\mathbb{K}$ is its inherent support for non-determinism. As $\mathbb{K}$ is based on rewriting logic [39], one can easily define, execute, and reason about non-deterministic specifications in $\mathbb{K}$. For example, a simplified for-in loop semantics can be defined as the following $\mathbb{K}$ rules:

$$\frac{\texttt{for}\ I\ \texttt{in}\ E\ Es\ \{\ S\ \}}{I = E\ ;\ S\ ;\ \ \texttt{for}\ I\ \texttt{in}\ Es\ \{\ S\ \}} \qquad \frac{\texttt{for}\ I\ \texttt{in}\ \cdot_{Set}\ \{\ S\ \}}{\cdot} \tag{2.15}$$

Suppose that for-in loop non-deterministically iterates through the given elements. In $\mathbb{K}$, such non-determinism can be easily described by representing the elements as a set and using set matching, which gives us the desired set-theoretical 'choice' operation. In the above semantics, '$E\ Es$' represents the set of elements to be iterated through, where E refers to an arbitrary element of the set, and Es the remaining elements. The rule in the left-hand side says that it chooses an arbitrary element E, runs the loop body S with the element, and proceeds to the next iteration with the remaining elements Es. The rule in the right-hand side specifies the termination condition of the loop. This way, one can easily describe and execute non-deterministic semantics. Furthermore, using $\mathbb{K}$'s 'search'-mode execution, one can explore all possible execution traces, in this case all possible iteration orders.

CHAPTER 3: CROSS-LANGUAGE PROGRAM EQUIVALENCE

Intuitively, two (possibly non-terminating) programs are equivalent when given the same input they reach the same relevant states in the same order.

In the formal methods literature, the notion of program (or semantic) equivalence is usually formalized as a bisimulation relation between pairs of states of the two programs that are subject to prove equivalence (which in turn are represented as state transition systems). Reducing program equivalence to proving a relation to be a bisimulation, allows for a coinductive proof that deals with recursion, loops, and non-termination in a uniform and elegant way.

Classic notions of bisimulation, however, are too strong for the purposes of program equivalence. Consider the simple program transformation example shown in Figure 3.1(a). This transformation is common in compiler optimization (e.g., partial redundancy elimination). The seemingly equivalent two programs, however, are not strongly bisimilar, mainly because the intermediate states are not "similar". Weaker variants, such as stuttering or branching bisimulation, can be used to prove their equivalence, since they are flexible to admit the irrelevant intermediate states.[1] Figure 3.1(b) shows two possible stuttering bisimulation relations. These relations, however, do not seem to be intuitive, especially the pairs involving the intermediate states (marked in red). The problem with these counter-intuitive relations becomes apparent when we consider the witness-based translation validation approach [40]. In that approach, the relation is generated by the compiler as a "witness" for the correctness of the transformation, and proving the equivalence is reduced to checking that the generated relation is a (bi)simulation. However, the counter-intuitive relations (especially the red pairs) are not easy to be generated by the compiler, as they are not directly related to the internal information that the compiler uses for the transformation. Thus, the non-intuitive (red) pairs should be inferred separately, which incurs additional overhead in proving equivalence.[2]

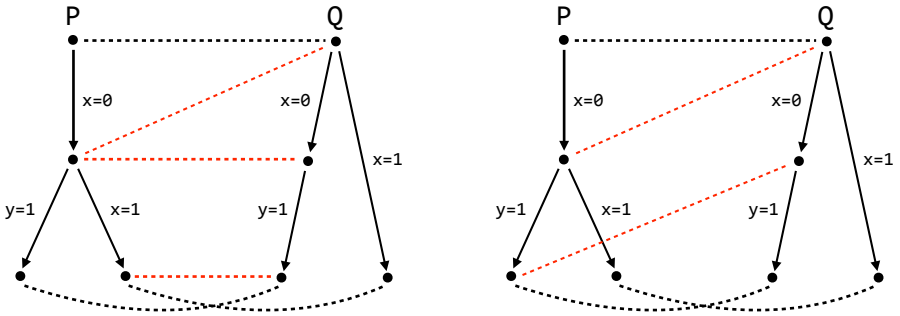
The counter-intuitive relation issue occurs because the classic variants of bisimulation necessitate every state being related to another, even for intermediate states that are not relevant for equivalence. Ideally, we want a bisimulation variant that allows us to simply relate the states where the two programs actually synchronize, i.e., at the start

[1] While stuttering bisimulation (and branching bisimulation) is originally defined over Kripke structures (and labeled transition systems, respectively), we can use them over transition systems by assuming a single atomic proposition and a single label.

[2] The time complexity of the best known algorithm for computing stuttering bisimulation is $O(m \log n)$ where m is the number of transitions and n is the number of states [41].

```
P:  x = 0; if (*) { y = 1; } else { x = 1; }
Q:  if (*) { x = 0; y = 1; } else { x = 1; }
```

(a)



(b)

Figure 3.1: Program transformation example (as part of partial redundancy elimination), and two stuttering bisimulation relations (dotted lines). The `if(*)` statement denotes the non-deterministic branching operation.

of corresponding functions or basic blocks, at the loop headers, etc. We also want to be able to control the granularity of these synchronization points, so that we can adapt how closely we follow the behavior of the programs depending on the knowledge of the transformations that can be easily extracted from the compiler.

For these reasons, we introduce a new notion of bisimulation, which we call *cut-bisimulation*, between relevant program points in the input and output programs. We call the relevant program states *cut points*, and their set simply a *cut*. The intuition for the cut of a transition system corresponding to a program is that the states in the cut suffice as observation points of the program behavior, that is, nothing relevant can happen which is not witnessed by a cut state. Then we can define bisimulations only between cut states; hence the name *cut bisimulations*. For the example in Figure 3.1, simply two synchronization points (at the beginning and the end of the shown code sequences) are enough to define a cut-bisimulation relation, allowing us to prove equivalence.

In order for cut bisimulations to correctly capture program equivalence, two conditions must be satisfied. First, there must be enough cut states in the two transition systems so that no relevant behavior of one program can pass unsynchronized with a behavior of the other program. This implies, in particular, that each final state must be in the cut. It also implies that each infinite execution must contain infinitely many cut states, because otherwise one of the programs may not terminate while the other terminates.

Second, any two states related by a cut bisimulation must be compatible. Otherwise, one can establish a cut bisimulation even for non-equivalent programs.[3] One of straightforward compatibility relations relates two states when their corresponding variables have the "same" value. However, what it precisely means for two values in different languages to be the same is not trivial, due to different representations (e.g., big-endian vs little-endian, or 32-bits vs 64-bits), different memory layouts (physically same location may point to different values, or contain garbage that has not been collected yet), etc. Also, state compatibility may require to check if specific memory locations (in the context of embedded systems), environment variables, input/output buffers, files, etc., are also "the same". Moreover, states corresponding to undefined behaviors (e.g., division by zero) may or may not be desired to be compatible, depending upon what kind of equivalence is desired. Thus, we design the notion of cut-bisimulation to be parameterized by a binary relation on states $\mathcal{A}$, which we call an *acceptability* (or *compatibility* or *indistinguishability*) relation.

3.1 CUT-BISIMULATION

We present our novel notion of cut bisimulation, which makes it easier to deal with intermediate states that are not relevant in identifying equivalence of programs written in potentially different languages.

Given a binary relation $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$, we write $a\ R\ b$ to denote $(a, b) \in R$; and $R_1 = \{a \mid \exists b\,.\, a\ R\ b\}$ and $R_2 = \{b \mid \exists a\,.\, a\ R\ b\}$ to denote the projections $\Pi_i(R)$ for $i \in \{1, 2\}$.

Let $\mathcal{S}$ be a set of states (thought of as all possible configurations/states of a language, over all programs in the language). Let $T = (S, \xi, \rightarrow)$ be an *$\mathcal{S}$-transition system*, or just a *transition system* when $\mathcal{S}$ is understood, that is a triple consisting of: a set of *states* $S \subseteq \mathcal{S}$, an *initial state* $\xi \in S$, and a (possibly nondeterministic) *transition relation* $\rightarrow\ \subseteq S \times S$. Let $next(s)$ denote the set $\{s' \mid s \rightarrow s'\}$. T is *finitely branching* iff $next(s)$ is finite for each $s \in S$. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^+$ be the transitive closure of $\rightarrow$.

A (possibly infinite) *trace* $\tau = s_0 s_1 \cdots s_n \cdots$ is a sequence of states with $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. Let $\tau[n]$ be the n^{th} state of τ where the index starts from 0, and let $\mathsf{size}(\tau)$ be the length of τ (∞ when τ is infinite). Let $\mathsf{first}(\tau) = \tau[0]$ be the first state of τ, and let

[3] For any two terminating programs, for example, there always exists a trivial cut bisimulation where all initial (and final) states are related to each other, respectively, if the compatibility of the states is not considered.

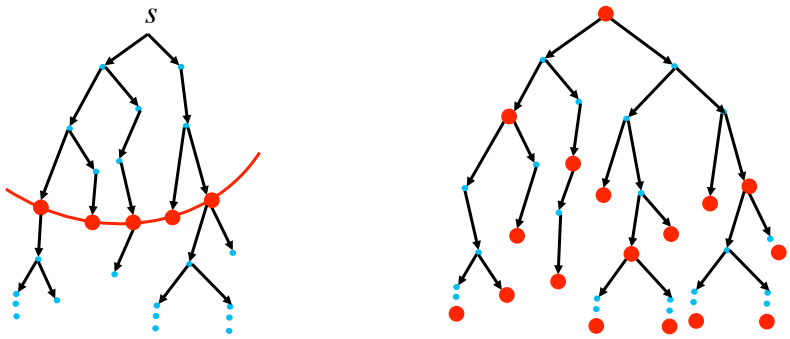


Figure 3.2: Left: a cut C for state s (each complete s-trace intersects C). Right: a cut C for a transition system (C contains the initial state and is a cut for itself, i.e., for each state in C)

$\mathsf{final}(\tau)$ be the final state of τ when τ is finite. Let $\mathsf{traces}(s)$ be the set of all traces starting with s, also called *s-traces*, and let $\mathsf{traces}(S)$ be $\bigcup_{s \in S} \mathsf{traces}(s)$. A *complete trace* is either an infinite trace, or a finite trace τ where $\mathit{next}(\mathsf{final}(\tau)) = \emptyset$.

Definition 3.1 (Cut and Cut Transition System)**.** *Let $T = (S, \xi, \rightarrow)$ be a transition system. A set $C \subseteq S$ is a* cut for $s \in S$*, iff for any complete trace $\tau \in \mathsf{traces}(s)$, there exists some strictly positive $k > 0$ such that $\tau[k] \in C$. The set $C \subseteq S$ is a* cut for T *iff $\xi \in C$ and C is a cut for each $s \in C$, in that case T is called a* cut transition system *and is written as a quadruple $(S, \xi, \rightarrow, C)$. See Figure 3.2.*

In a cut transition system, any finite complete trace starting with the initial state terminates in a cut state, and any infinite trace starting with the initial state goes through cut states infinitely often:

Lemma 3.1. *Let $T = (S, \xi, \rightarrow, C)$ be a cut transition system. Then for each complete trace $\tau \in \mathsf{traces}(\xi)$ and each $0 < i < \mathsf{size}(\tau)$, there is some $j \geq i$ such that $\tau[j] \in C$.*

Proof. Let $\tau \in \mathsf{traces}(\xi)$ be a complete trace. Assume to the contrary that there exists i such that $\forall j \geq i.\ \tau[j] \notin C$. Pick such an i. Then we have two cases. When $\forall k < i.\ \tau[k] \notin C$, we have $\forall k > 0.\ \tau[k] \notin C$, which is a contradiction since C is a cut for $\xi = \tau[0]$. Otherwise, $\exists k < i.\ \tau[k] \in C$, and let k be the largest such number. Then, we have $\forall l > k.\ \tau[l] \notin C$, which is a contradiction since C is a cut for each $s \in C$, thus a cut for $\tau[k] \in C$.

This result is reminiscent of the notion of Büchi acceptance [42]; specifically, if S is finite and $\mathit{next}(s) \neq \emptyset$ for all $s \in S$, then it says that the transition system T regarded as a Büchi automaton with C as final states, accepts all the infinite traces. This analogy was not intended and so far played no role in our technical developments.

Cuts do not need to be minimal in practice, and are not difficult to produce. For example, a typical cut includes all the final states (normally terminating states, error/exception states, etc.) and all the states corresponding to entry points of cyclic constructs in the language (loops, recursive functions, etc.). Such cut states can be easily identified statically using control-flow analysis, or dynamically using a language operational semantics.

Definition 3.2 (Cut-Successor)**.** *Let $T = (S, \xi, \rightarrow, C)$ be a cut transition system. A state s' is an (immediate)* cut-successor *of s, written $s \rightsquigarrow s'$, iff there exists a finite trace $ss_1 \cdots s_n s'$ where $s' \in C$ and $n \geq 0$ and $s_i \notin C$ for all $1 \leq i \leq n$.*

Definition 3.3 (Cut-Bisimilarity)**.** *Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). Relation $R \subseteq C_1 \times C_2$ is a* cut-simulation *iff whenever $(s_1, s_2) \in R$, for all s_1' with $s_1 \rightsquigarrow_1 s_1'$ there is some s_2' such that $s_2 \rightsquigarrow_2 s_2'$ and $(s_1', s_2') \in R$. Let $\leq$ be the union of all cut-simulations (also a cut-simulation). Relation R is a* cut-bisimulation *iff both R and R^{-1} are cut-simulations. Let $\sim$ be the union of all cut-bisimulations (also a cut-bisimulation).*

Cut-bisimulation generalizes standard (strong) bisimulation [19]. A cut-bisimulation on $(S_i, \xi_i, \rightarrow_i, C_i)$ is a bisimulation on $(S_i, \xi_i, \rightarrow_i)$, when $C_i = S_i$. The cuts, however, allow us to consider only the relevant states when comparing two program executions, and *completely hide* the irrelevant intermediate states in each of the two transition systems. As discussed earlier in this section, in our application domain of cross-language translation-validation, this hiding of irrelevant states is critical. It is not sufficient to consider them internal states connected via ϵ-transitions in the sense of the various weaker notions of bisimulation [19], simply because the execution of one of the programs may step through intermediate states which are not observable or related to intermediate states in the execution of the other program. Nevertheless, cut-bisimulation becomes bisimulation if we cut-abstract the transition systems:

Definition 3.4 (Cut-Abstract Transition System)**.** *Let T be a cut transition system $(S, \xi, \rightarrow, C)$. The* cut-abstract transition system of T*, written $\overline{T}$, is the (standard) transition system $(C, \xi, \rightsquigarrow)$.*

Proposition 3.1. *Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). A relation $R \subseteq C_1 \times C_2$ is a cut-bisimulation on T_1 and T_2, iff R is a (standard) bisimulation on $\overline{T_1}$ and $\overline{T_2}$.*

Corollary 3.1. *Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). Let R be a cut-bisimulation, and $(s_1, s_2) \in R$. For any state $s_1' \in C_1$ with $s_1 \rightarrow_1^+ s_1'$, there exists some $s_2' \in C_2$ with $s_2 \rightarrow_2^+ s_2'$ such that $(s_1', s_2') \in R$. The converse also holds.*

Now we formalize the equivalence of cut transition systems in the presence of a given acceptability (or compatibility, or indistinguishability) relation $\mathcal{A}$ on states.

Definition 3.5. *Let $\mathcal{A} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$, which we call an* acceptability relation*. Let $T_i = (S_i, \xi_i, \rightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). T_2* cut-simulates *T_1* (i.e., *T_1* cut-refines *T_2*) w.r.t. $\mathcal{A}$, written $T_1 \leq_{\mathcal{A}} T_2$, iff there exists a cut-simulation $\mathsf{P} \subseteq \mathcal{A}$ such that $\xi_1 \; \mathsf{P} \; \xi_2$. Furthermore, T_1* and *T_2* are cut-bisimilar w.r.t. $\mathcal{A}$, written $T_1 \sim_{\mathcal{A}} T_2$, iff there exists a cut-bisimulation $\mathsf{P} \subseteq \mathcal{A}$ such that $\xi_1 \; \mathsf{P} \; \xi_2$.*

Note that if a cut bisimulation P like above exists, then there also exists a largest one; that's because the union of cut bisimulations included in $\mathcal{A}$ is also a cut bisimulation included in $\mathcal{A}$. We let the relation $\sim_{\mathcal{A}}$ denote that largest cut bisimulation, assuming that it exists whenever we use the notation (and similarly for $\leq_{\mathcal{A}}$).

Our thesis is that $\sim_{\mathcal{A}}$ yields the right notion of program equivalence. That is, that two programs are equivalent according to a given state acceptability (or compatibility or indistinguishability) relation $\mathcal{A}$ between the states of the respective programming languages, iff for any input, the cut transition systems T_1 and T_2 corresponding to the two program executions satisfy $T_1 \sim_{\mathcal{A}} T_2$. The following result strengthens our thesis, stating that cut-bisimilar transitions systems reach compatible states at cut points, and, furthermore, that they cannot indefinitely avoid the cut points:

Theorem 3.1. *If $T_1 \sim_{\mathcal{A}} T_2$ then for each s_1 with $\xi_1 \rightarrow_1^+ s_1$ there exists some s_2 with $\xi_2 \rightarrow_2^+ s_2$, such that: (1) if $s_1 \in C_1$ then $s_1 \sim_{\mathcal{A}} s_2$; and (2) if $s_1 \notin C_1$ then there exists some $s_1' \in C_1$ such that $s_1 \rightarrow^+ s_1'$ and $s_1' \sim_{\mathcal{A}} s_2$. The converse also holds.*

Proof. We only need to show the forward direction, since the backward is dual. First we have $\xi_1 \sim \xi_2$ by Definition 3.5 and the fact that $\sim$ is the union of all cut-bisimulations. Let s_1 be a state with $\xi_1 \rightarrow_1^+ s_1$. Then we have two cases:

- When $s_1 \in C_1$. There exists s_2 such that $\xi_2 \rightarrow_2^+ s_2$ and $s_1 \sim s_2$ by Corollary 3.1.
- When $s_1 \notin C_1$. There exists s_1' such that $s_1 \rightarrow_1^+ s_1'$ and $s_1' \in C_1$ by Lemma 3.1 and the fact that C_1 is a cut for $\xi_1 \in C_1$. Then, there exists s_2 such that $\xi_2 \rightarrow_2^+ s_2$ and $s_1' \sim s_2$ by Corollary 3.1.

3.1.1 Property Preservation

Consider two cut transition systems where one cut-simulates another, but not the other way around. For example, an abstract model cut-simulates its concrete implementation, if implemented correctly, but the inverse may not hold since the model may omit to specify some details, leaving as implementation-dependent, for which the implementation can

freely choose any behavior. In this case, it is not trivial to see whether a property of the model is also held in the implementation. Intuitively, the set of all reachable cut-states of the model is a super set of that of the implementation. Thus, if a cut-state is not reachable in the model, then it is also not reachable in the implementation. This implies that safety properties of the model are preserved in the implementation, since a safety property can be represented as "nothing bad happens", i.e., in other word, "a bad state is not reachable". Liveness properties are also preserved, since a liveness property can be represented as "something good eventually happens", i.e., "a good state is reachable in any path", while there exists a path of the model that appears in the implementation.[4] In general, inductive invariants are preserved in the refined system.

Now we formulate the property preservation of cut-simulation. Let $T = (S, \xi, \to, C)$ be a cut transition system. Let P be a predicate over a domain D, and $f : S \to D$ be a state normalization function. Let P_f be a predicate over S, defined by $P_f(s) \stackrel{\text{def}}{=} P(f(s))$ for some $s \in S$. The predicate P_f is a cut-inductive invariant of T, if $P_f(\xi)$, and $P_f(s) \wedge s \rightsquigarrow s' \implies P_f(s')$ for any states $s, s' \in S$. A cut-inductive invariant, thus, holds for all reachable cut-states. Also, let $T_i = (S_i, \xi_i, \to_i, C_i)$ be two cut transition systems (for $i \in \{1, 2\}$). Suppose $T_1 \geq T_2$, that is, T_1 cut-simulates T_2, (in other word, T_2 cut-refines T_1). We say $\geq$ is right-total if for all $s_2 \rightsquigarrow_2 s_2'$, there exists $s_1 \rightsquigarrow_1 s_1'$ such that $s_1 \geq s_2$ and $s_1' \geq s_2'$.

Theorem 3.2. *Suppose $T_1 \geq T_2$. Suppose $\geq$ is right-total, and $\xi_1 \geq \xi_2$. Suppose P_{f_1} is a cut-inductive invariant of T_1, and $f_1(s_1) = f_2(s_2)$ if $s_1 \geq s_2$. Then, P_{f_2} is a cut-inductive invariant of T_2.*

Proof. $P_{f_2}(\xi_2)$ since $\xi_1 \geq \xi_2$ and $P_{f_1}(\xi_1)$. Suppose $P_{f_2}(s_2)$ and $s_2 \rightsquigarrow_2 s_2'$. Since $T_1 \geq T_2$ and $\geq$ is right-total, there exists $s_1 \rightsquigarrow_1 s_1'$ such that $s_1 \geq s_2$ and $s_1' \geq s_2'$. Then, $P_{f_1}(s_1)$ since $f_1(s_1) = f_2(s_2)$. Since P_{f_1} is inductive, $P_{f_1}(s_1')$. Thus, $P_{f_2}(s_2')$ since $f_1(s_1') = f_2(s_2')$.

3.2 LANGUAGE-PARAMETRIC PROGRAM EQUIVALENCE CHECKER IN K

We implemented a language-independent equivalence checking tool on top of the $\mathbb{K}$ framework [27] (`http://kframework.org`). $\mathbb{K}$ provides a language for defining operational semantics of programming languages, and a series of generic tools that take a language

[4] Indeed, this is true only when two systems are non-terminating. For terminating systems, we need to add an extra condition to the cut-simulation that prevents the implementation admits only sub-path(s) of the model. That is, we need a condition that restricts the cut-simulation relation relate an initial state (and final state) to only another initial state (and final state, respectively) or itself.

semantics as input and specialize themselves for that language: concrete execution engine (interpreter), symbolic execution engine, (bounded) model checker, and a deductive program verifier. The main idea underlying $\mathbb{K}$ is that a given language operational semantics is turned into a transition system generator, one for each program, and a suite of existing components provide the capability to work with such transition systems generically, in a language-independent manner. We developed a new such tool, KEQ, which takes *two* language semantics as input and yields a checker that takes two programs as input, one in each language, and a (symbolic) synchronization relation, and checks whether the two programs are indeed equivalent with the synchronization relation as witness.

Note that checking program equivalence in Turing complete languages is equivalent to checking the totality of a Turing machine (whether it terminates on all inputs), which is a known Π_2^0-complete problem [43], so strictly harder than recursively or co-recursively enumerable. It is therefore impossible to find any algorithm that can say whether any two given programs are equivalent or not. The best we can do is to find techniques and algorithms that work well enough in practice. Definition 3.5 suggests such a technique: find a (witness) relation P and show that it is a cut-bisimulation. While finding such a relation is hard in general, it is relatively easy to check if a given relation, for example one produced by an instrumented compiler, is a cut-bisimulation.

Our KEQ implementation follows the model of the theoretical Algorithm 3.1. Function main essentially checks whether P is a cut-bisimulation: for each pair $(p_1, p_2) \in \mathsf{P}$, for each p_1' with $p_1 \rightsquigarrow_1 p_1'$, there exists p_2' with $p_2 \rightsquigarrow_2 p_2'$ such that $p_1' \ \mathsf{P} \ p_2'$; and the converse. It first gets the cut-successors of p_i (at line 7), and checks whether each pair of the successors is related in P (line 9). The pairs found to be related in P are marked in black (line 10), while the others remain in red. If all of the successors are in black, it returns true (line 12). Note that the algorithm can also be used for checking whether P is a cut-simulation, by simply considering only N_1 in the line 12, i.e., replacing the if-condition with $\forall n \in N_1.\ n.\mathsf{color} = \mathsf{black}$.

Due to its concrete (as opposed to symbolic) nature, Algorithm 3.1 may not terminate in practice, since P could be infinite. Line 2 assumes that P is recursively enumerable, so iterable. Furthermore, lines 19 and 8 terminate only if T_i is finitely branching. We will explain how to refine Algorithm 3.1 to be practical shortly; for now we can show that it is refutation-complete, in the sense that if it does not terminate then the two programs are equivalent.

