Semantics-Based Program Verifiers for All Languages

Andrei Ștefănescu
University of Illinois at Urbana-Champaign, USA
stefane1@illinois.edu

Daejun Park
University of Illinois at Urbana-Champaign, USA
dpark69@illinois.edu

Shijiao Yuwen
University of Illinois at Urbana-Champaign, USA
yuwen2@illinois.edu

Yilong Li
Runtime Verification, Inc., USA
yilong.li@runtimeverification.com

Grigore Roșu
University of Illinois at Urbana-Champaign, USA
grosu@illinois.edu

Abstract

We present a 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 one academic language, KernelC, as well as with three recent semantics of 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 trees. This is the first approach that can turn the operational semantics of real-world languages into correct-by-construction automatic verifiers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs; F.3.1 [*Logics and Meanings Of Programs*]: Specifying and Verifying and Reasoning about Programs—mechanical verification; F.3.2 [*Logics and Meanings Of Programs*]: Semantics of Programming Languages—operational semantics

General Terms Languages, Theory, Verification

Keywords reachability logic, matching logic, $\mathbb{K}$ framework

OOPSLA '16, October 30-November 04 2016, Amsterdam, Netherlands

ACM 978-1-4503-4444-9/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2983990.2984027

1. Introduction

Operational semantics are easy to define and understand, similarly to implementing an interpreter. They require little formal training, scale up well, and, being executable, can be tested. Thus, operational semantics are typically used as trusted reference models for the defined languages. Despite these advantages, they are rarely used directly for program verification, because proofs tend to be low-level, as they work directly with the corresponding transition system. Hoare or dynamic logics allow higher level reasoning at the cost of (re)defining the language as a set of abstract proof rules, which are harder to understand and trust. The state-of-the-art in mechanical program verification is to develop and prove such language-specific proof systems sound w.r.t. a trusted operational semantics [3, 26, 36], but that needs to be done for each language separately and is labor intensive.

Defining multiple semantics for the same language and proving the soundness of one semantics in terms of another are highly uneconomical tasks when real languages are concerned, often taking several man-years to complete. For these reasons, many program verification tools forgo defining an operational or an axiomatic semantics altogether, and instead they implement ad-hoc strongest-postcondition or weakest-precondition generation. For example, tools for C like VCC [11] and Frama-C [21], and for Java like jStar [17] take this approach. Sometimes this is a two step process: first translate the high-level source code to a low-level intermediate verification language (IVL), and then perform the verification condition (VC) generation for the IVL. This leads to some re-usability: implementing a new program verifier for a language reduces to implementing a translator to the IVL, and then reusing the VC generation already implemented for the IVL. For example, VCC translates to Boogie [4] and Frama-C translates to Why3 [21].

However, defining correct language translations is not easy. Consider VCC. The translator consists of 5000 lines of F# [1]

and has to be correct with respect to the 650 page ISO C11 Standard. There is the added difficulty that the translation cannot be easily tested. Due to limitations in the translation to Boogie, VCC both misses behaviors and verifies incorrect programs. Consider the following snippet:

```
1  unsigned x = UINT_MAX;
2  unsigned y = x + 1;
3  _(assert y == 0)
```

VCC fails to verify it, reporting an overflow in line 2. However, according to the C11 standard, the result of operations on unsigned integers does not overflow, it is reduced modulo `UINT_MAX` + 1, and thus the assertion in line 3 holds. Due to this bug in the translation to Boogie, VCC reports a false positive. Consider another snippet:

```
1  int foo(int *p, int x)
2  _(ensures *p == x)
3  _(writes p)
4  { return (*p = x); }
5
6  void main() {
7    int r;
8    foo(&r, 0) == foo(&r, 1);
9    _(assert r == 1)
10 }
```

According to the C11 Standard, this program is well-defined but non-deterministic: the arguments of `==` can evaluate in any order, so `r` could be either 0 or 1 on line 9. We have witnessed both behaviors by using different compilation options of the GCC compiler. However, VCC reports no error for the assertion on line 9. These issues are caused solely by limitations in the translation from C to Boogie.

The purpose of these examples is not to bash VCC, but to illustrate a less glamorous aspects of program verification, namely handling the semantics of real-world languages. VCC is a state-of-the-art program verifier, able to efficiently reason about very complex aspects of C programs, like threads, and has been used to verify software components like the Microsoft Hyper-V hypervisor [29]. In particular, it should have no problem handling the examples above. Moreover, what makes VCC an effective program verifier are its reasoning capabilities (support for modular reasoning about concurrency, axiomatizations of different mathematical domains, integration with SMT solvers, etc) and its conventions for writing the correctness properties, both of which are orthogonal to the tricky language features. Unfortunately, in general, with the current state-of-the-art, the only way to ensure the absence of such false positives and false negatives is to prove the underlying axiomatic semantics, or VC generation, or IVL translation sound with respect to a trusted reference model of the language, typically an operational semantics. This is a very tedious task for real-world languages.