Data: $T_i = (S_i, \xi_i, \to_i, C_i)$; $\mathsf{P} \subseteq C_1 \times C_2$; `// P is r.e.`

1 **Function** `main()`:
2 **foreach** $\boxed{(p_1, p_2) \in \mathsf{P}}$ **do** `//` $\boxed{\overline{\mathsf{P}}}$
3 **if** $\mathsf{check}(p_1, p_2) = \mathit{false}$ **then**
4 **return** *false*;
5 **return** *true*;
6 **Function** $\mathsf{check}(p_1, p_2)$:
7 $N_1 \leftarrow \mathsf{next}_1(p_1)$; $N_2 \leftarrow \mathsf{next}_2(p_2)$;
8 **foreach** $(n_1, n_2) \in N_1 \times N_2$ **do**
9 **if** $\boxed{(n_1, n_2) \in \mathsf{P}}$ **then** `//` $\boxed{[\![(n_1, n_2)]\!] \subseteq [\![\overline{\mathsf{P}}]\!]}$
10 $n_1.\mathsf{color} \leftarrow \mathsf{black}$; $n_2.\mathsf{color} \leftarrow \mathsf{black}$;
11 **if** $\forall n \in N_1 \cup N_2.\ n.\mathsf{color} = \mathsf{black}$ **then**
12 **return** *true*;
13 **return** *false*
14 `// Returns cut-successors of` n
15 **Function** $\mathsf{next}_i(n)$:
16 $N \leftarrow \{n\}$; $\mathit{Ret} \leftarrow \emptyset$;
17 **while** N *is not empty* **do**
18 choose n from N; $N \leftarrow N \setminus \{n\}$;
19 $N' \leftarrow \{n' \mid n \ \boxed{\to_i}\ n'\}$; `//` $\boxed{\overline{\Rightarrow_i}}$
20 **foreach** $n' \in N'$ **do**
21 **if** $\boxed{n' \in C_i}$ **then** `//` $\boxed{[\![n']\!] \subseteq [\![\overline{C_i}]\!]}$
22 $n'.\mathsf{color} \leftarrow \mathsf{red}$;
23 $\mathit{Ret} \leftarrow \mathit{Ret} \cup \{n'\}$;
24 **else**
25 $N \leftarrow N \cup \{n'\}$;
26 **return** *Ret*;

Algorithm 3.1: Equivalence checking algorithm. For checking cut-simulation, replace $N_1 \cup N_2$ with N_1 at line 11. As given, the algorithm works with concrete data and thus is not practical. Replace boxed expressions with their grayed variants to the right for a practical, symbolic algorithm, as implemented in KEQ.

Theorem 3.3 (Correctness of Algorithm 3.1)**.** *Suppose that cut transition systems* $T_i = (S_i, \xi_i, \to_i, C_i)$ *are finitely branching (*$i \in \{1, 2\}$*) and* $\mathsf{P} \subseteq \mathcal{A}$ *is recursively enumerable with* $(\xi_1, \xi_2) \in \mathsf{P}$*. If Algorithm 3.1 does not terminate with false, then* $T_1 \sim_{\mathcal{A}} T_2$*.*

Indeed, suppose that Algorithm 3.1 does not terminate with *false*. Then none of the $\mathsf{check}(p_1, p_2)$ calls (line 3) terminates with *false* (line 13). Since the loop at Line 8 always terminates (T_1 and T_2 are finitely branching), it means that all nodes are colored black at line 11. Therefore, for each $(p_1, p_2) \in \mathsf{P}$, each cut-successor of p_1 can be paired in P with a cut-successor of p_2, and vice versa. Then P is a cut-bisimulation (Definition 3.3), that is, $T_1 \sim_{\mathcal{A}} T_2$ (Definition 3.5).

Note that Algorithm 3.1 may also terminate with *true*, namely when P is finite. Unfortunately, P is not expected to be finite in practice. For example, P may include all the synchronization points at the beginning of the main loop in a reactive system implementation. Nevertheless, in practice it is often the case that we can over-approximate infinite sets symbolically. For example, we can use a logical formula φ to describe a symbolic state, which denotes a potentially infinite set $[\![\varphi]\!]$ of concrete states that satisfy it. Then we may be able to describe the sets of states S_i and C_i of the cut transition systems T_i ($i \in \{1, 2\}$) with finite sets $\overline{S_i}$ and $\overline{C_i}$, respectively, of symbolic states. Similarly, symbolic pair (φ, φ') can describe infinite sets $[\![(\varphi, \varphi')]\!]$ of pairs of states in the two transition systems, related through free/symbolic variables that φ and φ' can share. Then we may also be able to describe P as a finite set $\overline{\mathsf{P}}$ of pairs of symbolic states. If all these are possible, then Algorithm 3.1 can be modified by replacing the boxed expressions with their symbolic variants (grayed); n, n', n_1, n_2, p_1, p_2, etc., are symbolic now.

Once an operational semantics of a programming language is given, $\mathbb{K}$ provides us with an API to calculate symbolic successors of symbolic program configurations. This allows us to conveniently implement the symbolic $\overline{\Rightarrow_i}$ transitions at line 19. Also, $\mathbb{K}$ is fully integrated with the Z3 solver [29], allowing us to implement the inclusions at lines 9 and 21 by requesting Z3 to solve the implications of the corresponding formulae. It is clear that the symbolic variant of Algorithm 3.1 terminates provided that Z3 terminates. Also, working symbolically allows us to usually eliminate the restriction that T_i must be finitely branching, as infinite branching can often be modeled symbolically (e.g., a random number generator can be modeled as a fresh symbolic variable).

We implemented the symbolic variant of Algorithm 3.1 in a tool called Kᴇq for checking language-independent program equivalence.[5]

To illustrate how Kᴇq works, consider the example in Figure 3.3. At the beginning of the programs, we have the symbolic synchronization point p_{init} which is a triple $(s_{p_{\mathsf{init}}}, s'_{p_{\mathsf{init}}}, \psi_{p_{\mathsf{init}}})$, as shown in Figure 3.4. $s_{p_{\mathsf{init}}}$ and $s'_{p_{\mathsf{init}}}$ are the symbolic state of the first and second program, respectively ($*$ is a separator for map bindings), and $\psi_{p_{\mathsf{init}}}$ is

[5] Kᴇq also supports program refinement, but for simplicity we only discuss equivalence.

```
int foo(unsigned n) {
  int i = 0;
  while (i < n) {
    i = i + 1;
  }
  return i;
}
```

```
int foo(unsigned n) {
  int i = 0;
  while (i < n) {
    i = i + 2;
  }
  return i;
}
```

Figure 3.3: Program transformation example. Two programs are equivalent provided that n is an even natural number.

$$
\begin{aligned}
s_{p_{\mathsf{init}}} &\equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \quad \text{where } \ n \bmod 2 = 0 \\
s'_{p_{\mathsf{init}}} &\equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \quad \text{where } \ n' \bmod 2 = 0 \\
\psi_{p_{\mathsf{init}}} &\equiv n = n'
\end{aligned}
$$

$$
\begin{aligned}
s_{p_{\mathsf{loop}}} &\equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \quad \text{where } \ i \bmod 2 = 0 \ \wedge \ n \bmod 2 = 0 \\
s'_{p_{\mathsf{loop}}} &\equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \quad \text{where } \ i' \bmod 2 = 0 \ \wedge \ n' \bmod 2 = 0 \\
\psi_{p_{\mathsf{loop}}} &\equiv i = i' \ \wedge \ n = n'
\end{aligned}
$$

$$
\begin{aligned}
s_{p_{\mathsf{final}}} &\equiv \mathtt{i} \mapsto i \ * \ \mathtt{n} \mapsto n \\
s'_{p_{\mathsf{final}}} &\equiv \mathtt{i} \mapsto i' \ * \ \mathtt{n} \mapsto n' \\
\psi_{p_{\mathsf{final}}} &\equiv i = i' \ \wedge \ n = n'
\end{aligned}
$$

Figure 3.4: Symbolic synchronization points example

the constraint for $s_{p_{\mathsf{init}}}$ and $s'_{p_{\mathsf{init}}}$ to be related, essentially saying that the inputs of the two programs are the same and they are even. Mathematically, p_{init} denotes the set of infinitely many pairs of states $\{(\mathtt{i} \mapsto i * \mathtt{n} \mapsto n, \ \mathtt{i} \mapsto i' * \mathtt{n} \mapsto n) \mid i, i', n \in \mathbb{N} \ \wedge \ n \text{ is even}\}$. Also, we have a synchronization point p_{loop} at the beginning of each loop iteration (i.e., the loop head), which is a triple $(s_{p_{\mathsf{loop}}}, s'_{p_{\mathsf{loop}}}, \psi_{p_{\mathsf{loop}}})$, as shown in Figure 3.4. Finally, we have a synchronization point p_{final} at the end of the programs, which is a triple $(s_{p_{\mathsf{final}}}, s'_{p_{\mathsf{final}}}, \psi_{p_{\mathsf{final}}})$. Note that p_{final} is relaxed, capturing more states than the reachable states. This is allowed as long as it is admitted by the acceptability relation (in this case, equality between the same variables). Indeed, the more synchronization points are relaxed, the easier the compiler can generate them. Below we will show this relaxed synchronization point is enough to prove the equivalence.

```
int cnt(unsigned n) {
  int c = 0;
  int i = 0;
  while (i < n) {
    i = i + 1;
    c = c + 1;
  }
  return c;
}
```

```
int cnt(unsigned n) {
  int c = 0;
  int i = n;
  while (i > 0) {
    i = i - 1;
    c = c + 1;
  }
  return c;
}
```

Figure 3.5: Two equivalent programs with the out-of-order loop iteration.

Next we illustrate how KEQ symbolically runs Algorithm 3.1. Let $\mathsf{P} = \{p_{\mathsf{init}}, p_{\mathsf{loop}}, p_{\mathsf{final}}\}$. First, KEQ picks a point (say p_{init}) from P (line 2 of Algorithm 3.1) and executes the function check with it. In check, it first symbolically executes each program (lines 7 and 19) until they reach another synchronization point (line 21). In this case they reach states s_1 and s_1' that are matched by $s_{p_{\mathsf{loop}}}$ and $s'_{p_{\mathsf{loop}}}$ respectively, where:

$$s_1 = \mathtt{i} \mapsto 0 \ * \ \mathtt{n} \mapsto n \quad \text{where} \ \ n \bmod 2 = 0 \tag{3.1}$$

$$s_1' = \mathtt{i} \mapsto 0 \ * \ \mathtt{n} \mapsto n' \quad \text{where} \ \ n' \bmod 2 = 0 \tag{3.2}$$

KEQ checks if $(s_1, s_1', \psi_{p_{\mathsf{init}}})$ is matched by p_{loop} (line 9), which is true. Since s_1 is the only pair that reaches p_{loop}, the check function returns true (line 12).

Next, suppose KEQ picks p_{loop} (line 2). Symbolic execution starting from p_{loop} yields two pairs of symbolic traces, that reach synchronization points p_{loop} (through the for-loop body) and p_{final} (escaping the for-loop), respectively. Let us consider the first case. We have the pair of states s_2 and s_2' that are matched by $s_{p_{\mathsf{loop}}}$ and $s'_{p_{\mathsf{loop}}}$ respectively, where:

$$s_2 = \mathtt{i} \mapsto i + 2 \ * \ \mathtt{n} \mapsto n \quad \text{where} \ \ i \bmod 2 = 0 \ \wedge \ n \bmod 2 = 0 \tag{3.3}$$

$$s_2' = \mathtt{i} \mapsto i' + 2 \ * \ \mathtt{n} \mapsto n' \quad \text{where} \ \ i' \bmod 2 = 0 \ \wedge \ n' \bmod 2 = 0 \tag{3.4}$$

Note that s_2 is resulted from executing the loop twice, since the result of the single loop iteration ($\mathtt{i} \mapsto i + 1 \ * \ \mathtt{n} \mapsto n$) is not matched by $s_{p_{\mathsf{loop}}}$ because $(i + 1) \bmod 2 \neq 0$, that is, it is *not* in the cut (line 21). KEQ checks if $(s_2, s_2', \psi_{p_{\mathsf{loop}}})$ is matched by p_{loop} (line 9), which is true. (Here KEQ needs to rename the free variables in p_{loop} to avoid the variable capture.) The other case is similar and check with p_{loop} eventually returns true. Then, KEQ continues to pick from the remaining synchronization points and execute check with each of them (loop at lines 2-4), eventually returning true (line 5).

Figure 3.5 shows an example of equivalent programs with the out-of-order loop. The first program iterates the loop increasing the loop index $\mathtt{i}$, while the second program iterates decreasing $\mathtt{i}$. The cut-bisimulation is expressive enough to capture this out-of-order execution. The following three synchronization points are sufficient for KEQ to prove the equivalence:

- At the beginning: $\mathtt{n} = \mathtt{n}'$
- At the loop head: $\mathtt{n} = \mathtt{n}'$, $\mathtt{i} + \mathtt{i}' = \mathtt{n}$, and $\mathtt{c} = \mathtt{c}'$
- At the end: $\mathtt{c} = \mathtt{c}'$

where the primed variables refer to the second program's variables. Note that the non-trivial part of the synchronization points is the equality $\mathtt{i} + \mathtt{i}' = \mathtt{n}$, but the compiler can provide this information that must be known to perform such an out-of-order loop transformation.

3.3 APPLICATIONS

We present the use of cut-bisimulation and the program equivalence checker in two applications, translation validation and specification refinement.

3.3.1 Translation Validation

Most, if not all, of the high-profile production compilers do *not* provide a formal correctness guarantee of equivalence between the input and output programs. Indeed, the compiler bugs exist even in widely-used mature compilers [44]. Translation validation [45] is a compiler verification technique that aims to prove correctness of a single instance of compilation, by considering only the specific input and output programs and treating the compiler mostly as a black box. Translation validation is effective for compiler verification since it can be composed to validate a sequence of compilation steps, it can easily retrofit to existing high-profile production compilers, it can be maintained independently from the compiler itself, and it can be fully automated.

The cut-bisimulation based program equivalence checker can be used for translation validation. Here the compiler generates the synchronization points using the internal information used in the compilation, similar to the witness-based translation validation approach [40].

```
function transfer(address _to, uint256 _value) public returns (bool) {
  require(_value <= balances[msg.sender]);
  balances[msg.sender] = balances[msg.sender].sub(_value);
  balances[_to] = balances[_to].add(_value);
  Transfer(msg.sender, _to, _value);
  return true;
}
```

(a) Solidity ERC20 transfer function

```
@public
def transfer(_to : address, _value : num256) -> bool:
  _sender = msg.sender
  self.balances[_sender] = num256_sub(self.balances[_sender], _value)
  self.balances[_to] = num256_add(self.balances[_to], _value)
  log.Transfer(_sender, _to, _value)
  return true
```

(b) Vyper ERC20 transfer function

Figure 3.6: Two ERC20 transfer functions, one written in Solidity and another in Vyper

We demonstrate its usability for the Ethereum smart contract bytecode. We have instantiated the language-independent program equivalence checker KEQ by plugging-in the Ethereum Virtual Machine (EVM) [46] semantics, KEVM [11], and derived the EVM equivalence checker that can be used to check equivalence between two EVM programs (bytecode). Then, we have applied the EVM equivalence checker to validate compatibility between two different language implementations of the same smart contract. Specifically, we took two ERC20 token contracts—one written in Solidity [47], another written in Vyper [48]—and verified equivalence of their compiler EVM bytecode. Figure 3.6 shows the code snippet of the Solidity and Vyper ERC20 token contracts, and Figure 3.7 presents synchronization points of their EVM bytecode. Given the two EVM programs (bytecode) and the synchronization points, the EVM equivalence checker successfully verifies the equivalence.

3.3.2 Specification Refinement

When a high-level specification is refined into a low-level one, a refinement relation between them can be formulated as a cut-simulation. The idea is to consider a specification as a transition system and establish a cut-simulation relation between them. For example, a class specification can be seen as a transition system, where a node is a state of the

```
<k>          K          </k>           <wordStack>  STACK      </wordStack>
<schedule>   SCHEDULE   </schedule>     <localMem>   MEMORY     </localMem>
<output>     OUTPUT     </output>       <pc>         PC         </pc>
<statusCode> STATUS     </statusCode>   <gas>        GAS        </gas>
<callStack>  CALL_STACK </callStack>    <log>        LOG        </log>
<id>         ACCT_ID    </id>           <refund>     REFUND     </refund>
<caller>     CALLER_ID  </caller>       <balance>    BALANCE    </balance>
<callData>   CALL_DATA  </callData>     <storage>    STORAGE    </storage>
<callValue>  CALL_VALUE </callValue>
```

(a) Template for synchronization points

At the beginning:

- `K = K' = #execute`
- `SCHEDULE = SCHEDULE' = CONSTANTINOPLE`
- `STACK = STACK' = .WordStack`
- `MEMORY = MEMORY' = .Map`
- `PC = PC' = 0`
- equalities between other variables

At the end:

- `K = K' = #halt`
- `OUTPUT = OUTPUT'`
- `STATUS = STATUS'`
- `LOG = LOG'`
- `BALANCE = BALANCE'`
- `STORAGE` $\sim$ `STORAGE'`

(b) Synchronization points

Figure 3.7: Synchronization points for two EVM bytecode programs

class, and an edge is a pair of pre-/post-conditions of a method. This way, one can systematically reason about the soundness of specification refinements.

Moreover, the property preservation of cut-bisimulation enables to prove meta properties[6] of a refined specification by only showing the cut-similarity to the original specification that holds the meta-properties. This approach has values especially when the meta-properties are easier to be proved in the original specification than the refined specification.

We demonstrate the idea using the ERC20 specification. ERC20-K [49] formulates the ERC20 standard [13] at a high-level, and ERC20-EVM [50] specifies the behavior of its

[6]Indeed, many of security properties are formulated as meta-properties.

```
rule <k> transfer(To, Value) => true ...</k>
     <caller> From </caller>
     <account>
       <id> From </id>
       <balance> BalanceFrom => BalanceFrom -Int Value </balance>
     </account>
     <account>
       <id> To </id>
       <balance> BalanceTo => BalanceTo +Int Value </balance>
     </account>
     <log> Log => Log Transfer(From, To, Value) </log>
  requires To =/=Int From    // sanity check
   andBool Value >=Int 0
   andBool Value <=Int BalanceFrom
   andBool BalanceTo +Int Value <=Int MAXVALUE
```

(a) ERC20-K transfer specification

```
[transfer-success]
callData: #abiCallData("transfer", #address(TO), #uint256(VALUE))
statusCode: _ => EVMC_SUCCESS
output: _ => #asByteStackInWidth(1, 32)
log: ... (. => #abiEventLog(FROM, "Transfer",
    #indexed(#address(FROM)), #indexed(#address(TO)), #uint256(VALUE))))
storage:
  #hashedLocation({BALANCES}, FROM) |-> (BAL_FROM => BAL_FROM -Int VALUE)
  #hashedLocation({BALANCES}, TO) |-> (BAL_TO => BAL_TO +Int VALUE)
  ...
requires:
  andBool FROM =/=Int TO
  andBool VALUE <=Int BAL_FROM
  andBool BAL_TO +Int VALUE <Int (2 ^Int 256)
```

(b) ERC20-EVM transfer specification

Figure 3.8: Two ERC20 specifications: ERC20-K and ERC20-EVM

EVM bytecode implementation. We prove that ERC20-EVM is a sound refinement of ERC20-K by showing that ERC20-EVM cut-refines ERC20-K (i.e., ERC20-K cut-simulates ERC20-EVM). Figure 3.8 shows the snippet of the ERC20-K and ERC20-EVM specifications, and Figure 3.9 shows a cut-simulation relation between them. Figure 3.9a presents the configurations of two specifications.[7] Since the two configurations are structurally different, we need to specify when the two configurations are considered "the same". We define three notions of state equivalence, $\sim_k$, $\sim_l$, and $\sim_s$, that are used to specify

[7]For the simplicity of presentation, here we present only relevant part of the whole ERC20-EVM configuration. The full configuration can be found at [50].

```
<caller> CALLER </caller>
<k> K </k>
<log> LOG </log>
<accounts>
  ACCOUNTS   // id -> balance
</accounts>
<allowances>
  ALLOWANCES // (owner * spender) -> amount
</allowances>
<supply> SUPPLY </supply>
<resource> RESOURCE </resource>
```

```
<caller> CALLER </caller>
<callData> CALL_DATA </callData>
<statusCode> STATUS </statusCode>
<output> OUTPUT </output>
<log> LOG </log>
<storage> STORAGE </storage>
<gas> GAS </gas>
```

(a) Configurations of ERC20-K (left) and ERC20-EVM (right)

- $\texttt{CALLER} = \texttt{CALLER}'$
- $\texttt{K} \sim_k (\texttt{CALL\_DATA}, \texttt{STATUS}, \texttt{OUTPUT})$
- $\texttt{LOG} \sim_l \texttt{LOG}'$
- $(\texttt{ACCOUNTS}, \texttt{ALLOWANCES}, \texttt{SUPPLY}) \sim_s \texttt{STORAGE}$
- $\texttt{RESOURCE} \sim_r \texttt{GAS}$

(b) Cut-simulation relation

Figure 3.9: Cut-simulation relation of ERC20-K and ERC20-EVM

the cut-simulation in Figure 3.9b. In ERC20-K, we have three data structures, ACCOUNTS, ALLOWANCES, and SUPPLY, that represent the current status of the ERC20 token, while in ERC20-EVM, all of the data structures are stored in a single place STORAGE. Thus we define $\sim_s$ to relate the three logical data structures to the single storage as follows. For any ACCOUNTS b, ALLOWANCES a, SUPPLY t, and STORAGE s, we write $(b, a, t) \sim_s s$ iff the following holds:

- s includes b, a, and t: $\forall i \in \mathsf{dom}(b).\ b(i) = s(\mathsf{hash}_b(i))$, $\forall (o, s) \in \mathsf{dom}(a).\ a(o, s) = s(\mathsf{hash}_a(o, s))$, and $t = s(\mathsf{hash}_t)$.
- s includes *only* b, a, and t: $\forall l.\ s(l) \neq 0 \implies (\ \exists i \in \mathsf{dom}(b).\ \mathsf{hash}_b(i) = l \vee\ \exists (o, s) \in \mathsf{dom}(a).\ \mathsf{hash}_a(o, s) = l \vee\ \mathsf{hash}_t = l)$
- No hash collision: $\forall i \in \mathsf{dom}(b), (o, s) \in \mathsf{dom}(a).\ \mathsf{hash}_b(i) \neq \mathsf{hash}_a(o, s) \neq \mathsf{hash}_t \neq \mathsf{hash}_b(i)$.

In ERC20-K, the <k> cell is used to hold both the function call and its return value, while in ERC20-EVM, they are separated into multiple cells. We define $\sim_k$ to relate these

cells. First, when the <k> cell holds the function call, we relate it to the `callData` cell. For example, for the `transfer` function, $\sim_k$ is defined as follows:

$$\forall t, v, s, o.\ \texttt{Transfer}(t, v) \sim_k (\texttt{\#abiCallData("transfer",\#address}(t)\texttt{,\#uint256}(v)\texttt{)}, s, o) \tag{3.5}$$

Second, when the <k> cell holds the return value, we relate it to the `statusCode` and `output` cells, as follows:

$$\forall c.\ \texttt{true} \sim_k (c, \texttt{EVMC\_SUCCESS}, \texttt{\#asByteStackInWidth(1, 32)}) \tag{3.6}$$

$$\forall c, o.\ \texttt{throw} \sim_k (c, \texttt{EVMC\_REVERT}, o) \tag{3.7}$$

Similarly, we define $\sim_l$ to relate two different forms of logs for each type of events. For example, for the `Transfer` event, $\sim_l$ is defined as follows:

$$\begin{aligned}\forall f, t, v.\ \texttt{Transfer}(f, t, v) \sim_l \texttt{\#abiEventLog}(f, \texttt{"Transfer"}, \texttt{\#indexed(\#address}(f)\texttt{)},\\ \texttt{\#indexed(\#address}(t)\texttt{),\#uint256}(v)\texttt{)}\end{aligned} \tag{3.8}$$

We also define $\sim_r$ to relate their resources. ERC20-K abstracts the resource, only specifying the fact that each function execution requires sufficient resources, and the amount of resources decreases for each function execution even if the execution fails. This resource requirement and its limited availability ensures to prevent infinite execution. In ERC20-EVM, however, the resource is realized, consisting of several components, e.g., gas (an execution fee for every operation made on EVM), and call stack (the stack of nested function calls). The two forms of resources are related when they are sufficient to execute the same sequence of functions.

Property Preservation Now we present an example use of the property preservation of the cut-simulation given in Figure 3.9. First, we lift the cut-simulation $\geq$ to be right-total by introducing a bad state of ERC20-K that has no transition, and relating it to all the invalid states of ERC20-EVM. Then, let `total` and `total`$'$ be a function that returns the total amount of balances of the ERC20-K and ERC20-EVM states, `ERC20-K` and `ERC20-EVM`, respectively, which can be defined as follows:

$$\texttt{total}(\texttt{ERC20-K}) = \Sigma\{v \mid k \mapsto v \in \texttt{ERC20-K}_{\texttt{<accounts>}}\} \tag{3.9}$$

$$\texttt{total}'(\texttt{ERC20-EVM}) = \Sigma\{\texttt{ERC20-EVM}_{\texttt{<storage>}}(k') \mid \exists k.\ \mathsf{hash}_b(k) = k'\} \tag{3.10}$$

where $\texttt{ERC20-K}_{\texttt{<accounts>}}$ and $\texttt{ERC20-EVM}_{\texttt{<storage>}}$ refer to the content of the <accounts> and <storage> cells of the ERC20-K and ERC20-EVM configurations, respectively. Similarly, let supply and supply′ be a function that returns the total supply, defined as follows:

$$\texttt{supply}(\texttt{ERC20-K}) = \texttt{ERC20-K}_{\texttt{<supply>}} \tag{3.11}$$

$$\texttt{supply}'(\texttt{ERC20-EVM}) = \texttt{ERC20-EVM}_{\texttt{<storage>}}(\mathsf{hash}_t) \tag{3.12}$$

Now we have:

$$\texttt{ERC20-K} \geq \texttt{ERC20-EVM} \implies (\ \texttt{total}(\texttt{ERC20-K}) = \texttt{total}'(\texttt{ERC20-EVM}) \tag{3.13}$$

$$\wedge\ \texttt{supply}(\texttt{ERC20-K}) = \texttt{supply}'(\texttt{ERC20-EVM})\ ) \tag{3.14}$$

Now, let $P_{\texttt{total,supply}}$ and $P_{\texttt{total}',\texttt{supply}'}$ be predicates over the ERC20-K and ERC20-EVM states, respectively, defined as follows:

$$P_{\texttt{total,supply}}(\texttt{ERC20-K}) \stackrel{\texttt{def}}{=} \texttt{total}(\texttt{ERC20-K}) = \texttt{supply}(\texttt{ERC20-K}) \tag{3.15}$$

$$P_{\texttt{total}',\texttt{supply}'}(\texttt{ERC20-EVM}) \stackrel{\texttt{def}}{=} \texttt{total}'(\texttt{ERC20-EVM}) = \texttt{supply}'(\texttt{ERC20-EVM}) \tag{3.16}$$

Then, if $P_{\texttt{total,supply}}$ is inductive, then $P_{\texttt{total}',\texttt{supply}'}$ is also inductive, by Theorem 3.2. This means that we only need to show that $P_{\texttt{total,supply}}$ is an inductive invariant since $P_{\texttt{total}',\texttt{supply}'}$ follows immediately.

CHAPTER 4: DEDUCTIVE PROGRAM VERIFICATION

As a comprehensive evaluation of the universal formal methods and performance of their derived formal analysis tools, we have instantiated a language-independent deductive program verifier [12] by plugging-in four real-world language semantics, C, Java, JavaScript, and Ethereum Virtual Machine (EVM); and used them to verify full functional correctness of challenging heap-manipulating programs and high-profile commercial smart contracts for the end-to-end verification, demonstrating their scalability and practicality. Both JavaScript and EVM verifiers are the first deductive program verifier for the languages to the best of our knowledge. Much of the content in this chapter comes from Park *et al.* [51] and Stefanescu *et al.* [12].

4.1 LANGUAGE-PARAMETRIC PROGRAM VERIFIER

We evaluate the language-independent verification framework that can be instantiated with an operational semantics to automatically generate a program verifier. The framework treats both the operational semantics and the program correctness specifications as reachability rules between matching logic patterns, and uses the sound and relatively complete reachability logic proof system to prove the specifications using the semantics. We instantiate the framework with the semantics of three real-world languages, C, JAVA, and JAVASCRIPT, developed independently of our verification infrastructure. We evaluate our approach empirically and show that the generated program verifiers can check automatically the full functional correctness of challenging heap-manipulating programs implementing operations on list and tree data structures, like AVL and red-black trees. This is the first work that turns the operational semantics of real-world languages into correct-by-construction automatic verifiers.

We build the framework on the *theoretical* work [5, 4, 2, 1, 3] that proposes a *language-independent proof system* which derives program properties directly from an operational semantics, *at the same granularity and compositionality* as a language-specific axiomatic semantics. Specifically, it introduces *(one-path) reachability rules*, which generalize operational semantics reduction rules, and *(all-path) reachability rules*, which generalize Hoare triples. Then, it gives a proof system that derives new reachability rules (program properties) from a set of given reachability rules (the language semantics).

Figure 4.1 describes the architecture of our verification framework. We developed it as part of the $\mathbb{K}$ framework [27] (`http://kframework.org`), in which the semantics of the

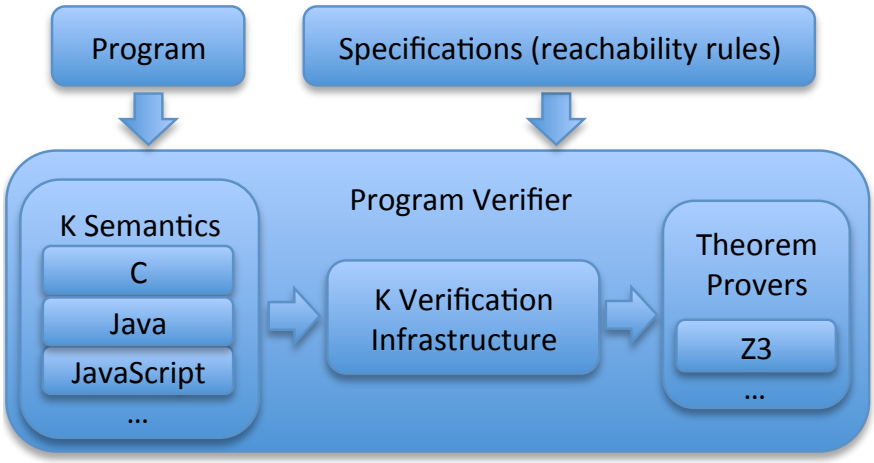


Figure 4.1: Architecture of Semantic-Based Verification

above languages were defined. However, the technique applies to any reduction-based semantics. Our $\mathbb{K}$ verification infrastructure takes an operational semantics given in $\mathbb{K}$ and generates queries to a theorem prover (for example, Z3 [29]). The program correctness properties are given as reachability rules between matching logic patterns [20]. Internally, the verifier uses the operational semantics to perform symbolic execution. Also, it has an internal matching logic prover for reasoning about implication between patterns (states), which reduces to SMT reasoning.

Our hypothesis is that many of the tricky language-specific details (type systems, scoping, implicit conversions, etc) are orthogonal to features that make program verification hard (reasoning about heap-allocated mutable data structures, integers/bit-vectors/floating-point numbers, etc). As such, we propose a methodology to separate the two: (1) define an operational semantics, and (2) implement reasoning in the language-independent infrastructure.

To validate our approach, we first developed our verification infrastructure using it only in connection with KernelC, a small C-like language, during development. Then, we evaluated it with the operational semantics of C, Java, and JavaScript, checking the full functional correctness of challenging heap manipulation programs implementing the same data-structures in C, Java, and JavaScript. The verifiers were successful in automatically proving all the programs correct, and the verification times are competitive. The times are dominated by symbolic execution, which reflects the complexity of the operational semantics (Section 4.1.3). Further, the development time required to write the specifications, including the semantics-specific details, and to fix bugs in the semantics was negligible compared to the development time of these semantics (Section 4.1.4).

```
1  struct node {
2      int value;
3      struct node *left, *right;
4  };
5
6  struct node* new_node(int v) {
7      struct node *node;
8      node = (struct node *) malloc(sizeof(struct node));
9      node->value = v;
10     node->left = NULL;
11     node->right = NULL;
12     return node;
13 }
14
15 struct node* insert(int v, struct node *t) {
16     if (t == NULL)
17         return new_node(v);
18     if (v < t->value)
19         t->left = insert(v, t->left);
20     else if (v > t->value)
21         t->right = insert(v, t->right);
22     return t;
23 }
```

Figure 4.2: Binary search tree code snippet in C

The semantics of C, Java, and JavaScript were developed independently from the verification infrastructure in the sense that they were developed with the goal of giving a straightforward yet complete operational-style semantics for each of these languages, without verification in mind. The verification infrastructure was developed without detailed knowledge of the semantics. This makes us confident that our verification infrastructure would work with future semantics with only minimal changes.

4.1.1 Illustrating Example

Here we illustrate our approach by checking the correctness of binary search tree (BST) insertion implemented in C, Java, and JavaScript. A BST is a tree where the value stored in each node is greater than any value in the left subtree and less than any value in the right subtree. Insert recursively traverses the tree and adds a new leaf with the value, if the value is not already in the tree. We use the operational semantics of these languages for symbolic execution, and delegate reasoning about trees in the heap and BST invariants to the verification infrastructure. Although the three definitions feature

```
1  class Node {
2      int value;
3      Node left, right;
4
5      public Node(int value) {
6          this.value = value;
7          left = right = null;
8      }
9
10     public static Node insert(int v, Node t) {
11         if (t == null)
12             return new Node(v);
13         if (v < t.value)
14             t.left = insert(v, t.left);
15         else if (v > t.value)
16             t.right = insert(v, t.right);
17         return t;
18     }
19 }
```

Figure 4.3: Binary search tree code snippet in JAVA

```
1  function make_node(v) {
2      var node = {
3          value : v,
4          left : null,
5          right : null
6      };
7      return node;
8  }
9
10 function insert(v, t) {
11     if (t === null)
12         return make_node(v);
13     if (v < t.value)
14         t.left = insert(v, t.left);
15     else if (v > t.value)
16         t.right = insert(v, t.right);
17     return t;
18 }
```

Figure 4.4: Binary search tree code snippet in JAVASCRIPT

different language constructs and memory models, the operational semantics successfully abstracts these details.

Figure 4.2, 4.3, and 4.4 show the implementation in C, JAVA, and JAVASCRIPT. C uses "`struct node`" to represent a tree node, while JAVA uses "`class Node`". JAVASCRIPT is a class-free, prototypal language, where objects dynamically inherit from other objects.

In C, dynamically allocated memory (the "heap") is untyped; `malloc` allocates a block of bytes, which is then associated the effective type **`struct`** `node`. In JAVA all memory is typed; `new` creates an instance of `class Node`. In JAVASCRIPT, objects are modeled in memory as maps from property names (strings) to values (of any type). Each language has different memory access mechanisms. The C and JAVA trees store integers, while the JAVASCRIPT tree stores floats (JAVASCRIPT integers are syntactic sugar for floats). Other language-specific aspects are automatic type conversions and function/method calls.

Before we discuss the correctness specifications, we introduce some useful $\mathbb{K}$ conventions. Specifications are reachability rules $\varphi \Rightarrow^{\forall} \varphi'$, with φ and φ' matching logic patterns (i.e. (symbolic) program configurations with constraints). If φ and φ' share program configuration context, we only mention the context once and distribute "$\Rightarrow^{\forall}$" through the context where the changes take place. Logical variables starting with "?" are existentially quantified. Rules only mention the parts of the configuration they read or write; the rest stays unchanged. The "requires" clause is implicitly conjuncted with the left-hand-side configuration, and "ensures" with the right-hand-side. It is common for operational semantics to have a preprocessing/initializing phase. C computes structure and function tables, JAVA a class table, while JAVASCRIPT creates objects and environments for all functions. A variable with the same name as a cell but with capital letters is a placeholder for the initial value of that cell after the preprocessing phase, which we statically compute using the semantics.

Figure 4.5 shows the correctness specifications. We discuss the C one first. The rule states that the call to `insert` with value V and pointer L_1 returns pointer $?L_2$. Since C is typed, each value is tagged with its type, in this case **`int`** or pointer to **`struct`** `node`. When the function is called, the memory contains a binary tree with root L_1 storing the algebraic tree T_1. When the function returns, the initial tree is replaced by another tree with root $?L_2$ storing $?T_2$. The requires clause states that T_1 is a BST and V is in the appropriate range for signed 32-bit integers. The ensures clause states that T_2 is also a BST, and the value set of $?T_2$ is the value set of T_1 union with V. The "$\cdots$" in the mem cell stands for a variable matching the rest of the memory (the *heap frame*), which stays unchanged. Similarly, the parts of the program configuration that are not explicitly mentioned (the *configuration frame*) do not change. The threads cell contains only one thread and no "$\cdots$", which means this program is verified in a single-threaded environment (the program is not thread-safe). Variables FUNCTIONS, STRUCTS, and MEM are placeholders for the tables of function declarations and structure declarations, and the initial memory layout. Note that here we assume signed integers are represented on 32 bits. The C standard

C

rule
⟨functions⟩ FUNCTIONS:Map ⟨/functions⟩
⟨structs⟩ STRUCTS:Map ⟨/structs⟩
⟨mem⟩...
MEM:Map (tree(L_1, T_1:Tree) $\Rightarrow^{\forall}$ tree(?L_2, ?T_2:Tree))
...⟨/mem⟩
⟨threads⟩ ⟨thread⟩... ⟨k⟩
`insert`(tv(V:Int, `int`), tv(L_1:Loc, `struct node`))
$\Rightarrow^{\forall}$ tv(?L_2:Loc, `struct node`)
...⟨/k⟩ ...⟨/thread⟩ ⟨/threads⟩
requires bst(T_1) $\wedge$ $-2147483648 \leq V \wedge V \leq 2147483647$
ensures bst(?T_2) $\wedge$ tree_keys(?T_2) $= \{V\} \cup$ tree_keys(T_1)

Java

rule
⟨classes⟩ CLASSES:Bag ⟨/classes⟩
⟨objectStore⟩...
tree(R_1, T_1:Tree) $\Rightarrow^{\forall}$ tree(?R_2, ?T_2:Tree)
...⟨/objectStore⟩
⟨threads⟩ ⟨thread⟩... ⟨k⟩
`(class Node).insert(`
V:Int :: `int`, R_1:Ref :: `class Node`)
$\Rightarrow^{\forall}$?R_2:Ref :: `class Node`
...⟨/k⟩ ...⟨/thread⟩ ⟨/threads⟩
requires bst(T_1) $\wedge$ $-2147483648 \leq V \wedge V \leq 2147483647$
ensures bst(?T_2) $\wedge$ tree_keys(?T_2) $= \{V\} \cup$ tree_keys(T_1)

JavaScript

rule
⟨envs⟩... ENVS:Bag (.Bag $\Rightarrow^{\forall}$?_:Bag) ...⟨/envs⟩
⟨objs⟩...
OBJS:Bag (.Bag $\Rightarrow^{\forall}$?_:Bag)
(tree(L_1:Loc, T_1:Tree) $\Rightarrow^{\forall}$ tree(?L_2:Loc, ?T_2:Tree))
...⟨/objs⟩
⟨k⟩ `insert`(V:Float, O_1:Object) $\Rightarrow^{\forall}$?O_2:Object ...⟨/k⟩
requires bst(T_1) $\wedge$ $\neg$isNaN(V)
ensures bst(?T_2) $\wedge$ tree_keys(?T_2) $= \{V\} \cup$ tree_keys(T_1)

Figure 4.5: Binary search tree correctness specifications for C, Java, and JavaScript

allows other choices (e.g., 64 bits), and we can handle those by modifying the require clause with the appropriate value range.

The Java specification is in many ways similar to the C one, reflecting the similarities between C and Java. The call to `insert` uses the fully qualified method name, which

includes the class name `Node`. The type of R_1 and $?R_2$ mentioned in the rule is the static type of these references, `class Node`. The dynamic type can be any sublcass of `class Node`. Variable CLASSES:Bag stands for the statically computed class table.

Finally, we discuss the JavaScript specification. Since JavaScript is untyped, its values do not carry a type. V is not `NaN`, since `NaN` does not respect the order relation on non-`NaN` floats, and the code is incorrect if V or the values in T_1 were `NaN`. The JavaScript semantics creates new environments and objects at function call, which it does not garbage-collect at return (an artifact of the semantics rather than of the language). The ".Bag $\Rightarrow^{\forall}$?_:Bag" in both the envs and objs cells states that there may be garbage left after the function returns (“.” is the unit, while “_” is an anonymous variable, here existentially quantified). JavaScript does not have threads.

The tree heap abstraction is defined in matching logic, and is different for each language, taking into account the specifics of the memory model of each language. Also bst, tree_keys, etc., are domain operation symbols in the signature.

At a high level, the three specifications are very similar. The differences are down to language-specific and semantics-specific details: type systems, name resolution, garbage collection, or the statically computed information by each semantics. The tree heap abstraction hides the differences in memory models. Our generic verification infrastructure reasons about the tree abstraction and the mathematical properties of BST while deferring the symbolic execution to the semantics. The verification is fully automated and takes a few seconds (see Table 4.1 in Section 4.1.3).

It is possible to generate the specification rules automatically from classic verification annotations (pre/post conditions, loop invariants, class invariants, etc). This has been done previously by MatchC [3]. We have not implemented this feature, using instead a general-purpose notation which is faithful to both reachability logic and our implementation.

4.1.2 Evaluation

We evaluate the $\mathbb{K}$ verification infrastructure by instantiating it with four different semantics, thus obtaining program verifiers for four different languages: KernelC (a small C-like language), C, Java, and JavaScript (complex real-world languages). Our goal is to validate our hypothesis that building program verifiers by using $\mathbb{K}$ operational semantics and its verification infrastructure is effective both in terms of verification capabilities and tool building effort. To evaluate this hypothesis, first we implemented all the features required to verify the programs in Table 4.1 with KernelC:

symbolic execution, reasoning with heap abstractions, integration with Z3, etc. Then we instantiated our framework with the off-the-shelf semantics of C11 [23, 24], JAVA 1.4 [52], and JAVASCRIPT 5.1 [7] to obtain corresponding program verifiers. We evaluated these verifiers by proving the correctness of the same programs in Table 4.1, but written in C, JAVA, and JAVASCRIPT. The implementation and the experiments are available as part of the $\mathbb{K}$ framework at `http://github.com/kframework/k/wiki/Program-Verification`.

The semantics we use are the most complete to date for their languages (see Table 4.2 for their size). As we mentioned before, given the complexity of real-world languages, we would like to separate the tricky language-specific features that are orthogonal to the verification process from the language-independent issues that make program verification hard. We achieve this by deferring to the semantics to handle the language-specific features (automatic promotions of integers in C, type checking, function call resolution, etc.). The $\mathbb{K}$ verification infrastructure handles the language-independent reasoning (heap-allocated mutable data structures, integers/bit-vectors/floating-points, etc.).

4.1.3 Verification Result

Here we discuss how effective in terms of proving capabilities it is to build program verifiers using $\mathbb{K}$ operational semantics. To this end, we have verified using our approach a number of challenging heap manipulating programs implementing the same data structure operations in KERNELC, C, JAVA, and JAVASCRIPT. These programs have been used before to evaluate verification approaches, e.g., in [3, 17, 53, 31]. Our goal here is to show that we can also verify such programs at comparable performance, but in a language-independent setting. We conducted the experiments on a machine with Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB.

Our examples fall in two categories. (1) Singly-linked list manipulating programs, including implementations of common sorting algorithms. For each sorting function, we prove that the returned sequence is indeed sorted and has exactly the same elements as the original sequence. (2) Implementations of binary search tree, AVL tree, red-black tree (RBT), and Treap data-structure operations. For each function, we prove that it maintains the data-structure invariants and that the set of elements is as expected.

Table 4.1 summarises our experiments. For KERNELC, which is idealized for verification, proving the implications required by CONSEQUENCE (shown in the Reasoning column) dominates the total verification time. C, JAVA, and JAVASCRIPT are complex languages, so the semantics-based symbolic execution (shown in the Execution column) dominates the verification time. Note that since the programs implement the same data structure opera-

Programs	KernelC				C				Java				JavaScript			
	Execution		Reasoning		Execution		Reasoning		Execution		Reasoning		Execution		Reasoning	
	Time	#Step	Time	#Query	Time	#Step	Time	#Query	Time	#Step	Time	#Query	Time	#Step	Time	#Query
BST find	0.6	192	1.2	95	10.4	1,028	3.6	246	1.9	322	2.8	244	4.5	1,736	1.8	93
BST insert	0.8	336	2.9	160	23.0	2,481	7.2	414	4.1	691	4.5	342	5.4	3,394	2.8	158
BST delete	1.4	582	5.6	420	55.1	4,540	16.6	938	9.8	1,274	15.1	1,125	15.6	5,052	5.6	373
AVL find	0.6	192	1.2	95	9.9	1,028	3.1	214	2.2	322	2.7	244	4.5	1,736	1.9	93
AVL insert	6.2	1,980	42.1	1,133	210.7	12,616	70.6	1,865	42.4	3,753	62.8	2,146	102.5	26,977	32.5	1,221
AVL delete	9.5	2,933	45.4	1,758	514.8	26,003	118.9	3,883	122.2	8,144	149.4	4,866	184.3	38,591	55.3	2,233
RBT find	0.6	192	1.1	95	11.5	1,064	3.0	214	2.1	322	2.9	244	4.9	1,736	1.9	93
RBT insert	7.6	2,331	48.1	1,392	722.0	30,924	181.8	4,394	39.9	4,240	75.7	2,547	84.9	28,082	29.6	1,381
RBT delete	10.6	3,891	33.7	2,033	1593.8	50,389	308.3	15,429	95.8	8,312	75.4	4,460	144.2	51,356	39.4	2,009
Treap find	0.6	200	1.4	118	11.2	1,064	3.2	214	2.0	322	2.9	244	4.6	1,736	1.9	116
Treap insert	1.4	753	4.5	247	52.4	4,954	15.3	724	12.7	1,469	10.4	563	13.7	7,738	5.2	243
Treap delete	2.0	831	9.4	509	73.9	5,512	16.5	656	12.0	1,694	16.4	1,021	24.8	8,333	8.4	460
List reverse	0.4	142	0.3	21	6.6	815	4.8	76	1.5	222	2.6	46	5.0	1,162	0.5	20
List append	0.4	171	0.5	45	7.4	909	7.4	128	1.8	239	5.5	106	4.5	1,392	0.8	46
Bubble sort	0.9	391	26.8	190	28.4	2,401	38.0	357	3.4	589	35.4	345	5.6	2,688	25.7	145
Insertion sort	1.1	468	24.5	300	26.6	2,555	35.3	451	4.1	731	27.0	371	8.3	3,119	36.5	213
Quick sort	1.1	604	31.6	269	31.0	3,601	48.2	518	7.1	958	40.0	413	15.0	5,046	33.1	252
Merge sort	1.7	970	55.0	478	81.6	6,589	89.0	1,070	14.1	1,566	72.9	737	22.8	7,021	43.2	480
Total	47.7	17,159	335.2	9,358	3470.5	158,473	970.6	31,791	379.3	35,170	604.5	20,064	654.9	196,895	326.3	9,629

Table 4.1: Summary of verification experiments: 'Execution' shows time (seconds) and number of operational semantic steps for symbolic execution (Section 2.3.1); 'Reasoning' shows time (seconds) and number of Z3 queries for reasoning (Section 2.3.2 & 2.3.3).

	C	Java	JavaScript
Semantics development (months)	40	20	4
Semantics size (#rules)	2,572	1,587	1,378
Semantics size (LOC)	17,791	13,417	6,821
Language-specific effort (days)	7	4	5
Semantics changes size (#rules)	63	38	12
Semantics changes size (LOC)	468	95	49
Specifications	36	31	31
Abstractions	6	6	6
Function definitions	14	14	14
Lemmas	7	7	7

Table 4.2: The development costs

tions in different languages, the complexity of implications required by Consequence tends to be similar. Thus, the complexity of the operational semantics is the most important factor contributing to the difference in the verification times reported. As expected, since C has the most complex operational semantics, the times for C are the largest. The number of queries of logical reasoning for C and Java is higher than for JavaScript because of 32-bit integer range constraints, while the time spent on each query is similar along the different languages, reflecting that the reasoning is language-independent. Furthermore, each step of symbolic execution for JavaScript is much smaller than for C and Java, because the JavaScript semantics is more fine-grained.

The AVL and RBT insert and delete programs take considerably longer than the other programs because some of the auxiliary functions (like balance, rotate, etc) are not given specifications and thus their bodies are being inlined, resulting in a larger number of paths to analyze. To put this in perspective, VCDryad [17], a state-of-the-art separation logic verifier for C build on top of VCC, takes 260s to verify only the balance function in AVL, while it takes our generic infrastructure instantiated with the C semantics 210s to verify AVL insert (including balance). In general, we believe Table 4.1 suggests that our approach is practical and competitive with the state-of-the-art on such data-structures.

4.1.4 Development Cost

We discuss how cost effective in terms of tool development it is to build program verifiers using $\mathbb{K}$ operational semantics and our verification infrastructure. Recall that the semantics of C, Java, and JavaScript were developed as separate projects, independently from the verification infrastructure.

Table 4.2 shows the development effort of our approach. The language-specific effort consists of familiarizing with the semantics in order to be able to write the correctness specifications as reachability rules (like the ones in Section 4.1.1), and of making changes to the semantics. Most of changes to the semantics are bug fixes, but some are performance improvements or simplifications. The development effort scales with the language complexity. The effort for C is considerably larger than for JAVA and JAVASCRIPT due to the low level complexity of C. Overall, the numbers in Table 4.2 validate our hypothesis that program verification based on operational semantics and the $\mathbb{K}$ verification infrastructure is cost effective in terms of development effort.

For comparison, the state-of-the-art is to define a translator to an intermediate verification language, like Boogie, or to define a verification condition (VC) generator. For example, the VCC translator from C to Boogie consists of approximately 5000 lines of F# [54]. We believe that writing such a translator takes considerably more effort than we reported for our approach in Table 4.2 (we do not include the time to define the semantics into this comparison, since we assume the semantics already exist, and they serve other purposes as well). Moreover, we believe that one would have more confidence in an operational semantics to handle the tricky details of complex languages than in a translation or a VC generator, for two reasons. First, an operational semantics is more amenable to visual inspection, as it is written in a domain-specific language for defining semantics. Second, an operational semantics is executable and can be thoroughly tested. While this does not guarantee the absence of bugs, it greatly reduces their occurrence.

Even if a semantics is not already available, we believe that developing an operational semantics has an important advantage over building a translator or a VC generator: the semantics is used not only for verification, but for other purposes as well, so overall the semantics development cost is amortized. For example, the JAVASCRIPT semantics was used for bug finding in browsers [7].