In this work we explore an alternative way to building program verifiers for real-world languages. Specifically, we would like to build sound-by-construction program verifiers directly from the operational semantics and without defining any other semantics, VC generator, or translator. We view

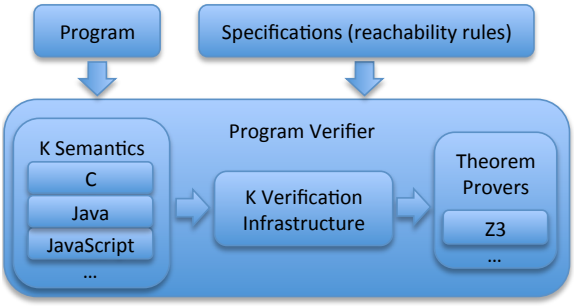


Figure 1: Architecture of Semantic-Based Verification

operational semantics as a mathematical models of programming languages which should exist independently of any program analysis. Thus, we aim to use the semantics unchanged, and do not count the effort of defining the operational semantics towards the total effort of building the program verifiers. We build on recent *theoretical* work [12, 43, 45–47] 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).

But does it really work in practice, with complex real-world languages? To our knowledge, MatchC [47] is the only program verifier based on the language-independent proof system, which is a prototype hand-crafted for a toy language, KernelC. MatchC mixes the language-independent reasoning with the operational semantics of KernelC, e.g., it hard-codes when to perform Case Analysis (for constructs like if), and when to perform heap abstractions folding/unfolding. To answer this question, we build a verification framework that takes existing operational semantics of real-world languages, like the ones of C [18, 25], Java [8], and JavaScript [38], and automatically turns them into correct-by-construction program verifiers, without any language-specific hardcoding.

Figure 1 describes the architecture of our verification framework. We developed it as part of the open-source $\mathbb{K}$ semantic framework [44] (`http://kframework.org`), in which the semantics of the 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 [15]). The program correctness properties are given as reachability rules between matching logic patterns [42]. 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 during development. Then, we evaluated it with recent 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 6.1). 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 6.2). The semantics of C, Java, and JavaScript were developed independently from the verification infrastrucure in the sense that they were developed with the goal of giving a straignt-forward 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.

Our approach has two advantages over the state-of-the-art: (1) provides a way to obtain semantics-based verifiers without a need for multiple semantics, equivalence proofs, or translators; and (2) separates reasoning from language-specific operational details.

Contributions. This paper makes the following contributions:

- A language-independent verification infrastructure, which can be instantiated with a $\mathbb{K}$ semantics to obtain a program verifier for the respective language.
- Program verifiers for C, Java, and JavaScript generated from their existing $\mathbb{K}$ semantics, and an evaluation of the development cost of building these verifiers from the operational semantics.
- Empirical evaluation of these verifiers on challenging heap manipulation programs implementing data-structure.

2. Motivating 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

C

```c
struct node {
   int value;
   struct node *left, *right;
};

struct node* new_node(int v) {
   struct node *node;
   node = (struct node *)
      malloc(sizeof(struct node));
   node->value = v;
   node->left = NULL;
   node->right = NULL;
   return node;
}

struct node* insert(int v, struct node *t) {
   if (t == NULL)
      return new_node(v);
   if (v < t->value)
      t->left = insert(v, t->left);
   else if (v > t->value)
      t->right = insert(v, t->right);
   return t;
}
```

Java

```java
class Node {
   int value;
   Node left, right;

   public Node(int value) {
      this.value = value;
      left = right = null;
   }

   public static Node insert(int v, Node t) {
      if (t == null)
         return new Node(v);
      if (v < t.value)
         t.left = insert(v, t.left);
      else if (v > t.value)
         t.right = insert(v, t.right);
      return t;
   }
}
```

JavaScript

```javascript
function make_node(v) {
   var node = {
      value : v,
      left : null,
      right : null
   };
   return node;
}

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);
   return t;
}
```

Figure 2: Binary search tree code in C, Java, and JavaScript

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

Figure 2 shows 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 3 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 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

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) ∧ −2147483648 ≤ V ∧ V ≤ 2147483647
ensures bst($?T_2$) ∧ tree_keys($?T_2$) = {V} ∪ 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) ∧ −2147483648 ≤ V ∧ V ≤ 2147483647
ensures bst($?T_2$) ∧ tree_keys($?T_2$) = {V} ∪ 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) ∧ ¬isNaN(V)
ensures bst($?T_2$) ∧ tree_keys($?T_2$) = {V} ∪ tree_keys(T_1)

Figure 3: Binary search tree correctness specifications for C, Java, and JavaScript

`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 1 in Section 6.1).