Regarding number of annotations, our approach is comparable to the state-of-the-art language-specific approaches that do not infer invariants (VCC, Frama-C). The user provides one specification for each recursive function and loop. The user also provides the definitions for heap abstractions and auxiliary functions used in specifications. The user does not provide anything similar to ghost code or hints for the verifier. The user may need to provide additional lemmas and those lemmas apply to a class of programs rather than one particular program (e.g., the lemmas for list segments in Section 2.3.2 are shared by all sorting-related programs in all languages).

Operational Semantics Bugs We found bugs in all the three operational semantics used for verification, despite the fact that these semantics are thoroughly tested on thousands of programs [23, 24, 52, 7].

The main source of bugs is the unintended non-determinism in the semantics. A semantics models a non-deterministic feature by having multiple rules that can apply at the same time. Such a feature is the expression evaluation order in C: "`f() + g()`" may call `f()` first and `g()` second or `g()` first and `f()` second. As a result, only a fraction of the possible behaviors are observed under testing. During symbolic execution, the $\mathbb{K}$ verifier considers all the rules that can apply (according to `nextAll` in Figure 2.1). This revealed that each semantics contained unintended non-determinism: pairs of rules where the semantics developers intended for one rule to always apply before the other, but in fact both rules can apply simultaneously. Applying the rules in the other order causes an incorrect result. We also found other kinds of bugs, mostly caused by incorrect side conditions of the semantics rules, or incorrect assumptions about the configuration.

4.2 END-TO-END VERIFICATION OF ETHEREUM SMART CONTRACTS

Smart contract failures have caused millions of dollars of lost funds [8],and rigorous formal methods are required to ensure the correctness and security of contract implementations [9, 10].The smart contract is usually written in a high-level language such as Solidity [47] or Vyper [48], and then it is compiled down to the Ethereum Virtual Machine (EVM) bytecode [46] that actually runs on the blockchain.

We present a formal verification tool for the EVM bytecode. We chose the EVM bytecode as the verification target language so that we can directly verify what is actually executed without the need to trust the correctness of the compiler. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted KEVM [11], a *complete* formal semantics of the EVM, and instantiated the K-framework's reachability logic theorem prover [12] to generate a *correct-by-construction* deductive program verifier for the EVM. While it is sound, the initial out-of-box EVM verifier was relatively slow and failed to prove many correct programs. We further optimized the verifier by introducing custom abstractions and lemmas specific to EVM that expedite proof searching in the underlying theorem prover. Our EVM verifier has been used to verify the *full functional correctness* of high-profile smart contracts including various ERC20 token [13], Ethereum's Casper FFG [14], DappHub's MakerDAO [15], and Gnosis Safe [16] contracts. Our verification tool and artifact is publicly available at [55].

Our methodology for formal verification of smart contracts is as follows. First, we formalize the high-level business logic of the smart contracts, based on a typically informal specification provided by the contract developers, to provide us with a precise and comprehensive specification of the functional correctness properties of the smart contracts. This high-level specification needs to be confirmed by the developer, possibly after several rounds of discussions and changes, to ensure that it correctly captures the intended behavior of their contracts. Then we refine the specification all the way down to the Ethereum Virtual Machine (EVM) level, often in multiple steps, to capture the EVM-specific details. The role of the final EVM-level specification is to ensure that nothing unexpected happens at the bytecode level, that is, that only what was specified in the high-level specification will happen when the bytecode is executed. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted KEVM [11], a complete formal semantics of the EVM, and instantiated the language-independent deductive program verifier [12] to generate a correct-by-construction deductive program verifier for the EVM. We use the verifier to verify the compiled EVM bytecode of the smart contract against its EVM-level specification. Note that the compiler of a high-level contract language (such as Solidity or Vyper) is not part of our trust base, since we directly verify the compiled EVM bytecode. Therefore, our verification results do not depend on the correctness of the compilers.

4.2.1 EVM Verification Challenges

Verifying the EVM bytecode is challenging, especially due to the internal byte manipulation operations that require non-linear integer arithmetic reasoning, which is undecidable in general [56]. Here we provide a few examples of the challenges.

Byte-Manipulation Operations The EVM provides three types of storage structures: a local memory, a local stack, and the global storage. Of these, only the local memory is byte-addressable (i.e., represented as an array of bytes), while the others are word-addressable (i.e., each represented as an array of 32-byte words). Thus, a 32-byte (i.e., 256-bit) word needs to be split into 32 chunks of bytes to be stored in the local memory, and those 32 chunks need to be merged back to be loaded in either the local stack or the global storage. These byte-wise splitting and merging operations can be formalized using

non-linear integer arithmetic operations, as follows.[1] Suppose x is a 256-bit integer. Let x_n be the n^{th} byte of x in its two's complement representation, where the index 0 refers to the least significant bit (LSB), defined as follows:

$$x_n \stackrel{\texttt{def}}{=} (x/256^n) \bmod 256 \tag{4.1}$$

Let *merge* be a function that takes as input a list of bytes and returns the corresponding integer value under the two's complement interpretation, recursively defined as:

$$\begin{aligned} \mathit{merge}(x_i \cdots x_{j+1} x_j) &\stackrel{\texttt{def}}{=} \mathit{merge}(x_i \cdots x_{j+1}) \mathbin{\underline{*}} 256 \mathbin{\underline{+}} x_j \quad \text{when } i > j \\ \mathit{merge}(x_i) &\stackrel{\texttt{def}}{=} x_i \end{aligned} \tag{4.2}$$

where $\underline{*}$ and $\underline{+}$ are multiplication and addition over words (modulo 2^{256}). If the byte-wise operations are blindly encoded as SMT theorems, then Z3, a state-of-the-art SMT solver, times out attempting to prove "$x = \mathit{merge}(x_{31} \cdots x_0)$". The SMT query can be simplified to allow Z3 to efficiently terminate, for example, by omitting the modulo reduction for multiplication and addition in *merge* with additional reasoning about the soundness of the omission. Despite these improvements, the merge operation still incurs severe performance penalties as solving the large formula is required for every load/store into memory, an extremely common operation.

Arithmetic Overflow Since EVM arithmetic instructions perform modular arithmetic (i.e., $+$, $-$, $*$, / mod 2^{256}), detecting arithmetic overflow is critical for preventing potential security holes due to an unexpected overflow. Otherwise, for example, increasing a user's token balance could trigger an overflow, resulting in loss of the funds as the balance wraps around to a lower-than-expected value. There is no standard EVM-level overflow check, so the overflow detection varies across compilers and libraries. For example, the Vyper compiler inserts the following runtime check for an addition `a + b` over the 256-bit unsigned integers `a` and `b`, `b == 0 || a` $\underline{+}$ `b > a`, where $\underline{+}$ represents addition modulo 2^{256}. It seems straightforward to show that the above formula is equivalent to `a` $+$ `b` $< 2^{256}$ (where $+$ is the pure addition *without* modulo reduction), but it is no longer trivial once the above is compiled down to EVM. The compiled EVM bytecode of the above conditional expression can be encoded in the SMT-LIB format as follows, `(not (= (chop (+ (bool2int (= b 0)) (bool2int (> (chop (+ a b)) a)))) 0))`, where `(chop x)` denotes

[1] It is also possible to formalize the byte-manipulation using the bit-vector theory, but the formalization using the mathematical integer theory has an advantage of the functional specifications being succinct. Indeed, the KEVM semantics adopted the integer formalization because of the advantage.

(x mod 2^{256}), and `(bool2int x)` is defined by `(ite x 1 0)`. However, Z3 fails (timeout) to prove that the above SMT formula is equivalent to `a` + `b` $< 2^{256}$.

Hash Collision Precise reasoning about the SHA3 hash[2] is critical. Since it is not practical to consider the hash algorithm details every time the hash function is called in the EVM bytecode, an abstraction for the hash function is required. Designing a sound but efficient abstraction is not trivial because while the SHA3 hash is not cryptographically collision-free, the contract developers assume collisions will not occur during normal execution of their contracts.[3] A naive way of capturing the assumption would be to simply abstract the SHA3 hash as an injective function. However, it is not sound simply because of the pigeonhole principle, and thus we need to be careful when abstracting the hash function.

4.2.2 EVM-Specific Abstractions

K's reachability logic theorem prover can be seen as a symbolic model checker equipped with coinductive reasoning about loops and recursions (refer to [12] for details of the underlying theory and implementation). The prover, in its current form, often delegates domain reasoning to SMT solvers. The performance of the underlying SMT solvers is critical for the overall performance. The domain reasoning involved in the EVM bytecode verification is not tractable in many cases, especially due to non-linear integer arithmetic. We had to design custom abstractions and lemmas to avoid the non-tractable domain reasoning and improve the scalability.

Abstraction for Local Memory We present an abstraction for the EVM local memory to allow word-level reasoning. As mentioned in Section 4.2.1, since the local memory is byte-addressable, the load and store operations involve the conversion between a word and a list of bytes, which is not tractable to reason about in general. Our abstraction helps to make the reasoning easier by abstracting away the byte-manipulation details of the conversion. Specifically, we introduce uninterpreted function abstractions and lemmas for word-level reasoning as follows.

The term $\mathsf{nthByteOf}(v, i, n)$ represents the i^{th} byte of the two's complement representation of v in n bytes (0 being the most significant bit), with discarding high-order bytes

[2] `https://keccak.team/index.html`

[3] The assumption is not unreasonable, as virtually all blockchains rely heavily on the collision-resistance of hash functions.

when v does not fit in n bytes. Precisely, it is defined as follows:

$$\mathsf{nthByteOf}(v, i, n) = \mathsf{nthByteOf}(\lfloor v/256 \rfloor, i, n-1) \qquad \text{when } n > i+1 \tag{4.3}$$

$$\mathsf{nthByteOf}(v, i, n) = v \mod 256 \qquad \text{when } n = i+1 \tag{4.4}$$

However, we want to keep it uninterpreted (i.e., do not unfold the definition) when the arguments are symbolic, to avoid the expensive non-linear arithmetic reasoning.

We introduce lemmas over the uninterpreted functional terms. The following lemmas are used for symbolic reasoning about MLOAD and MSTORE instructions. They capture the essential mechanisms used by the two instructions: splitting a word into a list of bytes and merging it back into the word. First, we have the bound of $\mathsf{nthByteOf}(v, i, n)$ by definition: $0 \leq \mathsf{nthByteOf}(v, i, n) < 256$. Then we have the following lemma for the merging operation:

$$\mathit{merge}(\mathsf{nthByteOf}(v, 0, n) \cdots \mathsf{nthByteOf}(v, n-1, n)) = v \qquad \text{if } 0 \leq v < 2^{8n} \text{ and } 1 \leq n \leq 32 \tag{4.5}$$

Refer to [55] for the other lemmas of the memory abstraction.

Abstraction for Hash We do not model the hash function as an injective function simply because it is not true due to the pigeonhole principle. Instead, we abstract it as an uninterpreted function, hash, that takes as input a list of bytes and returns an (unsigned) integer: $\mathsf{hash} : \{0, \cdots, 255\}^* \to \mathbb{N}$. Note that this abstraction allows the possibility of hash collision.

However, one can avoid reasoning about the potential collision by assuming all the hashed values appearing in each execution trace are collision-free. This can be achieved by instantiating the injectivity property only for the terms appearing in the symbolic execution, in a way analogous to universal quantifier instantiation.

Arithmetic Simplification Rules We introduce simplification rules, specific to EVM, that capture arithmetic properties, which reduce a given term into a smaller one. These rules help to improve the performance of the underlying theorem prover's symbolic reasoning. For example, we have the following simplification rule:

$$(x \times y) \;/\; y = x \quad \text{if } y \neq 0 \tag{4.6}$$

where / is the integer division.[4] We also have a rule for the masking operation, 0xff$\cdots$f & n, as follows, m & $n = n$ if $m + 1 = 2^{1+\log m}$ and $0 \leq n \leq m$, where & is the bitwise AND operator, and m denotes a bitmask 0xff$\cdots$f. Refer to [55] for other simplification rules.

4.3 CASE STUDY: ERC20 TOKEN CONTRACT VERIFICATION

We present a case study of completely verifying high-profile, practically deployed implementations of the ERC20 token contract [13], one of the most popular Ethereum smart contracts that provides the essential functionality of maintaining and exchanging tokens.

4.3.1 Formal Specification

The ERC20 standard [13] informally specifies the correctness properties that ERC20 token contracts must satisfy. Unfortunately, however, it leaves several corner cases unspecified, which makes it less than ideal to use in the formal verification of token contracts.

We specified ERC20-K [49], a complete formalization of the high-level business logic of the ERC20 standard, in the K framework. ERC20-K clarifies what data (e.g., balances and allowances) are handled by the various ERC20 functions and the precise meaning of those functions on such data. ERC20-K also clarifies the meaning of all the corner cases that the ERC20 standard omits to discuss, such as transfers to itself or transfers that result in arithmetic overflows, following the most natural implementations that aim at minimizing gas consumption. The complete specifications are available at [49].

Figure 4.6, for example, shows part of the (simplified) specification of `transfer`. It specifies two possible behaviors: success and failure.[5] For each case, it specifies the function parameters (`callData`), the return value (`output`), whether an exception occurred (`statusCode`), the log generated (`log`), the storage update (`storage`), and the path-condition (`requires`). Specifically, the success case (denoted by `[transfer-success]`) specifies that the function succeeds in transferring the `VALUE` tokens from the `FROM` account to the `TO`

[4]Note that Z3 fails to prove this seemingly trivial formula at the time of this writing. Indeed, this issue has been reported and fixed in the develop branch: https://github.com/Z3Prover/z3/issues/1683

[5]`transfer` admits four possible behaviors: success and failure of regular transfer (i.e., `FROM` $\neq$ `TO`), and success and failure of self-transfer (i.e., `FROM` = `TO`). Here we omit the self-transfer behaviors due to space limit. Refer to [49] for the complete specification.

```
[transfer-success]
callData: #abiCallData("transfer", #address(TO), #uint256(VALUE))
statusCode: _ => EVMC_SUCCESS
output: _ => #asByteStackInWidth(1, 32)
log: ... (. => #abiEventLog(FROM, "Transfer",
    #indexed(#address(FROM)), #indexed(#address(TO)), #uint256(VALUE))))
storage:
  #hashedLocation({BALANCES}, FROM) |-> (BAL_FROM => BAL_FROM -Int VALUE)
  #hashedLocation({BALANCES}, TO) |-> (BAL_TO => BAL_TO +Int VALUE)
  ...
requires:
  andBool FROM =/=Int TO
  andBool VALUE <=Int BAL_FROM
  andBool BAL_TO +Int VALUE <Int (2 ^Int 256)

[transfer-failure]
callData: #abiCallData("transfer", #address(TO), #uint256(VALUE))
statusCode: _ => EVMC_REVERT
output: _ => _
log: ...
storage:
  #hashedLocation({BALANCES}, FROM) |-> BAL_FROM
  #hashedLocation({BALANCES}, TO)   |-> BAL_TO
  ...
requires:
  andBool FROM =/=Int TO
  andBool ( VALUE >Int BAL_FROM
   orBool   BAL_TO +Int VALUE >=Int (2 ^Int 256) )
```

Figure 4.6: Formal specification of `transfer` function

account, with generating the corresponding log message, and returns 1 (i.e., true), if no overflow occurs (i.e., the `FROM` account has a sufficient balance, and the `TO` account has enough room to receive the tokens). The failure case (`[transfer-failure]`) specifies that the function throws an exception without modifying the account balances, if an overflow occurs.

4.3.2 Verification Result

For this case study, we consider three ERC20 token implementations: the Vyper ERC20 token[6], the HackerGold (HKG) ERC20 token[7], and OpenZeppelin's ERC20 token[8]. Of these, the Vyper ERC20 token is written in Vyper, and the others are written in Solidity.

[6] `https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20_solidity_compatible/ERC20.vy`

[7] `https://github.com/ether-camp/virtual-accelerator/blob/master/contracts/StandardToken.sol`

[8] `https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol`

We compiled the source code down to the EVM bytecode using each language compiler, and executed our verifier to verify that the compiled EVM bytecode satisfies the aforementioned specification. During this verification process, we found divergent behaviors across these contracts that do not conform to the ERC20 standard. Due to the deviation from the standard, we could not verify those contracts against the original ERC20-K specification. In order to show that they are "correct" w.r.t. the original specification *modulo* the deviation, we modified the specification to capture the deviation and successfully verified them against the modified specification. Table 4.3 provides the performance of the verifier. Below we describe the results.

Vyper ERC20 Token The Vyper ERC20 token is successfully verified against the original specification, implying its full conformance to the ERC20 standard.

HackerGold (HKG) ERC20 Token In addition to the well-known security vulnerability of the HKG token,[9] we found that the HKG token implementation deviates from our specification as follows:

- *No totalSupply function*: No `totalSupply` function is provided in the HKG token, which is *not* compliant to the ERC20 standard.
- *Returning false in failure*: It returns false instead of throwing an exception in the failure cases for both `transfer` and `transferFrom`. It does not violate the standard, as throwing an exception is recommended but not mandatory according to the ERC20 standard.
- *Rejecting transfers of 0 values*: It does not allow transferring 0 values, returning false immediately without logging any event. It is *not* compliant to the standard. This is a potential security vulnerability for any API clients assuming the ERC20-compliant behavior.
- *No overflow protection*: It does not check arithmetic overflow, resulting in the receiver's balance wrapping around the 256-bit unsigned integer's maximum value in case of the overflow. It does not violate the standard, as the standard does not specify any requirement regarding it. However, it is potentially vulnerable, since it will result in loss of the funds in case of the overflow as the receiver's balance wraps around to a lower-than-expected value.

[9] `https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued` Note that the token contract had been manually audited by Zeppelin, but they failed to find the vulnerability, which implies the need of the rigorous formal verification.

	Vyper	HKG	Zeppln.		Vyper	HKG	Zeppln.
`totalSupply`	36.4	N/A	34.3	`approve`	33.9	48.4	35.4
`balanceOf`	33.3	37.3	37.1	`transfer`	148.5	198.5	219.7
`allowance`	36.4	42.3	39.6	`transferFrom`	174.4	257.6	179.2

Table 4.3: Verification time (secs) of ERC20 token contracts

OpenZeppelin ERC20 Token The OpenZeppelin ERC20 token is a high-profile ERC20 token library developed by the security audit consulting firm Zeppelin[10]. We found that the OpenZeppelin token deviates from the ERC20-K specification as follows:

- *Rejecting transfers to address 0*: It does not allow transferring to address 0, throwing an exception immediately. It does not violate the standard, as the standard does not specify any requirement regarding it. However, it is questionable since while there are many other invalid addresses to which a transfer should not be made, it is not clear how useful rejecting a single invalid address is, at the cost of the additional gas consumption for every transfer transaction.

4.4 CASE STUDY: BIHU SMART CONTRACT VERIFICATION

We present another case study of the end-to-end verification of the Bihu KEY token operation contract, a commercially deployed smart contract on Ethereum mainnet.[11] Bihu[12] is a blockchain-based ID system, and KEY is the utility token for the Bihu ID system and community. Our formal verification artifact for the Bihu KEY contracts is publicly available at [57].

4.4.1 Formal Specification and Verification

The target of our formal verification is the `collectToken` function of the `KeyRewardPool` contract. We faithfully formalized the behavior of the function consulting the informal specification[13] provided by the Bihu team.

We adopted the same verification methodology, where we formalized the high-level business logic of the target smart contract, refined it down to the EVM level, and verified

[10] `https://zeppelin.solutions/security-audits`

[11] `https://etherscan.io/address/0x4cd988afbad37289baaf53c13e98e2bd46aaea8c\#code`

[12] `https://bihu.com/`

[13] `https://docs.google.com/document/d/1-PilHhInQxGod7FZNbtfv2bbgV1045ROT5TO3WLhDOE`

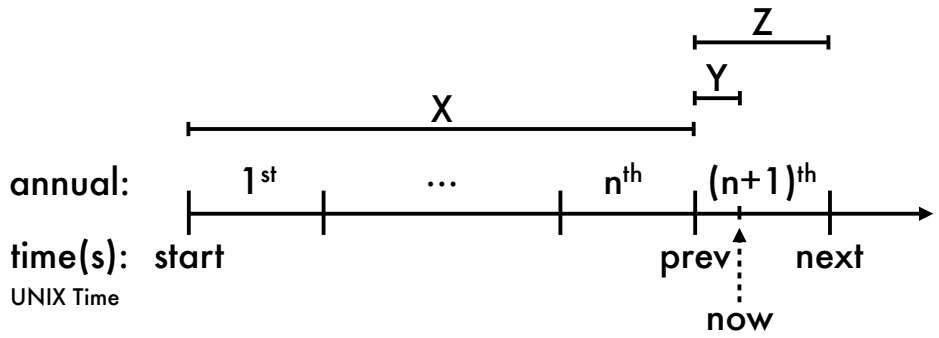


Figure 4.7: Illustration of important variables of $\mathsf{collectToken}_{\mathsf{spec}}$

the contract bytecode against the EVM-level specification. A notable difference from that of ERC20 token contract verification, however, is that here the refinement was conducted in multiple steps. Indeed, we defined the following four (refined) specifications, where each subsequent specification refines the previous one:

1. $\mathsf{collectToken}_{\mathsf{spec}}$: high-level definitional specification
2. $\mathsf{collectToken}_{\mathsf{code}}$: high-level constructive specification
3. $\texttt{collectToken}_{\texttt{solidity}}$: Solidity-level functional specification
4. $\texttt{collectToken}_{\texttt{evm}}$: EVM-level functional specification

Here, $\mathsf{collectToken}_{\mathsf{spec}}$ is the high-level specification written with the purpose of communication with the client to ensure that it correctly captures the intended behavior of their contract. While $\mathsf{collectToken}_{\mathsf{spec}}$ is rather a mathematical definition, $\mathsf{collectToken}_{\mathsf{code}}$ refines it to make the computation steps explicit, being more constructive. Since $\mathsf{collectToken}_{\mathsf{code}}$ employs simply the real arithmetic for the computation steps, $\texttt{collectToken}_{\texttt{solidity}}$ refines it to capture the unsigned integer arithmetic of Solidity (and thus EVM) including rounding errors of integer division. Finally, $\texttt{collectToken}_{\texttt{evm}}$ refines $\texttt{collectToken}_{\texttt{solidity}}$ further down to the EVM level to capture EVM-specific details.

Specification refinement is critical for this verification effort because of the inherent gap between the (high-level) code written by developers and the (low-level) code that actually runs on the blockchain/EVM. Moreover, we split the refinement process into multiple small steps, which makes it easier to prove soundness of each refinement step.

High-Level Definitional Specification $\mathsf{collectToken}_{\mathsf{spec}}$ is a formal high-level specification of the `collectTokens` function, the main function of the `KeyRewardPool` contract. We

derive it from the following informal English specification provided to us by the developer[14].

> "KeyRewardPool contract is responsible for releasing key tokens in the reward pool (initially about 45 billion tokens). The reward pool has a start time. From the start time, every 365 days, we define it as an annual. In each year, a total of 10% of the remaining amount is released. In each year, the token is released linearly."

Figure 4.7 illustrates the important variables of $\mathsf{collectToken}_{\mathsf{spec}}$. We formalize each English sentence, starting from the second one.

> "The reward pool has a start time."

Let start denote the start time as shown in Figure 4.7.

> "From the start time, every 365 days, we define it as an annual."

Suppose n full-years (i.e., *annual*s) have passed since start, where the time periods of each annual are denoted by 1^{st}, $\cdots$, n^{th}, and $(n+1)^{\text{th}}$, respectively, in Figure 4.7. Let now be the time (as seconds since Unix epoch) when the `collectToken` function is called, and suppose now is in the middle of the $(n+1)^{\text{th}}$ annual. Let prev (and next) be the time of the beginning (and the end, respectively) of the current $(n+1)^{\text{th}}$ annual.

> "In each year, a total of 10% of the remaining amount is released. In each year, the token is released linearly."

Let T be the total number of tokens. If T is fixed over the lifetime of the contract, the number of tokens to be released each annual can be formalized as follows. After the first annual has passed, $0.1 \times T$ tokens are released and eligible to be collected; after the second annual, additional $0.1 \times (0.9 \times T)$ tokens are released; after the third annual, yet another additional $0.1 \times (0.9^2 \times T)$ tokens are released. In general, after each n^{th} annual has passed, $0.1 \times (0.9^{n-1} \times T)$ tokens are newly released. Thus, the sum of all of the released tokens up to the end of the n^{th} annual is:

$$0.1 \cdot T + 0.1 \cdot (0.9 \cdot T) + 0.1 \cdot (0.9^2 \cdot T) + \cdots + 0.1 \cdot (0.9^{n-1} \cdot T) \;=\; T(1 - 0.9^n) \qquad (4.7)$$

Let X, Y, and Z be the number of tokens that are supposed to be released for each time period (more precisely, right after the end of the period), respectively, as shown in

[14] `https://docs.google.com/document/d/1-PilHhInQxGod7FZNbtfv2bbgV1045ROT5TO3WLhDOE`

Figure 4.7. Let b be the fraction of the current annual up to now, defined as:

$$b = \frac{\mathsf{now} - \mathsf{prev}}{\mathsf{next} - \mathsf{prev}} \tag{4.8}$$

Now we have the following equalities:

$$Y = Z \times b \tag{4.9}$$

$$Z = T \times 0.9^n \times 0.1 \tag{4.10}$$

$$X = T(1 - 0.9^n) \tag{4.11}$$

$$X + Z = T(1 - 0.9^{n+1}) \tag{4.12}$$

When the `collectToken` function is called, it collects all of the tokens that have been released (but not yet collected) up to that point. Let C be the number of tokens that have already been collected until now since start. The `collectToken` function, when being called at now, collects newly the following amount of tokens:

$$X + Y - C \tag{4.13}$$

Let C' be the number of the collected tokens after this `collectToken` call. Then we have $C' = X + Y$.

Note that T may change in between calls to the `collectToken` function. We have two cases:

- Case 1. If T has *increased* since the last `collectToken` call, the number of the newly collected tokens (more precisely $X + Y$) will *increase* as it will collect the new percentage of tokens that have not been collected in the past.
- Case 2. If T has *decreased* since the last `collectToken` call, the number of the newly collected tokens (more precisely $X + Y$) will *decrease*, meaning that more tokens have already been collected, as a percentage, than previously allowed. In this case, however, if $X + Y < C$, do not collect any token, meaning that $C' = C$.

Also, note that n and b can be derived directly from start and now, as follows:

$$n = \left\lfloor \frac{\mathsf{now} - \mathsf{start}}{31536000} \right\rfloor \tag{4.14}$$

$$b = \frac{\mathsf{now} - \mathsf{prev}}{31536000} = \frac{\mathsf{now} - \mathsf{start} \bmod 31536000}{31536000} \tag{4.15}$$

```
/* @input:   balance                                                    // T − C
 *           collectedTokens                                            // C
 *           rewardStartTime                                            // start
 *           now                                                        // now
 *
 * @output: balance'                                                   // T − C'
 *           collectedTokens'                                           // C'
 *
 * @pre−condition: now > rewardStartTime
 */
procedure collectToken() {
  total := collectedTokens + balance                                    // T
  yearCount := floor(days(now − rewardStartTime) / 365)                 // n
  fractionOfThisYear := (days(now − rewardStartTime) % 365) / 365       // b

  remainingTokens := total * (0.9 ^ yearCount)
  totalRewardThisYear := remainingTokens * 0.1                          // Z
  canExtractThisYear := totalRewardThisYear * fractionOfThisYear        // Y
  canExtract := canExtractThisYear + (total − remainingTokens)
                                   − collectedTokens                    // Y + X − C

  collectedTokens' := collectedTokens + canExtract                      // C'
  balance' := balance − canExtract                                      // T − C'
}
```

Figure 4.8: $\mathsf{collectToken}_{\mathsf{code}}$: All of the arithmetic operations are the purely mathematical ones, with no overflow nor rounding errors.

where 31,536,000 is the number of seconds in a year ($= 365 \times 24 \times 3600$).

High-Level Constructive Specification We specify $\mathsf{collectToken}_{\mathsf{code}}$, a constructive definition of $\mathsf{collectToken}_{\mathsf{spec}}$, in a form of pseudo-code, as shown in Figure 4.8. The arithmetic operations used in $\mathsf{collectToken}_{\mathsf{code}}$ are the purely mathematical ones, with no overflow nor rounding errors.

Lemma 4.1 bridges the gap between $\mathsf{collectToken}_{\mathsf{code}}$ and $\mathsf{collectToken}_{\mathsf{spec}}$.

Lemma 4.1. *If the inputs of* `collectToken` *in Figure 4.8 satisfies the following equations:*

$$\mathit{balance} = T - C \tag{4.16}$$

$$\mathit{collectedTokens} = C \tag{4.17}$$

$$\mathit{rewardStartTime} = \mathit{start} \tag{4.18}$$

$$\mathit{now} = \mathit{now} \tag{4.19}$$

then the following holds for the outputs of the function:

$$\begin{array}{rcccl} \mathit{balance}' & = & T - C' & = & T - (X + Y) \\ \mathit{collectedTokens}' & = & C' & = & X + Y \end{array} \tag{4.20}$$

Proof. Immediate from the following equalities for the intermediate values, by definition of $\mathsf{collectToken}_{\mathsf{spec}}$ and $\mathsf{collectToken}_{\mathsf{code}}$.

$$\begin{array}{lcl} \mathsf{total} & = & T \\ \mathsf{yearCount} & = & n \\ \mathsf{fractionOfThisYear} & = & b \\ \mathsf{remainingTokens} & = & \mathsf{total} - X \\ \mathsf{totalRewardThisYear} & = & Z \\ \mathsf{canExtractThisYear} & = & Y \\ \mathsf{canExtract} & = & X + Y - C \end{array} \tag{4.21}$$

Solidity-Level Functional Specification Now we convert the high-level specification to $\texttt{collectToken}_{\texttt{solidity}}$ by replacing the pure mathematical operations with the unsigned integer arithmetic operations of Solidity, as shown in Figure 4.9. $\texttt{collectToken}_{\texttt{solidity}}$ serves as the functional correctness specification of `collectToken`. Note that we use different fonts to distinguish different specification variables: the sans serif font for the $\mathsf{collectToken}_{\mathsf{code}}$ variables, and the `teletype` font for those of $\texttt{collectToken}_{\texttt{solidity}}$.

Note that there is a gap between $\mathsf{collectToken}_{\mathsf{code}}$ and $\texttt{collectToken}_{\texttt{solidity}}$, especially due to the integer division rounding errors. We next analyze the rounding errors to bridge the gap between them.

First, let us define the integer division in terms of the mathematical one with the floor operation as follows:

$$x \mathbin{/} y \stackrel{\texttt{def}}{=} \left\lfloor \frac{x}{y} \right\rfloor \tag{4.22}$$

where / is the integer division. Since $r - 1 < \lfloor r \rfloor \leq r$ (for $r \in \mathbb{R}$), we have:

$$\frac{x}{y} - 1 < x \mathbin{/} y \leq \frac{x}{y} \tag{4.23}$$

Let us introduce the epsilon (ϵ) notation, a small non-deterministic number within the range $[0, 1)$, i.e., $0 \leq \epsilon < 1$, so that we can represent the above inequalities in a simpler

```
/* @input:  uint balance
 *          unit collectedTokens
 *          unit rewardStartTime
 *          unit now
 *
 * @output: unit balance'
 *          unit collectedTokens'
 *
 * @pre-condition: now > rewardStartTime
 */
procedure collectToken() {
  unit total := collectedTokens + balance
  unit yearCount := days(now - rewardStartTime) / 365
  unit fractionOfThisYear365 := days(now - rewardStartTime) % 365

  unit remainingTokens := power(total, 90, 100, yearCount)
  unit totalRewardThisYear := remainingTokens * 10 / 100
  unit canExtractThisYear := totalRewardThisYear
                               * fractionOfThisYear365 / 365
  unit canExtract := canExtractThisYear + (total - remainingTokens)
                                        - collectedTokens

  collectedTokens' := collectedTokens + canExtract
  balance' := balance - canExtract
}

// return (conceptually): acc * ((base_n / base_d) ^ exp)
function power(acc, base_n, base_d, exp) {
  if exp == 0 {
    return acc
  } else {
    return power(acc * base_n / base_d, base_n, base_d, exp - 1)
  }
}
```

Figure 4.9: $\texttt{collectToken}_{\texttt{solidity}}$: All of the arithmetic operations are the unsigned integer arithmetics, where an exception is thrown when an overflow occurs.

form as follows:

$$x \;/\; y = \left\lfloor \frac{x}{y} \right\rfloor = \frac{x}{y} - \epsilon \tag{4.24}$$

We start to analyze the error bound of `remainingTokens`, which is, `power(total,90,100,`n`)` for $n \geq 0$. First, we have:

$$\texttt{power(total,90,100,0)} = \texttt{total} \tag{4.25}$$

$$\begin{aligned}
\texttt{power(total,90,100,1)} &= \texttt{total} \times 90 \texttt{ / } 100 && (4.26)\\
&= \lfloor \texttt{total} \times 0.9 \rfloor && (4.27)\\
&= \texttt{total} \times 0.9 - \epsilon_1 && (4.28)
\end{aligned}$$

$$\begin{aligned}
\texttt{power(total,90,100,2)} &= (\texttt{total} \times 90 \texttt{ / } 100) \times 90 \texttt{ / } 100 && (4.29)\\
&= \lfloor \lfloor \texttt{total} \times 0.9 \rfloor \times 0.9 \rfloor && (4.30)\\
&= (\texttt{total} \times 0.9 - \epsilon_1) \times 0.9 - \epsilon_2 && (4.31)\\
&= \texttt{total} \times 0.9^2 - \epsilon_1 \times 0.9 - \epsilon_2 && (4.32)
\end{aligned}$$

$$\begin{aligned}
\texttt{power(total,90,100,3)} &= ((\texttt{total} \times 90 \texttt{ / } 100) \times 90 \texttt{ / } 100) \times 90 \texttt{ / } 100 && (4.33)\\
&= \lfloor \lfloor \lfloor \texttt{total} \times 0.9 \rfloor \times 0.9 \rfloor \times 0.9 \rfloor && (4.34)\\
&= (\texttt{total} \times 0.9^2 - \epsilon_1 \times 0.9 - \epsilon_2) \times 0.9 - \epsilon_3 && (4.35)\\
&= \texttt{total} \times 0.9^3 - \epsilon_1 \times 0.9^2 - \epsilon_2 \times 0.9 - \epsilon_3 && (4.36)
\end{aligned}$$

where `/` is the integer division. Thus, in general, for any $n \geq 0$, we have:

$$\texttt{power(total,90,100,}n\texttt{)} = \texttt{total} \times 0.9^n - \epsilon_1 \times 0.9^{n-1} - \cdots - \epsilon_{n-1} \times 0.9 - \epsilon_n \tag{4.37}$$

By the definition of ϵ, we have:

$$\begin{aligned}
\texttt{total} \times 0.9^n \geq \texttt{power(total,90,100,}n\texttt{)} &> \texttt{total} \times 0.9^n - 0.9^{n-1} - \cdots - 0.9 - 1 && (4.38)\\
&= \texttt{total} \times 0.9^n - \frac{1 - 0.9^n}{0.1} && (4.39)\\
&> \texttt{total} \times 0.9^n - 10 && (4.40)
\end{aligned}$$

Since `total` $= T$, we have

$$T \times 0.9^n - 10 < \texttt{remainingTokens} = \texttt{power(total,90,100,}n\texttt{)} \leq T \times 0.9^n \tag{4.41}$$

Next, we analyze the error bound of `totalRewardThisYear`, which is, `remainingTokens` $\times$ 10 / 100. By the definition of the integer division, we have:

$$\texttt{remainingTokens} \times 0.1 - 1 < \texttt{remainingTokens} \times 10 \texttt{ / } 100 \leq \texttt{remainingTokens} \times 0.1 \tag{4.42}$$

By the equation 4.41, we have:

$$\begin{aligned} \texttt{totalRewardThisYear} = \texttt{remainingTokens} \times 10 \texttt{ / } 100 &\leq \texttt{remainingTokens} \times 0.1 && (4.43)\\ &\leq T \times 0.9^n \times 0.1 && (4.44)\\ &= Z && (4.45) \end{aligned}$$

and

$$\begin{aligned} \texttt{totalRewardThisYear} = \texttt{remainingTokens} \times 10 \texttt{ / } 100 &> \texttt{remainingTokens} \times 0.1 - 1 && (4.46)\\ &> (T \times 0.9^n - 10) \times 0.1 - 1 && (4.47)\\ &= T \times 0.9^n \times 0.1 - 2 && (4.48)\\ &= Z - 2 && (4.49) \end{aligned}$$

That is,

$$Z - 2 < \texttt{totalRewardThisYear} \leq Z \tag{4.50}$$

Similarly, the error bound of `canExtractThisYear` is analyzed as follows. We have:

$$\texttt{totalRewardThisYear} \times b - 1 < \texttt{canExtractThisYear} \leq \texttt{totalRewardThisYear} \times b \tag{4.51}$$

By the equation 4.50, we have:

$$\texttt{canExtractThisYear} \leq \texttt{totalRewardThisYear} \times b \leq Z \times b = Y \tag{4.52}$$

and

$$\begin{aligned} \texttt{canExtractThisYear} &> \texttt{totalRewardThisYear} \times b - 1 && (4.53)\\ &> (Z - 2) \times b - 1 && (4.54)\\ &= Z \times b - 2b - 1 && (4.55)\\ &> Z \times b - 3 \qquad (\text{since } 0 \leq b < 1) && (4.56)\\ &= Y - 3 && (4.57) \end{aligned}$$

That is, we have:

$$Y - 3 < \texttt{canExtractThisYear} \leq Y \tag{4.58}$$

Finally, we analyze the error bound of `canExtract`. By the equations 4.41 and 4.58, we have:

$$(Y-3) - T \times 0.9^n < \texttt{canExtractThisYear} - \texttt{remainingTokens} < Y - (T \times 0.9^n - 10) \tag{4.59}$$

Since

$$\texttt{canExtract} = \texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens}) - \texttt{collectedTokens} \tag{4.60}$$

$$= \texttt{canExtractThisYear} + (T - \texttt{remainingTokens}) - C \tag{4.61}$$

we have:

$$(X + Y - C) - 3 < \texttt{canExtract} < (X + Y - C) + 10 \tag{4.62}$$

Thus, the number of collected tokens calculated by the `collectToken` function may be up to 10 more than, or 3 less than the mathematical definition, due to the integer division rounding errors. The accumulated rounding error is constant-bounded, thus stable, no matter how large n is. The following lemma formulates this fact.

Lemma 4.2. *If the inputs of* `collectToken` *in Figure 4.9 satisfies the following equations:*

$$\texttt{balance} = T - C \tag{4.63}$$

$$\texttt{collectedTokens} = C \tag{4.64}$$

$$\texttt{rewardStartTime} = \textit{start} \tag{4.65}$$

$$\texttt{now} = \textit{now} \tag{4.66}$$

then the following holds for the outputs of the function:

$$\begin{array}{rcccl} (T - C') - 10 & < & \texttt{balance'} & < & (T - C') + 3 \\ C' - 3 & < & \texttt{collectedTokens'} & < & C' + 10 \end{array} \tag{4.67}$$

Proof. By the equations 4.41, 4.50, 4.58, and 4.62.

Another important property of `collectToken` is the monotonicity. That is, the number of collected tokens increases as time goes on. In other words, the following should always

hold:

$$\texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens}) \geq \texttt{collectedTokens} \tag{4.68}$$

While it is clear that the above holds over the real arithmetic, it is not clear whether the above holds over the integer arithmetic (due to the rounding errors).

Lemma 4.3. *Suppose two* `collectToken` *function (as shown in Figure 4.9) calls are made at times t and t',respectively, where $t < t'$. Let r and r' be the output values of* `canExtractThisYear` $+$ $(\texttt{total} - \texttt{remainingTokens})$ *for each function call at t and t', respectively. Assume that* `total` *does* not *descrese between t and t'. Then, $r \leq r'$.*

Proof. We only need to show in the case `total` is fixed, from which it is trivial to show in the case `total` increases. Also, if t and t' are in the same annual, it is trivial to show, since `remainingTokens` is fixed and `canExtractThisYear` is monotone within the annual. Thus, we only need to show for the case t and t' are in the different annual. Specifically, it is sufficient to show when t is the last second of the $n+1^{\text{th}}$ annual, and t' is the first second of the $n+2^{\text{th}}$ annual, for arbitrary $n \geq 0$. Then, we have:

$$r = \texttt{power}(T,\texttt{90},\texttt{100},n) \times 10 \texttt{ / } 100 \times 31535999 \texttt{ / } 31536000 + (T - \texttt{power}(T,\texttt{90},\texttt{100},n)) \tag{4.69}$$

$$< \texttt{power}(T,\texttt{90},\texttt{100},n) \times 10 \texttt{ / } 100 + (T - \texttt{power}(T,\texttt{90},\texttt{100},n)) \tag{4.70}$$

$$= T - \texttt{power}(T,\texttt{90},\texttt{100},n) + \texttt{power}(T,\texttt{90},\texttt{100},n) \times 10 \texttt{ / } 100 \tag{4.71}$$

$$= T - \texttt{power}(T,\texttt{90},\texttt{100},n) + \texttt{power}(T,\texttt{90},\texttt{100},n) \times 0.1 - \epsilon \tag{4.72}$$

$$= T - \texttt{power}(T,\texttt{90},\texttt{100},n) \times 0.9 - \epsilon \tag{4.73}$$

$$r' = \texttt{power}(T,\texttt{90},\texttt{100},n) \times 10 \texttt{ / } 100 \times 0 \texttt{ / } 31536000 + (T - \texttt{power}(T,\texttt{90},\texttt{100},n+1)) \tag{4.74}$$

$$= T - \texttt{power}(T,\texttt{90},\texttt{100},n+1) \tag{4.75}$$

$$= T - \texttt{power}(T,\texttt{90},\texttt{100},n) \times 90 \texttt{ / } 100 \tag{4.76}$$

$$= T - (\texttt{power}(T,\texttt{90},\texttt{100},n) \times 0.9 - \epsilon) \tag{4.77}$$

$$= T - \texttt{power}(T,\texttt{90},\texttt{100},n) \times 0.9 + \epsilon \tag{4.78}$$

By the definition of ϵ, we conclude: $r \leq r'$.

Now we can show that `collectToken` is monotonic even over the integer arithmetic.

Corollary 4.1. *If* `balance` *does not decrease since the last* `collectToken` *function call, the following always hold:*

$$\texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens}) \geq \texttt{collectedTokens} \tag{4.79}$$

Proof. By Lemma 4.3, and the fact that `collectedTokens'` is set to (less than or) equal to $\texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens})$.

EVM-Level Functional Specification Now we specify $\texttt{collectToken}_{\texttt{evm}}$, a refinement of $\texttt{collectToken}_{\texttt{solidity}}$, that captures EVM-specific details. $\texttt{collectToken}_{\texttt{solidity}}$ is a Solidity-level specification, intentionally omitting EVM-specific details such as gas consumption, data layout in storage, ABI encoding, and byte representation of the program. However, reasoning about the low-level details is critical because many security vulnerabilities are related to the EVM quirks.

We refine $\texttt{collectToken}_{\texttt{solidity}}$ to EVM-level variant $\texttt{collectToken}_{\texttt{evm}}$, which captures all of the detailed behaviors that can happen when the code is compiled and executed at the EVM level. That includes laying out the contract state variables in the EVM storage, encoding the program and the call data in bytes, and specifying additional information such as gas consumption.

We verified a mathematically equivalent variant of the `collectToken` function, as shown in Figure 4.10. Due to time constraints (we only had two weeks to complete this verification project), we have *not* verified the external call to `key.balanceOf` and `key.transfer`. The differences are as follows:

- Instead of calling `time()` to get the current time, `now` and `rewardStartTime` are given as input to the `collectToken` function.
- We declare `balance` as a global variable rather than calling `key.balanceOf` function. In the end, we directly update `balance` instead of calling `key.transfer` function.
- We change:

  ```
  uint canExtract = canExtractThisYear + total - remainingTokens;
  ```

  to the following:

  ```
  uint canExtract = canExtractThisYear + (total - remainingTokens);
  ```

  in order to avoid potential unnecessary overflow.

```
pragma solidity ^0.4.18;

contract KeyRewardPool is DSMath {
  uint public collectedTokens;
  uint public balance;
  uint constant public yearlyRewardPercentage = 10;

  // @notice call this method to extract the tokens
  function collectToken(uint nowTime, uint rewardStartTime) public returns(bool){
    require(nowTime > rewardStartTime);

    uint total = add(collectedTokens, balance);
    uint remainingTokens = total;
    uint yearCount = yearFor(nowTime, rewardStartTime);

    for(uint i = 0; i < yearCount; i++) {
      remainingTokens = div(mul(remainingTokens, 100 - yearlyRewardPercentage), 100);
    }
    uint totalRewardThisYear = div(mul(remainingTokens, yearlyRewardPercentage), 100);

    // the reward will be increasing linearly in one year.
    uint canExtractThisYear =
        div(mul(totalRewardThisYear, (nowTime - rewardStartTime) % 365 days), 365 days);
    uint canExtract = canExtractThisYear + (total - remainingTokens);
    canExtract = sub(canExtract, collectedTokens);

    if (canExtract > balance) {
      canExtract = balance;
    }

    collectedTokens = add(collectedTokens, canExtract);
    balance = sub(balance, canExtract);
    return true;
  }

  function yearFor(uint nowTime, uint rewardStartTime) public constant returns(uint) {
    return nowTime < rewardStartTime
      ? 0
      : sub(nowTime, rewardStartTime) / (365 days);
  }
}
```

Figure 4.10: Modified `collectToken` source code

- We omit to log the `TokensWithdrawn` event.

```
[topLevel]
k: #execute => (RETURN RET_ADDR:Int 32 ~> _)
output: _
memoryUsed: 0 => _
callData: #abiCallData("collectToken", #uint256(NOW), #uint256(START))
wordStack: .WordStack => _
localMem: .Map => .Map[ RET_ADDR := #asByteStackInWidth(1, 32) ] _:Map
pc: 0 => _
gas: GASCAP => _
log: _
refund: _ => _
storage:
  #hashedLocation({COLLECTEDTOKENS}) |-> (COLLECTED => COLLECTED +Int VALUE)
  #hashedLocation({BALANCE}        ) |-> (BAL       => BAL       -Int VALUE)
  _:Map
requires:
  /\  0 <=Int COLLECTED                          <Int (2 ^Int 256)
  /\  0 <=Int BAL                                <Int (2 ^Int 256)
  /\  0 <=Int START                              <Int (2 ^Int 256)
  /\  0 <=Int (COLLECTED +Int BAL)
  /\          (COLLECTED +Int BAL) *Int 3153600 <Int (2 ^Int 256)
  /\  0  <Int (NOW -Int START)                   <Int (2 ^Int 256)
  /\  COLLECTED +Int 3
            <Int #accumulatedReleasedTokens(BAL, COLLECTED, START, NOW)
            <Int (BAL +Int COLLECTED) -Int 10
  /\  GASCAP >=Int (293 *Int ((NOW -Int START) /Int 31536000)) +Int 43000
ensures:
  VALUE ==Int @canExtractThisYear(COLLECTED +Int BAL, NOW, START)
              +Int BAL
              -Int @remainingTokens(COLLECTED +Int BAL, NOW, START)
```

Figure 4.11: $\texttt{collectToken}_{\texttt{evm}}$: top-level specification

In order to verify the `collectToken` function, we present the top-level spec, together with the spec for the loop invariant. Specifically, we provide the specification template parameters from which the full specifications are derived by instantiating the template [58] and focus on explaining the EVM-specific detailed behaviors. For more details of the specification template and template parameters, refer to the eDSL [58].

Top-Level Specification Figure 4.11 shows the functional specification at the EVM level, describing the pre- and post-conditions of the `collectToken` function. Below we explain the template parameters of the specification.

- `k` specifies that the execution eventually reaches the `RETURN` instruction, meaning that the program will successfully terminate.
- `memoryUsed` specifies that the initial memory consumption is initially 0. During the execution, `collectTokens` will write values into the memory. However, the exact amount of used memory by the end of the execution is irrelevant.
- `callData` specifies the call data using the `#abiCallData` eDSL notation [58].
- `wordStack` specifies that the local stack is initially empty. By the end of the execution, the stack may not be empty, but that is not relevant.
- `localMem` specifies that the local memory is empty in the beginning, but in the end, it will store the return value `true`, represented as `1`.
- `pc` specifies the program counter starting from 0.
- `gas` specifies the maximum gas amount, `GASCAP`, ensuring that the program does not consume more gas than the limit. Here we give a loose upper bound which is specified in `requires`.
- `log` specifies that no log is generated during the execution.
- `refund` specifies that a refund may be issued. Note that, however, we have not specified the refund detail since it is not essential for the functional correctness.
- `storage` specifies that the value of `collectedTokens` is `COLLECTED` and the value of `balance` is `BAL`. Other entries are not relevant (and could be arbitrary values). The `{COLLECTEDTOKENS}` and `{BALANCE}` terms are the eDSL notations (called program-specific template parameters) that represent the position index of the corresponding variables.
- `requires` specifies the pre-conditions of the function.
  - The first three lines specify the range of symbolic values based on their types.
  - Lines 4–5 specify the range of the total number of tokens. It should be sufficiently small to avoid multiplication overflow. Note that 3153600 is the seconds in an year divided by 10, i.e., $\frac{365\times24\times3600}{10}$.
  - Line 6 specifies the total seconds that have passed since `rewardStartTime` should be in the proper range of 256-bit unsigned integers.

  - Lines 7–9 specify that `#accumulatedReleasedTokens(···)`[15], the accumulated number of the released token until `now` since `rewardStartTime`, should be marginally greater than `collectedTokens` and smaller than the total number of tokens, considering the (bounded) rounding errors due to the integer division in $\mathsf{collectToken}_{\mathsf{code}}$.
  - Line 10 specifies the loose upper bound on the gas cost, which depends on the input.