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 [47]. We have not implemented this feature, using instead a general-purpose notation which is faithful to both reachability logic and our implementation.

3. Matching Logic Reachability

Here 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 using the semantics.

3.1 Matching Logic

Matching logic [42] 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 [48]; here we use the latest variant in [42].

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 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:*

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

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

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 $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \rightarrow \mathcal{P}(M_s)$ 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 3. *Given a model M and a map $\rho : Var \rightarrow M$, called an M-valuation, let its extension $\overline{\rho} :$ Pattern $\rightarrow \mathcal{P}(M)$ be inductively defined as follows:*

- $\overline{\rho}(x) = \{\rho(x)\}$*, for all* $x \in Var_s$
- $\overline{\rho}(\sigma(\varphi_{s_1}, ..., \varphi_{s_n})) = \sigma_M(\overline{\rho}(\varphi_1), ...\overline{\rho}(\varphi_n))$
- $\overline{\rho}(\neg\varphi_s) = M_s \setminus \overline{\rho}(\varphi_s)$ *("\" is set difference)*
- $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : Var \rightarrow M,\ \rho' \restriction_{Var \setminus \{x\}} = \rho \restriction_{Var \setminus \{x\}}\}$ *("$\rho\restriction_A$" 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 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 [42] 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., [37]) can be framed as a matching logic theory over a map model [42]. 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 $M \models \varphi$ [42].

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 [42], we can now prove:

$$1 \mapsto 3 * 2 \mapsto 0 * 3 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * 9 \mapsto 0 \;\to\; tree(7, node(9, node(3, leaf, leaf), leaf))$$

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 [18, 25], whose configuration has more than 100 semantic cells like the ones in Figure 3. 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 [42]:

$$\langle\langle 1 \mapsto 3 * 2 \mapsto 0 * 3 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * 9 \mapsto 0 * m \rangle_{\mathsf{mem}}\; c \rangle_{\mathsf{cfg}} \;\to\; \langle\langle tree(7, node(9, node(3, leaf, leaf), leaf)) * m \rangle_{\mathsf{mem}}\; c \rangle_{\mathsf{cfg}}$$

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

3.2 Specifying Reachability

We recall the two types of reachability statements that our proof system in Section 3.3 derives: the one-path reachability rule [46], and the all-path reachability rule [12]. These are pairs of matching logic patterns, in this paper 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, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*,

Definition 5. *[46] A* ***one-path reachability rule*** $\varphi \Rightarrow^{\exists} \varphi'$*, with* φ *and* φ' *patterns (may have free variables). Rule* $\varphi \Rightarrow^{\exists} \varphi'$ *is* ***weakly well-defined*** *iff for any* $\gamma \in \mathcal{T}_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$ *with* $(\gamma,\rho) \models \varphi$*, there exists* $\gamma' \in \mathcal{T}_{Cfg}$ *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*** $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ *on the configuration model:* $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ *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* $(\gamma',\rho) \models \varphi'$*.* $A \Rightarrow^{\mathcal{T}}_{\mathcal{S}}$***-path*** *is a finite sequence* $\gamma_0 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} ... \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_n$ *with* $\gamma_0,...,\gamma_n \in \mathcal{T}_{Cfg}$*.* $A \Rightarrow^{\mathcal{T}}_{\mathcal{S}}$*-path is* ***complete*** *iff it is not a strict prefix of any other* $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$*-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 [19], the CHAM [7], and $\mathbb{K}$ [44]. Several large languages have been given semantics in such styles, including the ones used in this paper: 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 4.1 (and particularly Figure 5) 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 6. *[46] 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 $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path from γ to some γ' such that $(\gamma', \rho) \models \varphi'$, or there is a diverging execution $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \cdots$ from γ.*

We next recall the all-path variant from [12].

Definition 7. *[12] With the notation in Definition 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 $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-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

$$\langle\langle \texttt{x = x + 1; } || \texttt{ x = x + 1;} \rangle_{\text{code}} \langle \texttt{x} \mapsto m \rangle_{\text{state}} \rangle_{\text{cfg}} \Rightarrow^{\forall} \exists n \, (\langle\langle\rangle_{\text{code}} \langle \texttt{x} \mapsto n \rangle_{\text{state}} \rangle_{\text{cfg}} \wedge (n = m +_{Int} 1 \vee n = m +_{Int} 2)$$

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

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

See Section 4.3 (and particularly Figure 6c) for a full reachability logic proof of this rule.