- `ensures` specifies that the number of newly collected token, `canExtract`, is correct, i.e., it is the same with that of $\mathtt{collectToken}_{\mathtt{solidity}}$. We use two macros `@canExtractThisYear` and `@remainingTokens` to succinctly specify that, defined as follows[16]:

```
rule @canExtractThisYear(TOTAL, NOW, START)
  => ((#roundpower(TOTAL, 90, 100, (NOW -Int START) /Int 31536000)
        *Int 10 /Int 100)
        *Int ((NOW -Int START) %Int 31536000)) /Int 31536000  [macro]

rule @remainingTokens(TOTAL, NOW, START)
  => #roundpower(TOTAL, 90, 100, (NOW -Int START) /Int 31536000)  [macro]
```

Loop Invariant Specification Figure 4.12 shows the loop invariant of the `for` loop inside the `collectTokens` function. Below we explain the noteworthy parameters.

- `memoryUsed` specifies that the local memory is never used (no read/write) inside the loop. Indeed, the local variables are stored in the stack instead of the memory in this EVM bytecode.

- `wordStack` specifies all of the elements in the local stack before and after the execution of the loop. In the end, the loop index becomes `YEARCOUNT`, and the `remainingTokens`'s value becomes `#roundpower(REMAINING, 90, 100, YEARCOUNT -Int INDEX)`, which is the most important loop invariant.

- `pc` specifies the program counters for the loop head and the end of the loop.

- `gas` specifies the gas consumption which depends on the number of loop iterations.

[15] `#accumulatedReleasedTokens` corresponds to $X + Y$ in $\mathsf{collectToken}_{\mathsf{spec}}$

[16] `#roundpower` is a macro corresponding to the `power` in the $\mathtt{collectToken}_{\mathtt{solidity}}$.

```
[loopInvariant]
k: #execute => #execute
output: _
memoryUsed: MU
callData: _
wordStack:
  CANEXTRACT : CANEXTRACTTHISYEAR : TOTALREWARDTHISYEAR
      : INDEX : YEARCOUNT
      : REMAINING
      : TOTAL : RETURNVAL : START : NOW : RPC : FUNID : .WordStack
  =>
  CANEXTRACT : CANEXTRACTTHISYEAR : TOTALREWARDTHISYEAR
      : YEARCOUNT : YEARCOUNT
      : #roundpower(REMAINING, 90, 100, YEARCOUNT -Int INDEX)
      : TOTAL : RETURNVAL : START : NOW : RPC : FUNID : .WordStack
localMem: _
pc: 498 => 545
gas: GASCAP => GASCAP -Int (293 *Int (YEARCOUNT -Int INDEX)) -Int 26
log: _
refund: _
storage: _
requires:
  /\  0 <=Int MU         /\  MU                 <Int (2 ^Int 256)
  /\  0 <=Int YEARCOUNT  /\  YEARCOUNT          <Int (2 ^Int 256)
  /\  0 <=Int REMAINING  /\  REMAINING *Int 90  <Int (2 ^Int 256)
  /\  0 <=Int INDEX      /\  INDEX             <=Int YEARCOUNT
  /\  GASCAP >=Int ((293 *Int (YEARCOUNT -Int INDEX)) +Int 26)
```

Figure 4.12: $\texttt{collectToken}_{\texttt{evm}}$: loop invariant specification

- `requires` specifies the proper range of the symbolic values. Especially, `REMAINING` should be sufficiently small to avoid multiplication overflow.

We mechanize Corollary 4.1, which is trusted by the verifier, as follows:

```
rule @canExtractThisYear(COLLECTED +Int BAL, NOW, START)
     +Int ((COLLECTED +Int BAL)
     -Int @remainingTokens(COLLECTED +Int BAL, NOW, START)) >=Int COLLECTED
  => true
  requires #shouldReleaseSofar(BAL, COLLECTED, START, NOW) >Int COLLECTED +Int 3
```

We took the modified source code, inlined the `DSMath` contract and compiled it to the EVM bytecode using Remix Solidity IDE (of the version `soljson-v0.4.20`). The verification

proof of the `collectToken` function can be reproduced by using the verifier. Refer to [57] for more details.

Threats to Validity The formal verification results presented in this thesis only show that the target contract behaviors meet the formal (functional) specifications. Moreover, the correctness of the generated formal proofs assumes the correctness of the specifications and their refinement, the correctness of KEVM, the correctness of the K-framework's reachability logic theorem prover, and the correctness of the Z3 SMT solver. The presented result makes no guarantee about properties not specified in the formal specification. Importantly, the presented formal specification considers only the behaviors *within* the EVM, *without* considering the block/transaction level properties or off-chain behaviors, meaning that the verification result does *not* completely rule out the possibility of the contract being vulnerable to existing and/or unknown attacks.

CHAPTER 5: COMPLETE FORMAL SEMANTICS OF JAVASCRIPT AND TOOLS

In this chapter, we present a complete formal semantics of JavaScript, evaluating the effectiveness of specifying operational language semantics, as well as using it for instantiating the universal formal methods. Much of the content in this chapter comes from Park *et al.* [7].

JavaScript is the most popular client-side programming language. Recently, JavaScript has started to be used in not only client-side, but also server-side programming [59], and even beyond web applications [60, 61]. Despite its popularity, JavaScript suffers from several language design inconsistencies [62], which can lead to security vulnerabilities. Nontransparent behaviors are good targets for attackers [63, 64]. To address the utmost importance of security in web applications, there have been several formal analysis studies proposed recently for JavaScript [65, 66, 67, 68, 69], but these address fragments of the language and are not fully validated with a complete, formal JavaScript semantics. Guha *et al.* [70] admit they cannot show their static analysis sound due to the absence of a complete formal semantics of JavaScript.

A formal semantics should serve as a solid foundation for JavaScript language development, so it must be correct and complete (to be trusted and useful), executable (to yield a reference implementation), and appropriate for program reasoning and verification.

Several efforts to give JavaScript a formal semantics have been made, most notably by Politz *et al.* [66] and Bodin *et al.* [71]. Unfortunately, no existing semantics comes close to having the desired properties mentioned above. First, as shown in Tables 5.1 and 5.2, they are incomplete and contain errors. Second, they require different formalizations for different purposes, e.g., an operational/computational semantics for execution and a axiomatic/declarative semantics for deductive reasoning. Having to define two or more different semantics for a real-life language, together with proofs of equivalence, is a huge burden in itself, not to mention that these all need to be maintained as the language evolves. Third, due to the functional nature of their interpreters, these semantics cannot handle the non-determinism of JavaScript well. Finally, their interpreters are not suited for symbolic execution, and thus for developing program reasoning tools.

For these reasons, we developed yet another JavaScript semantics in order to have a single, clean-slate semantics that can be used not only as a reference model for JavaScript, but also to develop formal analysis tools for it. We employed $\mathbb{K}$ [38] (`http://kframework.org`) as the formalism medium. In $\mathbb{K}$, a language semantics is described as a term rewriting system. At no additional cost, $\mathbb{K}$ provides an execution engine, which yields an

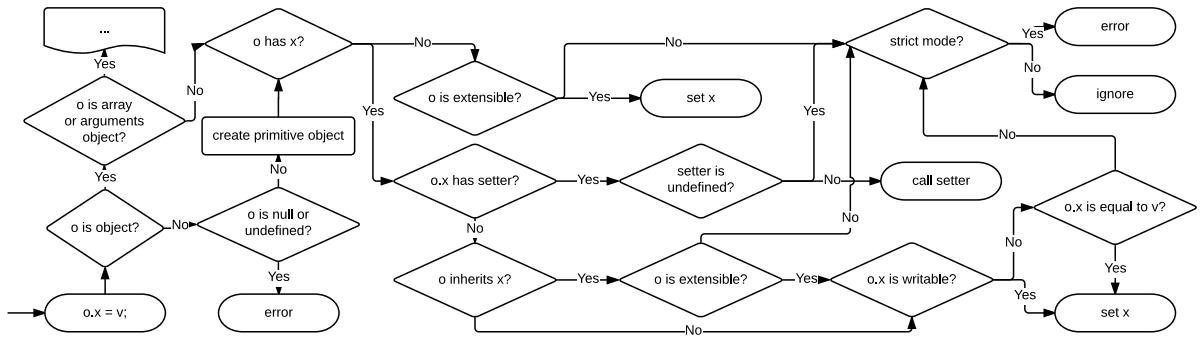


Figure 5.1: Semantics of Object Property Update: o.x = v;

interpreter for the defined language, as well as a sound and relatively complete deductive verification system based on symbolic execution, which can be used to reason about programs.

Challenges in Formalizing JavaScript JavaScript is an unusual language, full of tricky corner cases. Like HTML, JavaScript programs do not easily fail. Seemingly nonsensical programs work by design, i.e., they have properly defined semantics according to the language standard. Completely defining all of the corner cases is highly non-trivial, especially because the language standard, a 250-page English document, contains various ambiguities and unspecified behaviors (which have led to divergence between JavaScript implementations). To handle these difficulties, we decided to make our semantics executable, so that we can test our semantics incrementally. Incremental testing allowed us to eliminate ambiguities one by one and to enhance our understanding of JavaScript's corner cases.

JavaScript is complex. Beside typical difficulties of scripting languages such as dynamic (implicit) casting and the eval construct, the standard ECMAScript 5.1 introduced new features such as the strict mode and explicit getters/setters. The mixed use of the strict and non-strict modes, and of the data and accessor (getter/setter) properties, makes it inevitable to have complex case analyses in the semantics. For example, Figure 5.1 describes the "simple" object property update o.x = v semantics at a high-level, showing how many cases need to be distinguished: o is a normal object or not; o is extensible or not; x is inherited or not; x is writable or not; x is an accessor property or not; the code is strict or not. To keep better track of all such special cases, we chose to systematically, almost mechanically translate the language standard *as is* into formal semantics (as opposed to defining what we thought JavaScript ought to be doing).

JavaScript is non-deterministic. For example, the for-in construct iterates through all the enumerable properties of a given object non-deterministically. The enumeration order is unspecified, implementation-dependent, and may vary for different iterations of a for-in loop. Formalizing JavaScript's non-determinism in a semantics that has all the desirable properties mentioned earlier is non-trivial. A collection semantics defined as an inductive relation can capture non-determinism easily, but is unsuitable for execution, while a semantics defined as an eval function is suitable for execution but cannot capture non-determinism naturally. Rewriting Logic [39] is a sweet spot, as it can effectively define, execute, and reason about non-deterministic specifications.

Contribution We present KJS, the most complete and thoroughly tested formal semantics of JavaScript to date, specifically of ECMAScript 5.1, the latest language standard at the time of writing. It has been tested against the ECMAScript conformance test suite, and passed all 2,782 test programs for the core language. Table 5.1 shows that KJS is far more complete than any other semantics, and even more standards-compliant than production JavaScript engines such as Safari WebKit and Firefox SpiderMonkey.

KJS closely resembles the language standard (see Figure 5.2), which facilitates visual inspection, and allows to measure the semantic coverage of a test suite. We found that the ECMAScript 5.1 conformance test suite misses several semantic behaviors described in the language standard. We wrote tests for the uncovered semantics and discovered a number of bugs both in production JavaScript engines and in existing formal semantics. Measuring conformance test suite coverage has been considered infeasible in [72, 73, 74], because JavaScript implementations are highly optimized and do not follow the standard document line by line. KJS thus paves a way for the JavaScript language standard committee to systemically measure the semantic coverage of their conformance test suite.

KJS has been defined in a style that is suitable also for reasoning about JavaScript programs. We have verified several non-trivial programs and demonstrated how KJS can be used for finding a security vulnerability (Section 5.4).

The complete KJS semantics of JavaScript, as well as all the artifacts discussed in the chapter are publicly available at [75].

5.1 ECMASCRIPT 5.1

ECMAScript is the official JavaScript language standard. The latest version, at the time of writing, is ECMAScript 5.1 [76]. Compared to the previous version, ECMAScript

3,[1] ECMAScript 5.1 adds new features for more robust programming such as the strict mode, better integration with the DOM object such as accessor (getter/setter) properties, and new APIs such as JSON. The upcoming version ECMAScript 6 [78], under active development, will add new features such as classes, modules, iterators and collections, and generators and promises (for asynchronous programming).

ECMAScript 5.1 specifies not only the language core but also standard libraries. It consists of 16 chapters and 6 appendices, for a total of 258 pages. Chapters 1-5 give an overview of the language; Chapters 6-7 describe lexing and parsing; Chapter 8 describes runtime types such as string, number, and object; Chapter 9 discusses type conversions; Chapter 10 covers environments and execution contexts; Chapters 11-14 and 16 describe the semantics of language constructs: expressions, statements, functions, programs, and errors; Chapter 15 presents the standard libraries; Appendix A is dedicated to the language grammar and Appendix B compatibility.

ECMAScript 5.1 gives algorithmic descriptions for all language constructs, to precisely specify their behaviors. It also defines various internal semantic functions, called abstract operations, to effectively describe high-level language constructs. For example, Figure 5.2(a) presents an abstract operation, [[Get]], which takes an object O and a property name P, and returns P's value of O. This property lookup function precisely describes its behavior by using an informal pseudo-code algorithm. It also interacts with other internal semantic functions such as [[GetProperty]] and IsDataDescriptor.

5.2 FORMAL SEMANTICS OF JAVASCRIPT IN K

KJS faithfully describes ECMAScript 5.1 in $\mathbb{K}$. It defines the core language semantics, and also several standard libraries. KJS is systematically derived from, and has a close correspondence with, the language standard.

5.2.1 Program Configuration

Figure 5.3 shows the KJS configuration, or state, which holds objects, environments, and the execution context.

Objects An object is a map from property names to values with attributes. Each object is connected with another object via a [[Prototype]] link. An object inherits other objects

[1] ECMAScript 4 was abandoned due to fundamental disagreements between the committee members [77].

When the [[Get]] internal method of *O* is called with property name *P*, the following steps are taken:

1. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, return **undefined**.
3. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
4. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]].
5. If *getter* is **undefined**, return **undefined**.
6. Return the result calling the [[Call]] internal method of *getter* providing *O* as the **this** value and no arguments.

(a) ECMAScript 5

```
rule Get(O:Oid,P:Var)
  => Let $desc = GetProperty(O,P);                                    /* Step 1 */
     If $desc = Undefined then Return Undefined;                      /* Step 2 */
     If IsDataDescriptor($desc) = true then Return $desc."Value";     /* Step 3 */
     Let $getter = $desc."Get";                                       /* Step 4 */
     If $getter = Undefined then Return Undefined;                    /* Step 5 */
     Return Call($getter,O,Nil);                                      /* Step 6 */
```

(b) KJS

Figure 5.2: Correspondence between ECMAScript 5 and KJS semantics

along with the prototype chain. In the configuration, an object is represented by an obj cell. The "*" appearing next to the obj cell name in the configuration tells $\mathbb{K}$ that zero, one or more cells with that name can occur at that position in the configuration. An obj is identified by oid, and contains two maps: properties and internalProperties. The properties stores user-level properties, while internalProperties is for internal use only.

Environments An environment is a map from variables to values. Each environment is created when the program control enters a new scope, and is connected with its outer scope environment. The environment remains even after the program control exits from the scope. In the configuration, an environment is represented by the env cell. An env is identified by eid and contains an outer link and a declEnvRec map. In case of the global scope and the with block, however, the env has an objEnvRec map instead of declEnvRec. A "?" appearing next to a cell name tells $\mathbb{K}$ that zero or one cells with that name can appear in the configuration at that position.

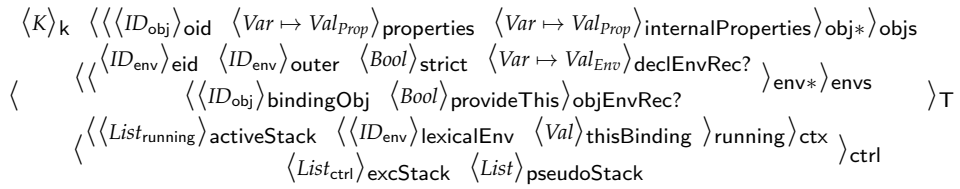


Figure 5.3: Configuration

Execution context An execution context consists of an environment and the this value. A new execution context is created whenever the program control enters a function, and discarded when the function returns. In the configuration, the current execution context is represented by the running cell. When a new execution context is created, the current one is pushed into the activeStack cell (structured as a list).

5.2.2 Semantics Description Language

KJS essentially defines two languages: the JavaScript language and its semantics description language. ECMAScript 5.1 presents semantic behaviors in pseudo-code; see Figure 5.2. To faithfully describe them, we first formally define this pseudo-code language, which is a minimal imperative language with let-bindings and branches. It does not have loops, since iteration can be achieved by recursively applying rules.

5.2.3 Semantics of Language Constructs

We define the semantics of each language construct by systematically translating its informal algorithmic description in the language standard into formal pseudo-code as defined in Section 5.2.2. Figure 5.2 shows an example of the translation. Each step of (a) is translated to its corresponding pseudo-code statement of (b). For example, step 1

> Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.

is translated to the formal definition of (b):

> Let $desc = GetProperty(O,P);

This approach not only contributes to the faithfulness of our semantics, but also expedites the semantics development.

We only describe a few relevant or interesting constructs.

```
var base = Object.create(Object.prototype, {
  y : {value:0, enumerable:false,configurable:true}
});
var derived = Object.create(base, {
  x : {value:1, enumerable:true, configurable:true},
  y : {value:2, enumerable:true, configurable:true}
});
var i = 0;
for (var k in derived) {
  if (i === 0) delete derived.y;
  console.log(k + ":" + derived[k] + ";");
  ++i;
}
```

Figure 5.4: Undefined for-in program: Safari WebKit and Chrome V8 output x:1; y:0;, while Firefox SpiderMonkey outputs x:1;.

The for-in construct, which iterates through all the enumerable properties of a given object, is non-deterministic. The enumeration order of the properties is not specified, but implementation-dependent. A loop may have a different iteration order even in the same program. In order to correctly specify this non-determinism, our semantics employs the set-theoretical 'choice' operation to select each property non-deterministically. $\mathbb{K}$ provides a 'search'-mode execution feature which explores all possible execution traces, in this case all possible enumeration orders.

Furthermore, certain semantic behaviors are under-specified in the language standard [79]. A property is enumerable when its enumerable attribute is true. The iterated properties include not only the object's own properties, but also the inherited ones. An inherited property, however, is excluded when it is shadowed. Also, if a property is deleted during the iteration before it is visited, the property is skipped. But what if a property is shadowed and the property causing the shadowing is deleted before its visit? Is the original property supposed to be visited? The language standard leaves this behavior unspecified, without even stating if it is implementation-dependent or not. The consequence is that different JavaScript implementations have different behaviors in this situation. Figure 5.4 shows a for-in loop on the derived object, which inherits the base object shadowing the property y. In the loop, the shadowing property derived.y is deleted before it is visited;[2] the shadowed property base.y now becomes visible and can be considered for enumeration in the next iteration. For this program, Safari WebKit and

[2]Suppose an iteration order where x is visited first.

RULE TRY

$$\langle \frac{\texttt{try}\ S\ \texttt{catch}\ (X)\ S'}{S \curvearrowright \texttt{endtry}} \curvearrowright K \rangle_{\mathsf{k}}\ \langle\langle \frac{\cdot}{(X, S', K, C)}\ \_ \rangle_{\mathsf{excStack}}\ C \rangle_{\mathsf{ctrl}}$$

RULE THROW

$$\langle \frac{\texttt{throw}\ V_e \curvearrowright \_}{\texttt{let}\ X = V_e\ \texttt{in}\ S' \curvearrowright K} \rangle_{\mathsf{k}}\ \langle\langle \frac{(X, S', K, C)}{\cdot}\ \_ \rangle_{\mathsf{excStack}}\ \frac{\_}{C} \rangle_{\mathsf{ctrl}}$$

RULE END-TRY

$$\langle \frac{V_S \curvearrowright \texttt{endtry}}{V_S}\ \cdots \rangle_{\mathsf{k}}\ \langle\langle \frac{(\_, \_, \_, \_)}{\cdot}\ \_ \rangle_{\mathsf{excStack}}\ \_ \rangle_{\mathsf{ctrl}}$$

Figure 5.5: Exception semantics

Chrome V8 output x:1;y:0; since they decided to visit base.y, while Firefox SpiderMonkey outputs x:1; since it does not visit base.y whose enumerable attribute is false.

KJS makes these unspecified behaviors explicit: it reports an 'unspecified' error when a for-in loop encounters the unspecified situation in Figure 5.4. This feature needs to be defined in order to have a complete semantics, and can be used to check the portability of JavaScript programs. Section 5.4.1 discuses this in more detail.

While user-level exceptions (raised with throw) are well described in ECMAScript 5.1, internal exceptions (e.g., ReferenceError) are not. The described exception propagation mechanism only applies to the user-level exceptions. To define both user-level and internal exceptions in a uniform way, KJS employs an exception handling mechanism that is commonly used by many programming language semantics. Figure 5.5 shows the essential rules. The rule TRY starts to execute S, pushing the current execution context in the excStack cell. If an exception occurs, the rule THROW restores the saved context C and executes the catch block S'. If no exception occurs, the rule END-TRY discards the saved execution context and proceeds to the next computation.

JavaScript's switch has a surprising fall-through semantics: it does *not* fall through at default. For example, the following switch statement contains two regular cases and one default with no break statement.

```
function foo(n) {
  switch(n) { case 1:  console.log("case 1;");
              default: console.log("default;");
              case 2:  console.log("case 2;"); } }
```

The function call foo(1) outputs "case 1; default; case 2;", and foo(2) outputs "case 2;". However, foo(0) outputs only "default;", because JavaScript does not allow the fall-through at default. This behavior is unusual compared to other programming languages in which foo(0) would output "default; case 2;".

JavaScript has a strict mode execution feature, which also contains tricky corner cases. It was newly introduced in the language standard, ECMAScript 5.1, as a workaround for several design mistakes (e.g., the this resolution). A strict mode execution is only applied to a strict mode code, indicated by a 'use strict' directive. For example, the following is a strict mode code; its execution throws a ReferenceError exception since the undeclared variable 'x' is not allowed to be used in strict mode:

```
`use strict';
eval(`x = 1;'); // throws a ReferenceError
```

However, the following program, which appears to be equivalent to the above, does not report the exception:

```
`use strict';
var myeval = eval;
myeval(`x = 1;'); // no ReferenceError
```

The reason is that an eval code inherits the strict mode only when it appears in a direct call to eval. In the first program, 'x = 1;' is evaluated in strict mode because eval is called directly on it. However, in the second program, 'x = 1;' is evaluated in *non*-strict mode because eval is called indirectly, and x is assigned 1 in the global scope.

Function and variable declarations are evaluated before other statements, with function declarations evaluated before variable declarations. In combination with shadowing lack of block scoping, unexpected results can occur. The following seemingly equivalent functions return different values, 2 and 1:

```
function f1() { function f() { return 1; }
                function f() { return 2; }
                return f(); } // 2
function f2() { var f = function () { return 1; };
                function f() { return 2; }
                return f(); } // 1
```

This is because the first line of f2 is a variable and not a function declaration, so is evaluated after the function declaration in the next line, overwriting it.

15.2.3.5 Object.create (O [, Properties])

The **create** function creates a new object with a specified prototype. When the **create** function is called, the following steps are taken:

1. If Type(*O*) is not Object or Null throw a **TypeError** exception.
2. Let *obj* be the result of creating a new object as if by the expression new Object() where Object is the standard built-in constructor with that name
3. Set the [[Prototype]] internal property of *obj* to *O*.
4. If the argument *Properties* is present and not **undefined**, add own properties to *obj* as if by calling the standard built-in function Object.defineProperties with arguments *obj* and *Properties*.
5. Return *obj*.

(a) ECMAScript 5

```
Object.create = function (O, Properties) {
  if (!(@IsObject(O) || O === null)) throw TypeError("Invalid arguments");   // Step 1
  var obj = new Object();                                                    // Step 2
  @SetInternalProperty(obj, "Prototype", O);                                 // Step 3
  if (Properties !== undefined) Object.defineProperties(obj, Properties);    // Step 4
  return obj;                                                                // Step 5
};
```

(b) Our semantics, independent of semantic framework

Figure 5.6: Self-hosted standard built-in objects semantics

When a function is called, an arguments object is created holding the function's arguments values. Modifying the arguments object is allowed, but it has different semantics depending on whether we are in a strict mode or not. If non-strict, arguments is aliased with the formal parameters; if strict, arguments has its own properties, not affecting the formal parameters. For example, below f(0) returns 1 while g(0) returns 0:

```
function f(x) { arguments[0] = 1; return x; }
function g(x) { "use strict";
                arguments[0] = 1; return x; }
```

The definition of the `eval` function is straightforward in KJS: it parses the argument and then evaluates it in the eval execution mode. Parsing is handled by the '`#parse`' primitive of the $\mathbb{K}$ framework, which uses a parser automatically generated from the given syntax declarations.

5.2.4 Standard Libraries

Although KJS aims at defining the semantics of the core JavaScript language, we have also given semantics to some essential standard built-in objects. For example, we completely defined the Object, Function, Boolean, and Error objects, because they expose internals of the language semantics. Also, we partially defined the Array, String, Number and Global objects; specifically, all their constructors and only a group of internal methods, such as Array's [[DefineOwnProperty]] and String's [[GetOwnProperty]]. These internal methods are essential because they determine the fundamental behavior of their corresponding objects, so that the rest of these objects' behaviors can be defined entirely in JavaScript invoking these internal methods, as explained shortly. Finally, we have not given semantics to the Math, Date, RegExp, and JSON objects, because these are orthogonal to the semantic approach and can be implemented in plain JavaScript [80].

Figure 5.6 shows by means of an example our simple approach to give semantics to built-in objects based on the already defined internal methods: JavaScript itself. Each step of (a) is translated to the corresponding JavaScript code of (b); Steps 1 and 3 employ the internal methods @IsObject and @SetInternalProperty.[3] KJS defines dozens of such internal methods that are difficult or impossible to define in JavaScript. Based on these, the built-in objects can be completely defined in JavaScript, concisely and independently from the employed semantic formalism.

5.3 EVALUATION

We evaluate KJS with respect to completeness and development cost.

5.3.1 Completeness

To evaluate the completeness of KJS and to measure the progress during its development, like the authors of previous JavaScript semantics [81, 71], we tested our semantics against the official ECMAScript 5.1 language conformance test suite, test262 [82]. The test262 consists of 11,578 test programs which are classified according to each of the chapters of ECMAScript 5.1. Chapters 1-5 have no tests; Chapters 6-7 have 716 tests for parsing; Chapters 8-14 have 2,782 tests for the language core; and Chapter 15 and Annex

[3] We employ a different namespace for the internal semantic functions, using names starting with '@' which cannot appear as program variables (since '@' is not an *IdentifierStart* character [76]). Thus we can safely introduce internal functions without polluting the global object.

Formal Semantics	Passed	Failed	% passed
KJS	2,782	0	100.0%
Politz *et al.* [81][5]	2,470	345	87.7%
Bodin *et al.* [71]	1,796	986	64.6%
JavaScript Engines	**Passed**	**Failed**	**% passed**
Chrome 35.0 (V8 3.25.28)	2,782	0	100.0%
Firefox 30.0 (SpiderMonkey 30)	2,780	2	99.9%
Safari 7.0.4 (WebKit 537.76.4)	2,780	2	99.9%

Table 5.1: Comparison of formal semantics and product engines tested against the ECMAScript conformance test suite

B have 8,080 tests for standard libraries. Like previous JavaScript semantics efforts, to keep the project manageable we targeted only the 2,782 tests corresponding to the core language. As explained in Section 5.2.4, we have also defined some essential standard built-in objects and internal methods, so that the remaining methods can be implemented in plain JavaScript. However, providing JavaScript code for the hundreds of standard library methods is beyond the scope of this thesis.

Table 5.1 shows that KJS is the most complete JavaScript semantics to date, passing all of the 2,782 ECMAScript 5.1 core tests. It is even more standards-compliant than production JavaScript engines such as Safari WebKit and Firefox SpiderMonkey. While the 2,782 tests are supposed to test the language core, several tests use library calls, e.g. to trigonometric functions. To test such programs modulo the unsupported libraries, we used a feature of $\mathbb{K}$ allowing to employ an external library implementation; specifically, we used the Node.js implementation of Math.sin, Number.toFixed, and Number.toString.[4] Further, to overcome some current parsing limitations of $\mathbb{K}$ (acknowledged by $\mathbb{K}$'s developers and scheduled for fixing), we pre-process the input JavaScript program using the SAFE framework [83] for automatic semicolon insertion and the sed utility for translating unicode characters.

Currently, the $\mathbb{K}$ interpreter takes an hour to execute all of 2,782 test programs in a machine with Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB 1333MHz. $\mathbb{K}$ development team, however, is currently working on an OCaml backend to compile $\mathbb{K}$ definitions to OCaml programs for faster execution. With that, the execution time is expected to drop from an hour to minutes.

[4]Only a dozen of tests depend on this, which is not a significant number.

[5]Note that S5 was tested against the previous version of the ECMAScript 5 test suite, and the total number of tests is slightly bigger than the latest one. Also, S5 reported test results for standard libraries, which is not presented here since we focus on the language core.

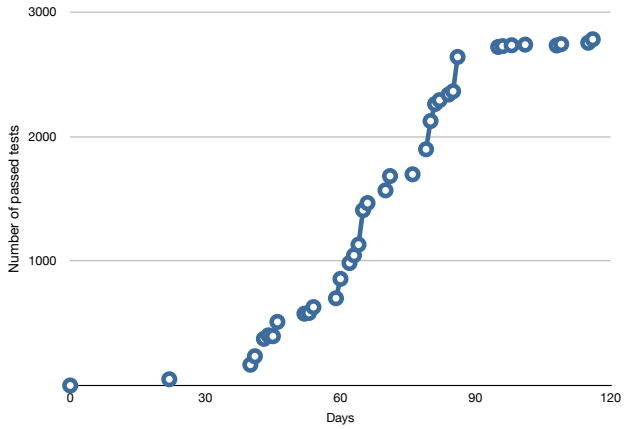


Figure 5.7: KJS semantics development progress

5.3.2 Development Cost

The development of KJS took only four months by a first year PhD student, with no prior knowledge of JavaScript or of the $\mathbb{K}$ semantic framework. We believe that this was possible thanks to the following: (1) $\mathbb{K}$'s executability, allowing us to test and fix the semantics immediately as inconsistencies were detected; (2) Formalizing the pseudo-code used in the language standard, which allowed us to easily and systematically formalize the informal semantics; (3) $\mathbb{K}$'s modularity, allowing us to change the structure of the program configuration (e.g., to add new features to the language) without having to change the existing rules (e.g., to add exceptions we had to add new cells to the configuration and three independent rules, but no other rules had to be touched—Figure 5.5).

A side objective of our effort was to demonstrate that the programming language semantics field has matured enough that language designers should consider defining a complete formal semantics to their language as part of the (long) standardization process. It is no longer true that defining a formal semantics to a language takes too long to be worthwhile. To bring more evidence in this direction, we measured and logged the KJS development progress rigorously. Figure 5.7 shows how many tests passed each day during the project timeframe. In the first month we developed the semantic foundations such as syntax, program configuration, prototype chains, environments, and execution contexts. In the next two months, we defined individual language constructs. Due to the modularity of the employed framework, during this period the number of passed tests linearly increased as each language construct was defined. In the last month we finished our semantics by addressing specific details and corner cases revealed by failed tests, until all of them eventually passed.

5.4 APPLICATIONS

Here we list a few applications of our semantics, mentioning that these were driven by our own interests and that they are by no means exhaustive. The message we want to convey is that a formal semantics can be useful well beyond just giving a reference model/implementation for the defined language.

5.4.1 Checking Portability

As seen in Section 5.2.3, ECMAScript 5.1 contains unspecified behaviors, e.g., the for-in loop. Since unspecified behaviors are implementation-dependent, JavaScript programs may not be portable, working differently with different JavaScript engines in different web browsers. Detecting unspecified behaviors in JavaScript programs is not trivial. Simply running the program in different JavaScript engines is not sufficient: even if they all agree on some unspecified behavior now, this may change in future releases.

KJS can be trivially used to detect unspecified behaviors of JavaScript programs, as it 'gets stuck' when no rule matches (i.e., no semantics exist). For the unspecified behavior in Figure 5.4, e.g., KJS gets stuck when the loop iteration encounters y, after the output x:1;. Besides unspecified behaviors, we also need to check for non-deterministic behaviors; e.g., to ensure that the iteration order of a for-in loop is irrelevant. $\mathbb{K}$ provides a 'search'-mode execution feature which explores all feasible execution traces.[6]

5.4.2 Finding Bugs and Improving the Test Suite

The ECMAScript standards committee has made an impressive effort to provide a conformance test suite that systematically ensures that all the features of ECMAScript 5.1 and their subtle interactions are covered, so that JavaScript engines converge on a language standard. However, the semantic coverage of the test suite has not been well-studied, and indeed, some behaviors have escaped untested [72]. Using KJS, we found that despite the large number of tests, certain semantic behaviors are still not tested. For example, surprisingly, there is no test for the peculiar fall-though semantics of the default case for switch (Section 5.2.3). Writing tests to cover the untested behaviors, we found bugs in all production JavaScript engines and in previous semantics.

[6]It is also possible to check confluence of unspecified behaviors (i.e., ensuring that unspecified behaviors are irrelevant) using the 'search'-mode execution, but developing such a sophisticated portability checker is an orthogonal problem, which we leave as future work.

Page #	Section # - Step #	KJS	Po	Bo	CR	FF	SF
p35	8.7.1 GetValue (V) - [[Get]], Step 6	○	×	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 2.a	○	○	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 2.b	○	⊗	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 4.a	-	-	-	-	-	-
p36	8.7.2 PutValue (V, W) - [[Put]], Step 4.b	-	-	-	-	-	-
p36	8.7.2 PutValue (V, W) - [[Put]], Step 6.a & 6.b	○	○	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 7.a	○	×	○	○	×	○
p40	8.12.4 [[CanPut]] (P) - Step 8.a	○	⊗	⊗	○	○	○
p53	10.2.1.1.3 SetMutableBinding (N,V,S) - Step 4	○	×	○	×	○	×
p53	10.2.1.1.4 GetBindingValue(N,S) - Step 3.a	-	-	-	-	-	-
p53	10.2.1.1.5 DeleteBinding (N) - Step 2	-	-	-	-	-	-
p54	10.2.1.1.5 DeleteBinding (N) - Step 4 & 5	○	⊗	○	○	○	○
p55	10.2.1.2.4 GetBindingValue(N,S) - Step 4.a	-	-	-	-	-	-
p59	10.5 Declaration Binding - Step 5.e.iii.1	○	○	○	○	○	○
p59	10.5 Declaration Binding - Step 5.e.iv, 1st	○	⊗	⊗	○	○	×
p59	10.5 Declaration Binding - Step 5.e.iv, 2nd	○	⊗	⊗	○	○	×
p62	10.6 [[DefineOwnProperty]], Step 4.a, else	-	-	-	-	-	-

○: Passed ×: Failed ⊗: Not applicable (unsupported) -: Infeasible semantic behaviors
Po: Politz *et al.* [81] *Bo*: Bodin *et al.* [71]
CR: Chrome 38.0 (V8 3.28.71) *FF*: Firefox 32.0 (SpiderMonkey 32) *SF*: Safari 7.0.4 (WebKit 537.76.4)

Table 5.2: Behaviors *not* covered by the ECMAScript 5.1 conformance test suite. Manually written tests exercising these uncovered behaviors revealed bugs in production JavaScript engines and in previous JavaScript semantics.

How can we measure the semantic coverage of a conformance test suite? One possibility is to run it through several JavaScript implementations using code coverage tools, and project the result back to ECMAScript 5.1. However, this is impractical, as it is not viable to match optimized implementation code to corresponding ECMAScript 5.1 pseudo-code and filter out implementation-specific code [73].

Due to its one-to-one correspondence with ECMAScript 5.1, KJS provides a direct semantic coverage measure for a test suite. This way we found that there are exactly 17 semantic rules in the core semantics which are not covered by the test suite, each corresponding to the language standard as shown in Table 5.2. We succeeded to manually write test programs that hit 11 out of 17 behaviors, thus improving the overall quality of the conformance test suite. It took two days to manually write (or show infeasibility of) the tests for the 17 cases. Finding tests for the semantics is essentially the same as finding tests for conventional programs. For each uncovered semantic rule, we examine a kind of a path condition that leads to the rule, and find a solution (i.e., a test program) that

satisfies the path condition. Automatic test case generation techniques may be used to mechanize this process, but in this thesis we have done all the work manually.

As seen in Table 5.2, the 11 new tests uncovered bugs in both production JavaScript engines and in existing semantics. Moreover, the remaining 6 semantic behaviors are infeasible, that is, they represent flaws in the language standard itself. These bugs were reported, confirmed, and fixed [84].[7] Below we discuss two out of the 11 new tests, and one of the 6 infeasible behaviors.

Step 5.e.iv of Section 10.5 in the language standard describes how to handle duplicate global function declarations and is not covered by the test suite. The following program

```
Object.defineProperty(this, "f", {
  "value"    : 0,     "enumerable"   : false,
  "writable" : false, "configurable" : false });
eval(" function f() { return 0; } "); // TypeError
```

is supposed to raise a `TypeError` exception according to the standard, since the function `f` is declared while there already exists another `f` whose `writable`, `enumerable`, and `configurable` attributes are all false. Safari WebKit wrongly ignores the duplicate function declaration, disobeying the standard; Chrome V8 and Firefox SpiderMonkey behave correctly.

Step 4 of Section 10.2.1.1.3 in the standard describes a case of updating an immutable variable which is not covered by the test suite either. In the following program

```
"use strict";
var f = function g() { g = 0; /*TypeError*/ }; f();
```

`g` is immutable, but the body attempts to update it. According to the standard, a TypeError exception must be raised. However, only Firefox SpiderMonkey conforms, while Chrome V8[8] and Safari WebKit do not, wrongly ignoring the update statement.

For an example of infeasible semantic behavior, consider Section GetBindingValue(N,S) in the standard which describes the environment lookup semantics for a given variable N, and its Step 3.a which discusses the case where N has an uninitialized immutable binding. However, this case is infeasible. There are only two situations where immutable bindings can occur, namely in the arguments object in a strict mode function and in the name of a recursive function expression[9] in its function body's environment. But according to the

[7] It turned out that two of them had already been reported [85, 86].

[8] Fixed in Chrome 41.0 (V8 4.1.0).

[9] Function 'expression' and *not* 'declaration', because in the latter the function name is declared in a global environment and is mutable.

```
function mkSend(rawSend) {
  var whiteList = { "http://www.trust.com": true,
                    "http://www.good.com" : true };
  function newSend(target, msg) {
    if (whiteList[target]) rawSend(target,msg);
    else console.error("Rejected."); }
  return newSend; }

var send = mkSend(function (target, msg) {
  console.info("Sent " + msg + " to " + target);});
```

Figure 5.8: Secure Message Sending

standard, in both cases the bindings are initialized right after creation, thus there is no way to have uninitialized immutable bindings.

We also ran the additional 11 tests on the existing semantics, and discovered a number of bugs, as shown in Table 5.2.

5.4.3 Symbolic Execution

Here and in Section 5.4.4 we illustrate how to derive JavaScript program reasoning tools from generic tools offered by the employed semantic framework. $\mathbb{K}$ allows for terms it reduces to be symbolic, that is, to contain mathematical variables and constraints on them. As semantic rules are applied, constraints are accumulated and solved using Z3 [29] (which is incorporated in $\mathbb{K}$). In this section we show how this capability can be used to find a known security vulnerability, and in the next section how it can be lifted into a fully-fledged JavaScript program verifier.

Consider the program in Figure 5.8, introduced by Fournet *et al.* [64], which contains a secure message sending function. The send method sends messages only to addresses in the white list. For example, the following should be rejected:

```
send("http://www.evil.com","msg"); // Rejected
```

Suspecting a global object poisoning attack [87], we construct a configuration adding a symbolic property P with symbolic value V in the Object.prototype object, equivalent to executing Object.prototype[P] = V. Then we execute the send request above using $\mathbb{K}$'s search mode, looking for a state where the message was sent. The symbolic search execution then returns the constraint, "P = "http://www.evil.com" $\wedge$ (V = true $\vee$ *V is a non-empty string* $\vee$ *V is a non-zero number* $\vee$ *V is an object*)", modeling the instances of the suspected attack model; e.g.,

```
Object.prototype["http://www.evil.com"] = true;
```

executed before the malicious send call above allows the message to be sent to the malicious address. That is because Object.prototype is inherited by all objects, so the if-condition `whiteList["http://www.evil.com"]` returns true even if the whiteList does not include the evil address. This problem can be fixed by creating an isolated object for `whiteList` using `Object.create(null)`:

```
var whiteList = Object.create(null);
whiteList["http://www.trust.com"] = true;
whiteList["http://www.good.com" ] = true;
```

5.4.4 Program Verification

𝕂 offers support for program verification based on rule-based semantics, at no additional cost (with no need to define another semantics) [3]. Program properties are specified as reachability rules. 𝕂 uses a sound and relatively complete proof system for deriving such rules from the operational semantics rules, which amounts to:

1. Performing symbolic execution of code without repetitive behavior using the semantics rules; and

2. Reasoning about repetitive constructs (loops, recursion).

Like in Hoare logic, all the repetitive constructs need to be annotated with specifications. The verification is automatic: the user only provides the specifications. The specifications are given as reachability rules between symbolic configurations with constraints. We keep the rules compact by:

1. Using the 𝕂 notations and conventions (as described in Section 2.5) to describe the symbolic configurations; and

2. Computing the static part of the symbolic configurations (e.g. the builtin-in objects) using the semantics.

For all practical purposes, the standard pre-/post-conditions can be automatically desugared into reachability rules, although we have not implemented it yet.

To test the viability of using the generic reachability verification infrastructure with the JavaScript semantics, we verified a few JavaScript programs implementing data-structures operations. Table 5.3 summarizes our experiments. For each function we verified the

Function	Size (LOC)	Time (s)
List reverse	13	8
List append	12	13
BST find	12	7
BST insert	23	12
BST delete	34	17
AVL find	11	7
AVL insert	87	109
AVL delete	106	174

Table 5.3: Verification Result

full functional correctness. Due to space limitations, we discuss only the AVL insert function (the code is shown in Figure 5.9). The specification of AVL insert in a form of a pre-/post-condition that would desugar into our current reachability rule is:

```
function insert(v, t)
//@requires tree(t)(T) /\ avl(T)
          /\ tree_height(T) < INT_MAX
//@ensures tree(t)(T') /\ avl(T')
         /\ tree_keys(T') == { v } U tree_keys(T)
         /\ | tree_height(T') - tree_height(T) | <= 1
```

The precondition requires that the function is passed an AVL tree t, and that the height h of t is small enough such that both h and $h+1$ can be represented on a float-point number without precision loss. The postcondition ensures that the function returns an AVL tree t', that the keys of t' are the keys of t plus the inserted key, and that the height h' of t' is either h or $h+1$. The bound on h is specific to JavaScript, because JavaScript only provides floating-point arithmetic. The AVL, keys, and height abstractions are defined recursively in a standard way.

The overall verification times in Table 5.3 are quite acceptable, considering that our program verifier is obtained for free from KJS and that, at the best of our knowledge, there is no other program verifier for JavaScript that can verify such complex programs to compare with ours. Also, our times are only twice slower on average than those in [3] for similar properties but for a toy C-like language. The times for AVL insert and delete are large due to the fact that the helper functions (balance, left_rotate) are not given specifications, instead they are called using their operational semantics, which leads to a larger number of paths to analyze. The effort to verify these examples took approximately one man-week. Most of the work went into finding the JavaScript specific

```
function insert(v, t) {
  if (t === null) return make_node(v);
  if (v < t.value) t.left = insert(v, t.left);
  else if (v > t.value) t.right = insert(v, t.right);
  else return t;
  update_height(t); return balance(t); }

function balance(t) {
  if (height(t.left) - height(t.right) > 1) {
    if (height(t.left.left) < height(t.left.right))
      t.left = left_rotate(t.left);
    t = right_rotate(t);
  } else if (height(t.left) - height(t.right) < -1) {
    if (height(t.right.left) > height(t.right.right))
      t.right = right_rotate(t.right);
    t = left_rotate(t); }
  return t; }

function left_rotate(x) {
  var y = x.right; x.right = y.left; y.left = x;
  update_height(x); update_height(y); return y; }

function right_rotate(x) { ... }
```

Figure 5.9: AVL Tree Insertion

part of the specifications (like the bound on the height in the AVL example). We believe that our preliminary evaluation shows a realistic potential of using the KJS semantics for JavaScript program verification.

5.4.5 Discussion

Although KJS passes all the tests in the ECMAScript 5.1 conformance test suite for the core language, which is the reason why we call it a 'complete semantics', there is no guarantee that our semantics is necessarily correct. In the absence of a reference semantics, we believe that the best we can do to validate our semantics at this stage is to test it heavily against as many tests as possible, which we did, and to reason with it and prove certain expected properties of it, which we have not done yet but we plan to do as soon as a Coq backend becomes available for $\mathbb{K}$. In particular, a formal relationship between our semantics and that by Bodin *et al.* [71] can also be shown then using Coq.

CHAPTER 6: RELATED WORK

This chapter presents related work on program verification, program equivalence, and language semantics.

6.1 PROGRAM VERIFICATION

The program verification literature is rich. We only discuss work close to ours, omitting theoretical work that has not been applied to large languages or work on interactive verification.

A popular approach to building program verifiers for real-world languages is to translate to an intermediate verification language (IVL) and do verification at the IVL level. This results in some reusability, as the VC generation and reasoning about state properties are implemented only once, at the IVL level. However, the development of translators is both time consuming and susceptible to bugs. Boogie [28] is a popular IVL integrated with Z3. There are several verifiers built on top of Boogie, including VCC [18], HAVOC [88], Spec# [89], Dafny [90], and Chalice [91]. VCDrayd [17] is a separation logic based verifier built on top of VCC. Why3 [92] is another IVL, also integrated with SMT solvers (and other provers). Tools built on top of Why3 include Frama-C [92] and Krakatoa [93]. There are many other practical VC generation based tools (with or without an IVL), including Verifast [94] and jStar [95]. In contrast, we use existing operational semantics directly for verification, without any translation to IVLs or language-specific VC generation.

Recent work proposes translating to a set of Horn clauses instead of an IVL [96]. A semantics based-approach to translation to Horn clauses for a fragment of C is presented in [97], but it is unclear if the approach is generic enough to scale to the entire C or to other real-world languages. An approach for using the interpreter source code as a model of the language in for symbolic execution is proposed in [98], but it is used to generate tests, not verify programs.

We fully share the goal of the mechanical verification community to reduce the correctness of program verification to a trusted formal semantics of the target language [99, 100, 101, 102, 103], although our methods are different. Instead of a framework to ease the task of giving multiple semantics of the same language and proving systematic relationships between them, we advocate developing *only one* semantics, operational, and offering an underlying theory and framework with the necessary machinery to achieve

the benefits of multiple semantics without the costs. Bedrock [104] is a Coq framework which uses computational higher-order separation logic and supports semi-automated proofs. It can serve as an IVL, and be the target of translations from other languages which can be certified in Coq based on their operational semantics. Our approach works with the operational semantics directly, and thus does not need any such proofs.

Dynamic logic [105] adds modal operators to FOL to embed program fragments within specifications, but still requires language-specific proof rules (e.g., invariant rules). KeY [106] offers automatic verification for Java based on dynamic logic. Matching logic also combines programs and specifications for static properties, but dynamic properties are expressed in reachability logic which has a language-independent proof system that works with any operational semantics.

Semantics-Based Verification A first version of a language-independent proof system for reachability is given in [1], and [2] shows a mechanical translation of Hoare logic proof derivations for IMP to it. The Circularity proof rule was introduced in [3]. Support for operational semantics using conditional rules is introduced in [4], and support for non-determinism in [5]. These previous results are mostly theoretical, with MatchC a prototype hand-crafted for KernelC mixing language-independent reasoning with the operational semantics of KernelC.

Smart Contract Verification While there exist several static analysis tools [107, 108, 109, 110] tailored to check certain predefined properties, here we consider only the verification tools backed by a full-fledged theorem prover that allows to reason about arbitrary (full functional correctness) properties. Specifically, Bhargavan et al. [111] and Grishchenko et al. [112] presented a verification tool based on the F* proof assistant, and Amani et al. [113] presented a tool based on Isabelle/HOL. These tools, however, adopt only a partial, incomplete semantics of EVM, and thus may miss certain critical corner-case behaviors of the EVM bytecode, which could undermine the soundness of the verifiers. Our EVM verifier, on the other hand, is a verification tool derived from a *complete* and *thoroughly tested* formal semantics of EVM [11], for the first time to the best of our knowledge.

6.2 PROGRAM EQUIVALENCE

The program equivalence literature is rich, and here we discuss only the formal proof systems of program equivalence. For the classic bisimulation and its variants, we refer the reader to Section 2.4.

Namjoshi *et al.* [40] uses a variant of stuttering-bisimulation with ranking functions, first introduced in [114]. Informally, the ranking function returns an integer rank for each pair in the relation which should represent how many times it is allowed for one of the transition systems to stutter while the other advances before the former has to advance. This variant requires matching single transitions only, similarly to strong bisimulation and unlike classic stuttering bisimulation, where a single transition may have to be matched with a finite but unbounded number of transitions, thus leading to large number of generated proof requirements. Cut-bisimulation shares the same property of matching single transitions only and is more appealing for proof automation, since it eliminates the need for the proof generator to produce ranking functions along with the set of synchronization points.

Hur *et al.* [115] presents the relation transition systems (RTS) as a technique for program equivalence proofs suitable for ML-like languages, that combine features such as higher-order functions, recursive types, abstract types, and mutable references. Bisimulation is used as part of the RTS equivalence proof technique. Our notion of cut-bisimulation is orthogonal to RTS and it can be the notion of bisimulation of choice within an RTS equivalence proof. More specifically, our notion of acceptability relation $\mathcal{A}$ is similar to the global knowledge relation used in bisimulation proofs within the RTS proof. However, whereas a global knowledge relation contains a subset relation (named local knowledge) that should be proven to consist only of equivalent pairs, an acceptability relation is assumed from the start to only contain equivalent pairs: this is unavoidable when we want to do an inter-language equivalence proof, since the knowledge of what states are considered equivalent is indispensable for even to define what it means for two different language programs to be equivalent. The authors argue that RTS is a promising technique for inter-language proofs that involve ML-like languages (although they leave the claim as future work), and we believe that the notion of cut-bisimulation can indeed help towards enabling RTS-style inter-language equivalence proofs.

Ciobaca *et al.* [116] proposes the notion of mutual equivalence and presents its proof system, by which our equivalence checking algorithm was inspired. Instead of a proof system, here we propose a bisimulation relation and an algorithm based on it and

symbolic execution, leading to the first language-independent implementation of a checker for equivalence between programs written in two different languages.

6.3 LANGUAGE SEMANTICS

There is a large body of literature on real-world language semantics, and we only discuss efforts that directly influenced us: JavaScript semantics and other large semantics in $\mathbb{K}$.

6.3.1 JavaScript Semantics

We only consider JavaScript semantics attempting to define the full language, not a subset, i.e., ones which like ours aim at establishing a solid foundation for formal JavaScript tools.

Herman and Flanagan (2007) [117] gave an executable semantics of ECMAScript 4. As language standard committee members (Ecma TC39-ECMAScript), their objective was to specify a definitional interpreter of the language. They used ML as a specification language, since it is executable, more precise than English prose, and more easily understandable than mathematical notation. They separately defined the standard libraries in JavaScript itself, which is also what we did. Their semantics, however, is based on ECMAScript 4 which was abandoned, never approved as a standard. Furthermore, unlike ours, their semantics does not facilitate formal reasoning.

Maffeis *et al.* (2008) [118] defined a small-step semantics of ECMAScript 3 and proved some basic properties. Their semantics is based on the older ECMAScript 3, and does not cover the modern JavaScript features such as the strict mode. Also, it is not executable, and cannot be validated against conformance test suites.

Guha *et al.* (2010) [119] and Politz *et al.* (2012) [81] presented a reduced semantics of JavaScript, based on ECMAScript 3 and 5, respectively. They defined a core language, λ_{JS}, and a translation from JavaScript to λ_{JS} together with a (runtime) environment containing internal semantic functions written in λ_{JS} itself. They also implemented an interpreter for λ_{JS}, which, combined with the translator and the runtime environment, allows to execute and test their semantics. Although the reduced semantics is helpful to understand the essentials of JavaScript, there is a gap between it and the actual language specification. Since their semantics does not directly follow the structure of the language specification, it is difficult to manually/visually inspect it and, indeed, it contains a number of bugs

(see Table 5.2). We found that the JavaScript language specification, unlike for other languages, is quite well written, so we decided to follow it faithfully.

Bodin *et al.* (2014) [71] defined a JavaScript semantics in Coq, which, like KJS, follows ECMAScript 5.1. To execute and thus test it, they also implemented an interpreter, manually. Moreover, in order to link it to their semantics, they had to prove their interpreter correct. This step was inevitable, because their Coq specification is not executable—Coq can only extract executables from functions or proofs, not from specifications defined as inductive relations— yet testing is paramount when it gets to large semantics. Defining an interpreter and proving it correct for a complex language like JavaScript is a huge effort;[1] while a laudable and impressive feat in itself, we believe that such heavy approaches may demotivate language designers, for example the standards committee, to adopt a formal semantics. Compare that with KJS, where an interpreter is obtained directly from the semantics at no additional effort, together with other language analysis tools. Moreover, their semantics is incomplete. They omitted several language components such as the for-in loop and array manipulations. Table 5.1 shows that their semantics passes only about 65% of the conformance test suite.

On non-determinism To our knowledge, KJS is the only JavaScript semantics that captures the non-determinism of the language. For example, for the for-in's iteration order, the standard says that the mechanics and order of enumerating the properties is left to the implementation; so from a semantic perspective, any order is possible. Without properly capturing the non-determinism of JavaScript, a semantics of it cannot execute and at the same time formally analyze JavaScript programs (e.g., show that the enumeration order is irrelevant in a given program). For example, Bodin *et al.* [71] chose to not provide a semantics for the for-in construct at all, Maffeis *et al.* [118] to define a partial semantics (with a hole for the enumeration order), and Guha *et al.* [119] and Politz *et al.* [81] to only consider a fixed, arbitrary order (given by Haskell's Hash Tables or OCaml's Map iteration order, respectively).

Verification of JavaScript programs While there is much work on finding bugs and security violations in JavaScript programs, verification of functional correctness of JavaScript programs is less developed. Gardner *et al.* [120] propose a (Hoare logic semantics with state properties specified using) separation logic for a JavaScript fragment. They follow the standard approach by defining an operational semantics as a model of the language,

[1] Indeed, Bodin *et al.* [71] involved 8 people, including domain experts of JavaScript and of Coq, for a year.

and then proving the separation logic sound w.r.t. the operational semantics. Like [71], this has the disadvantage of having to define different semantics of the same language for different purposes, together with soundness proofs, all huge efforts that require maintenance as the language evolves. Compare that to KJS, where only the operational semantics is required, and a deductive program verifier is automatically derived at no additional effort. Furthermore, their separation logic only supports manual reasoning and the programs they verified are significantly simpler than the programs in Table 5.3 which were verified automatically by KJS. Nordio *et al.* [121] present a program verifier for a JavaScript fragment. Their tool is implemented by translation to Boogie, and thus lacks a formal basis. Moreover, they can only verify simple properties that can be directly translated in Boogie.

Semantics for static analysis Other efforts to formally specify JavaScript semantics for the purpose of static analysis have been made. Lee *et al.* [83] provides a reduced semantics (i.e., defining an intermediate language into which the original language is translated), based on ECMAScript 5. Like Guha *et al.* [119] and Politz *et al.* [81], they do not directly follow the actual language specification, making manual/visual inspection hard. Kashyap *et al.* [122] also provides a reduced semantics for the purpose of abstract interpretation. Their semantics, however, is based on ECMAScript 3, and omitted the semantics of `eval`.

6.3.2 Real-World Language Semantics in $\mathbb{K}$

There are four real-world language semantics defined in $\mathbb{K}$ so far, which served as a great source of inspiration for our JavaScript semantics: C [23], PHP [123], Python [124], and Java [52]. All these semantics are executable and they have been validated by a large volume of tests, and demonstrated useful through formal analysis tools produced by the $\mathbb{K}$ framework, same like our KJS.

Ellison and Rosu [23] defined a formal semantics of C11, which was extensively tested against the GCC torture test suite passing 99.2% of the tests, which is more than GCC and Clang passed. The C semantics was also evaluated by debugging, monitoring, and (LTL) model checking of example programs using corresponding tools provided by the $\mathbb{K}$ framework. A main application of their C semantics is undefinedness checking, e.g., in the context of compiler testing, for automatic test-case reduction [125].

Filaretti and Maffeis [123] defined a formal semantics of PHP. Since, unlike for JavaScript, C and Java, there is no official language standard for PHP, they had to heavily

rely on testing against the reference implementation. They evaluated their semantics by model checking certain properties of a web database management tool, phpMyAdmin, and a cryptographic key generation library, pbkdf2.

Bogdanas and Rosu [52] gave a formal semantics of Java 1.4. To mitigate Java's complexity, they split their semantics into two phases: (1) the static semantics enriches the original program by annotating statically inferred information (e.g., types), and (2) the dynamic semantics gives the executable semantics. They evaluated the semantics by model checking multi-threaded programs.

Guth [124] defined a formal semantics of Python 3.3, providing semantics not only for the language constructs but also for the garbage collection mechanism. Being executable, it has been thoroughly tested against more than 600 hand-crafted tests. Like KJS, their semantics covers the core language but only essential parts of the standard libraries.

The most distinguished aspect of our semantics, compared to other language semantics described in $\mathbb{K}$, is the resemblance to the language standard (Figure 5.2); this facilitates visual inspection and allows us to measure the semantic coverage of a test suite. We did it by defining JavaScript on top of a semantics description language (Section 5.2.2), which was possible thanks to the JavaScript language standard being algorithmically described (unlike the language standards of other languages defined in $\mathbb{K}$).

CHAPTER 7: CONCLUSION

We have presented the theory, implementation, and comprehensive evaluation of the language-independent formal methods parameterized by operational semantics. We have developed the cross-language program equivalence checker with the novel property-preserving cut-bisimulation, and also improved the universal deductive program verifier. To demonstrate the practical feasibility of the language-parametric formal methods, we have instantiated the language-parametric deductive program verifier and equivalence checker by plugging-in four real-world language semantics, C, Java, JavaScript, and Ethereum Virtual Machine; and used them to verify full functional correctness of challenging heap-manipulating programs and high-profile commercial smart contracts in an end-to-end manner. We also have specified a complete formal semantics of a high-profile language JavaScript, showing that the specification effort is affordable. We believe that this approach will significantly reduce fragmentation in the verification tool community by eliminating the need to develop a dedicated verifier for each language.

REFERENCES

[1] G. Roșu and A. Ștefănescu, "Towards a unified theory of operational and axiomatic semantics," in *ICALP*, ser. LNCS, vol. 7392, 2012, pp. 351–363.

[2] G. Roșu and A. Ștefănescu, "From Hoare logic to matching logic reachability," in *FM*, ser. LNCS, vol. 7436, 2012, pp. 387–402.

[3] G. Roșu and A. Ștefănescu, "Checking reachability using matching logic," in *OOPSLA*. ACM, 2012, pp. 555–574.

[4] G. Roşu, A. Ştefănescu, S. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *LICS*. IEEE, 2013, pp. 358–367.

[5] A. Stefanescu, S. Ciobaca, R. Mereuta, B. Moore, T. F. Serbanuta, and G. Rosu, "All-Path Reachability Logic," in *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, vol. 8560. LNCS, 2014, pp. 425–440.

[6] B. Moore, L. Peña, and G. Roşu, "Program verification by coinduction," in *27th European Symposium on Programming (ESOP)*, 2018.

[7] D. Park, A. Stefanescu, and G. Rosu, "KJS: A complete formal semantics of JavaScript," in *PLDI*. ACM, 2015, pp. 346–356.

[8] V. Buterin, "Thinking about smart contract security," https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/, 2016.

[9] C. Reitwiessner, "Formal methods roadmap," https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/, 2016.

[10] P. Rizzo, "In formal verification push, ethereum seeks smart contract certainty," http://www.coindesk.com/ethereum-formal-verification-smart-contracts/, 2016.

[11] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Ştefănescu, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *Proceedings of the 31st IEEE Computer Security Foundations Symposium*, ser. CSF 2018, 2018.

[12] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016, 2016.

[13] F. Vogelsteller and V. Buterin, "Erc-20 token standard," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2015.

[14] D. Ryan and C.-C. Liang, "Ethereum Improvement Proposals 1011: Hybrid Casper FFG," https://eips.ethereum.org/EIPS/eip-1011, 2018.

[15] N. Mushegian, D. Brockman, and M. Brockman, "MakerDAO: The DAI Stablecoin System," https://makerdao.com/purple/, 2018.

[16] Gnosis Ltd., "Gnosis Safe," https://safe.gnosis.io/.

[17] E. Pek, X. Qiu, and P. Madhusudan, "Natural proofs for data structure manipulation in C using separation logic," in *PLDI*. ACM, 2014, pp. 440–451.

[18] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *TPHOLs*, ser. LNCS, vol. 5674, 2009, pp. 23–42.

[19] D. Sangiorgi, *Introduction to Bisimulation and Coinduction*. New York, NY, USA: Cambridge University Press, 2011.

[20] G. Roşu, "Matching logic — extended abstract," in *RTA*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 36. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 5–21.

[21] G. Roșu, C. Ellison, and W. Schulte, "Matching logic: An alternative to Hoare/Floyd logic," in *AMAST*, ser. LNCS, vol. 6486, 2010, pp. 142–162.

[22] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *CSL*, ser. LNCS, vol. 2142, 2001, pp. 1–19.

[23] C. Ellison and G. Rosu, "An Executable Formal Semantics of C with Applications," in *POPL*. ACM, 2012, pp. 533–544.

[24] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of C," in *PLDI*. ACM, 2015, pp. 336–345.

[25] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*. MIT, 2009.

[26] G. Berry and G. Boudol, "The chemical abstract machine," *Theoretical Computer Science*, vol. 96, no. 1, pp. 217–248, 1992.

[27] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[28] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*, ser. LNCS, vol. 4111, 2006, pp. 364–387.

[29] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, vol. 4963. LNCS, 2008, pp. 337–340.

[30] S. Antoy, R. Echahed, and M. Hanus, "A needed narrowing strategy," *J. ACM*, vol. 47, no. 4, pp. 776–822, 2000.

[31] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin, "Automated verification of shape and size properties via separation logic," in *VMCAI*, ser. LNCS, vol. 4349, 2007, pp. 251–266.

[32] P. Madhusudan, X. Qiu, and A. Ştefănescu, "Recursive proofs for inductive tree data-structures," in *POPL*. ACM, 2012, pp. 123–136.

[33] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan, "Natural proofs for structure, data, and separation," in *PLDI*. ACM, 2013, pp. 231–242.

[34] J. A. N. Pérez and A. Rybalchenko, "Separation logic + superposition calculus = heap theorem prover," in *PLDI*. ACM, 2011, pp. 556–566.

[35] L. M. de Moura and N. Bjørner, "Efficient E-matching for SMT solvers," in *CADE*, ser. LNCS, vol. 4603, 2007, pp. 183–198.

[36] L. M. de Moura and N. Bjørner, "Generalized, efficient array decision procedures," in *FMCAD*. IEEE, 2009, pp. 45–52.

[37] R. Milner, *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[38] T. F. Serbanuta, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Rosu, "The K Primer (version 3.3)," in *Proceedings of the Second International Workshop on the K Framework and its Applications*, vol. 304. ENTCS, 2013, pp. 57–80.

[39] J. Meseguer, "Conditional Rewriting Logic as a Unified Model of Concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73–155, 1992.

[40] K. S. Namjoshi and L. D. Zuck, *Witnessing Program Transformations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 304–323. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38856-9_17

[41] J. F. Groote, D. N. Jansen, J. J. A. Keiren, and A. J. Wijs, "An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation," *ACM Trans. Comput. Logic*, vol. 18, no. 2, pp. 13:1–13:34, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3060140

[42] J. R. Büchi, "On a Decision Method in Restricted Second-Order Arithmetic," in *International Congress on Logic, Methodology, and Philosophy of Science*. Stanford University Press, 1962, pp. 1–11.

[43] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*. Cambridge, MA, USA: MIT Press, 1987.

[44] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931074 pp. 294–305.

[45] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '98. London, UK, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646482.691453 pp. 151–166.

[46] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," https://ethereum.github.io/yellowpaper/paper.pdf.

[47] E. Foundation, "The solidity contract-oriented programming language," https://github.com/ethereum/solidity.

[48] E. Foundation, "Vyper: New experimental smart contract programming language," https://github.com/ethereum/vyper.

[49] G. Rosu, "Erc20-k: Formal executable specification of erc20," https://github.com/runtimeverification/erc20-semantics, 2017.

[50] Runtime Verification, Inc., "ERC20-EVM," https://github.com/runtimeverification/verified-smart-contracts/tree/master/erc20.

[51] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3236024.3264591 pp. 912–915.

[52] D. Bogdanas and G. Rosu, "K-Java: A complete semantics of Java," in *POPL*. ACM, 2015, pp. 445–456.

[53] A. Møller and M. I. Schwartzbach, "The pointer assertion logic engine," in *PLDI*, 2001, pp. 221–231.

[54] M. Research, "VCC: A verifier for concurrent C," http://vcc.codeplex.com, accessed: March 25, 2019.

[55] Runtime Verification, Inc., "Verified smart contracts," https://github.com/runtimeverification/verified-smart-contracts/, 2018.

[56] Y. V. Matiyasevich, *Hilbert's Tenth Problem*. MIT Press, 1993.

[57] Runtime Verification, Inc., "Bihu smart contract formal verification," https://github.com/runtimeverification/verified-smart-contracts/tree/master/bihu.

[58] Runtime Verification, Inc., "edsl: Domain-specific language for evm specifications," https://github.com/runtimeverification/verified-smart-contracts/blob/master/resources/edsl.md.

[59] Mean.io, "MEAN: A Fullstack Javascript Framework," http://mean.io/, 2014, accessed: March 25, 2019.

[60] A. Zakai, "Emscripten: An LLVM-to-JavaScript Compiler," in *SPLASH*. ACM, 2011, pp. 301–312.

[61] D. Herman, L. Wagner, and A. Zakai, "asm.js," http://asmjs.org, 2014, accessed: March 25, 2019.

[62] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[63] M. Samuel, "Properties of Interpreters or the Browser Environment that allow Privilege Escalation," https://code.google.com/p/google-caja/wiki/AttackVectors, 2009, accessed: March 25, 2019.

[64] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully Abstract Compilation to JavaScript," in *POPL*. ACM, 2013, pp. 371–384.

[65] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated Analysis of Security-Critical JavaScript APIs," in *S&P (Oakland)*. IEEE, 2011, pp. 363–378.

[66] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety: Type-based Verification of JavaScript Sandboxing," in *USENIX Security*. USENIX, 2011, pp. 12–12.

[67] S. Guarnieri and B. Livshits, "GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code," in *USENIX Security*. USENIX, 2009, pp. 151–168.

[68] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: Vetting Browser Extensions for Security Vulnerabilities," in *USENIX Security*. USENIX, 2010, pp. 22–22.

[69] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the World Wide Web from Vulnerable JavaScript," in *ISSTA*. ACM, 2011, pp. 177–187.

[70] A. Guha, S. Krishnamurthi, and T. Jim, "Using Static Analysis for Ajax Intrusion Detection," in *WWW*. ACM, 2009, pp. 561–570.

[71] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A Trusted Mechanised JavaScript Specification," in *POPL*. ACM, 2014, pp. 87–100.

[72] D. Bruant, "ECMAScript Bug 56," https://bugs.ecmascript.org/show_bug.cgi?id=56#c3, 2011, accessed: March 25, 2019.

[73] D. Bruant, "Mozilla Bug 641214," https://bugzilla.mozilla.org/show_bug.cgi?id=641214, 2011, accessed: March 25, 2019.

[74] Ecma TC39, "TC39 Meeting Minutes," https://github.com/rwaldron/tc39-notes/blob/master/es6/2014-09/sept-23.md#somehow-we-started-talking-about-test262, 2014, accessed: March 25, 2019.

[75] D. Park, "KJS: A Complete Formal Semantics of JavaScript," https://github.com/kframework/javascript-semantics.

[76] Ecma TC39, "Standard ECMA-262 ECMAScript Language Specification Edition 5.1," June 2011.

[77] Ecma TC39, "ECMAScript Harmony," https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html, 2008, accessed: March 25, 2019.

[78] Ecma TC39, "Draft Specification of ECMA-262 6th Edition," http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts, 2014, accessed: March 25, 2019.

[79] G. Smith, "ECMA-262 Bug 1444," https://bugs.ecmascript.org/show_bug.cgi?id=1444, 2013, accessed: March 25, 2019.

[80] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour, "Bootstrapping a Self-hosted Research Virtual Machine for JavaScript: An Experience Report," in *Proceedings of the 7th Symposium on Dynamic Languages*. ACM, 2011, pp. 61–72.

[81] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi, "A Tested Semantics for Getters, Setters, and Eval in JavaScript," in *Proceedings of the 8th Symposium on Dynamic Languages*. ACM, 2012, pp. 1–16.

[82] Ecma TC39, "Test262: ECMAScript Language Conformance Test Suite," http://test262.ecmascript.org, 2014, accessed: March 25, 2019.

[83] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript," in *Proceedings of the 2012 International Workshop on Foundations of Object-Oriented Languages*. ACM, 2012.

[84] D. Park, "WebKit Bug 138859, 138858; V8 Issue 3704; ECMA-262 Bug 3427, 3426; S5 Issues 55, 57, 59," https://bugs.webkit.org/show_bug.cgi?id=138859, https://bugs.webkit.org/show_bug.cgi?id=138858, https://code.google.com/p/v8/issues/detail?id=3704, https://bugs.ecmascript.org/show_bug.cgi?id=3427, https://bugs.ecmascript.org/show_bug.cgi?id=3426, https://github.com/brownplt/LambdaS5/issues/55, https://github.com/brownplt/LambdaS5/issues/57, https://github.com/brownplt/LambdaS5/issues/59, 2014, accessed: March 25, 2019.

[85] E. Arvidsson, "V8 Issue 2243," https://code.google.com/p/v8/issues/detail?id=2243, 2012, accessed: March 25, 2019.

[86] J. Orendorff, "Mozilla Bug 779682," https://bugzilla.mozilla.org/show_bug.cgi?id=779682, 2012, accessed: March 25, 2019.

[87] M. Samuel, "Attack Vectors: Global Object Poisoning," https://code.google.com/p/google-caja/wiki/GlobalObjectPoisoning, 2009, accessed: March 25, 2019.

[88] S. K. Lahiri and S. Qadeer, "Verifying properties of well-founded linked lists," in *POPL*, 2006, pp. 115–126.

[89] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter, "Specification and verification: the Spec# experience," *Commun. ACM*, vol. 54, no. 6, pp. 81–91, 2011.

[90] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR*, 2010, pp. 348–370.

[91] K. R. M. Leino, P. Müller, and J. Smans, "Deadlock-free channels and locks," in *ESOP*, ser. LNCS, vol. 6012, 2010, pp. 407–426.

[92] J. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *ESOP*, ser. LNCS, vol. 7792, 2013, pp. 125–128.

[93] J. Filliâtre and C. Marché, "The why/krakatoa/caduceus platform for deductive program verification," in *CAV*, ser. LNCS, vol. 4590, 2007, pp. 173–177.

[94] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for C and Java," in *Proceedings of the Third International Conference on NASA Formal Methods (NFM'11)*, ser. LNCS, vol. 6617, 2011, pp. 41–55.

[95] D. Distefano and M. J. Parkinson, "jStar: Towards practical verification for Java," in *OOPSLA*. ACM, 2008, pp. 213–226.

[96] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*. ACM, 2012, pp. 405–416.

[97] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Semantics-based generation of verification conditions by program specialization," in *PPDP*. ACM, 2015, pp. 91–102.

[98] S. Bucur, J. Kinder, and G. Candea, "Prototyping symbolic execution engines for interpreted languages," in *ASPLOS*. ACM, 2014, pp. 239–254.

[99] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen, *The RAISE Development Method*. Prentice Hall, 1995.

[100] T. Nipkow, "Winskel is (almost) right: Towards a mechanized semantics textbook," *Formal Aspects of Computing*, vol. 10, pp. 171–186, 1998.

[101] H. Liu and J. S. Moore, "Java program verification via a JVM deep embedding in ACL2," in *TPHOLs*, ser. LNCS, vol. 3223, 2004, pp. 184–200.

[102] B. Jacobs, "Weakest pre-condition reasoning for Java programs with JML annotations," *J. Logic and Algebraic Programming*, vol. 58, no. 1-2, pp. 61–88, 2004.

[103] A. W. Appel, "Verified software toolchain," in *ESOP*, ser. LNCS, vol. 6602, 2011, pp. 1–17.

[104] A. Chlipala, "Mostly-automated verification of low-level programs in computational separation logic," in *PLDI*, 2011, pp. 234–245.

[105] D. Harel, D. Kozen, and J. Tiuryn, "Dynamic logic," in *Handbook of Philosophical Logic*, 1984, pp. 497–604.

[106] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, 2007.

[107] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 2016, 2016.

[108] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, ser. NDSS 2018, 2018.

[109] P. Tsankov, A. M. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," *CoRR*, vol. abs/1806.01143, 2018.

[110] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *CoRR*, vol. abs/1802.06038, 2018.

[111] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS 2016, 2016.

[112] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Proceedings of the 7th International Conference on Principles of Security and Trust*, ser. POST 2018, 2018.

[113] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM International Conference on Certified Programs and Proofs*, ser. CPP 2018, 2018.

[114] K. S. Namjoshi, *A simple characterization of stuttering bisimulation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 284–296. [Online]. Available: http://dx.doi.org/10.1007/BFb0058037

[115] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis, "The marriage of bisimulations and kripke logical relations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103666 pp. 59–72.

[116] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, *A Language-Independent Proof System for Mutual Program Equivalence*. Cham: Springer International Publishing, 2014, pp. 75–90. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11737-9_6

[117] D. Herman and C. Flanagan, "Status Report: Specifying Javascript with ML," in *Proceedings of the 2007 Workshop on Workshop on ML*. ACM, 2007, pp. 47–52.

[118] S. Maffeis, J. C. Mitchell, and A. Taly, "An Operational Semantics for JavaScript," in *APLAS*, vol. 5356. LNCS, 2008, pp. 307–325.

[119] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The Essence of Javascript," in *ECOOP*, vol. 6183. LNCS, 2010, pp. 126–150.

[120] P. A. Gardner, S. Maffeis, and G. D. Smith, "Towards a Program Logic for JavaScript," in *POPL*. ACM, 2012, pp. 31–44.

[121] M. Nordio, C. Calcagno, and C. A. Furia, "Javanni: A Verifier for JavaScript," in *Fundamental Approaches to Software Engineering*, vol. 7793. LNCS, 2013, pp. 231–234.

[122] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A Static Analysis Platform for JavaScript," in *FSE*. ACM, 2014, pp. 121–132.

[123] D. Filaretti and S. Maffeis, "An Executable Formal Semantics of PHP," in *ECOOP*, vol. 8586. LNCS, 2014, pp. 567–592.

[124] D. Guth, "A Formal Semantics of Python 3.3," M.S. thesis, University of Illinois at Urbana-Champaign, July 2013.

[125] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case Reduction for C Compiler Bugs," in *PLDI*. ACM, 2012, pp. 335–346.