3.3 Reachability Proof System

Figure 4 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 [46] with the all-path one in [12], taking advantage of recent developments in matching logic in [42]. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Theorem 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 "$\mathcal{S}, \mathcal{A} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$", where $\mathcal{A}$ and C are sets of reachability rules. The rules in $\mathcal{A}$ are called *axioms* and rules in C are called *circularities*. If $\mathcal{A}$ or C does not appear in a sequent, it is empty: $\mathcal{S} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$ is shorthand for $\mathcal{S}, \emptyset \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$, 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 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 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 C become (coinductively) valid and can be used in derivations by AXIOM. During the proof, circularities can be added to C via CIRCULARITY, flushed into $\mathcal{A}$ by TRANSITIVITY, and used via AXIOM. The semantics of sequent $\mathcal{S}, \mathcal{A} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$ (read "$\mathcal{S}$ with axioms $\mathcal{A}$ and circularities C proves $\varphi \Rightarrow^{Q} \varphi'$") is: $\varphi \Rightarrow^{Q} \varphi'$ holds if the rules in $\mathcal{A}$ hold and those in C hold after taking at least one step from φ in the transition system $(\Rightarrow^{\mathcal{T}}_{\mathcal{S}}, \mathcal{T})$. Moreover, if $C \neq \emptyset$ 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 $\mathcal{S}, \mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'$ (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

STEP :

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

AXIOM :

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

REFLEXIVITY :

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

TRANSITIVITY :

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

CONSEQUENCE :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{S}, \mathcal{A} \vdash_C \varphi_1' \Rightarrow^{Q} \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^{Q} \varphi_2}$$

CASE ANALYSIS :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^{Q} \varphi \quad \mathcal{S}, \mathcal{A} \vdash_C \varphi_2 \Rightarrow^{Q} \varphi}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi}$$

ABSTRACTION :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{Q} \varphi' \quad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{S}, \mathcal{A} \vdash_C \exists \mathsf{X}\, \varphi \Rightarrow^{Q} \varphi'}$$

CIRCULARITY :

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

Figure 4: 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 $Q \in \{\forall, \exists\}$.

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 $\rho : \mathit{Var} \rightarrow \mathcal{T}$ 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 C empty to ensure that rules derived with non-empty C take at least one step. TRANSITIVITY enables the circularities as axioms for the second premise, since if 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 1. *The proof system in Figure 4 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 [12], and for the one-path case in [47]. When considering the completeness of program verification logics, notice that if the logic for 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 4 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.

4. Reachability Logic vs. Hoare Logic

Here we briefly compare reachability logic (Section 3.3) 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.

4.1 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 5 shows a reduction-style executable semantics of the needed language fragment; with the notation explained in the caption of Figure 5, 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:

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

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

Int ::= Arbitrarily large integers
Var ::= Arbitrarily variables (identifiers)
Exp ::= Int | $Exp + Exp$ | $Exp - Exp$ | $--Var$
$Stmt$ ::= `{}` | `{`$Stmt$`}` | Var `=` Exp`;` | $Stmt\ Stmt$
| `if(`Exp`)` $Stmt$ `else` $Stmt$
| `while(`Exp`)` $Stmt$
C ::= $\square$ | $C\ Stmt$ | Var `=` C`;` | $C + Exp$ | $Exp + C$
| `if(`C`)` $Stmt$ `else` $Stmt$

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

Figure 5: Reduction semantics of a simple imperative language with auto-decrement. Configurations have the form $\langle\langle ... \rangle_{\text{code}}\ \langle ... \rangle_{\text{state}}\rangle_{\text{cfg}}$. 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. $Context[t_1 \Rightarrow t'_1, ..., t_n \Rightarrow t'_n]$ is shorthand for $Context[t_1, ..., t_n] \Rightarrow Context[t'_1, ..., t'_n]$, and $t \Rightarrow t'$ is shorthand for $\langle\langle C[t \Rightarrow t']\rangle_{\text{code}}\ \langle\sigma\rangle_{\text{state}}\rangle_{\text{cfg}}$.

4.2 Hoare Logic Proof

The Hoare logic precondition ψ_{pre} is `n` $=_{Int}$ $n \wedge n \geq_{Int} 1$, and the postcondition ψ_{post} is `n` $=_{Int} 0 \wedge$ `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 value of `s` in the post-condition (the loop changes the value of `n`). A typical (over-)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:

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

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 an inherently language-specific operation.

For example, VCC [11] expands the loop above into one having more than a dozen statements in its translation to Boogie [4]. To keep it human readable, we manually modify `SUM` in a minimal (but adhoc) way to the equivalent `SUM'` below, which can be verified using conventional Hoare logic:

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

Recall the remaining Hoare logic rules required for this proof:

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

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

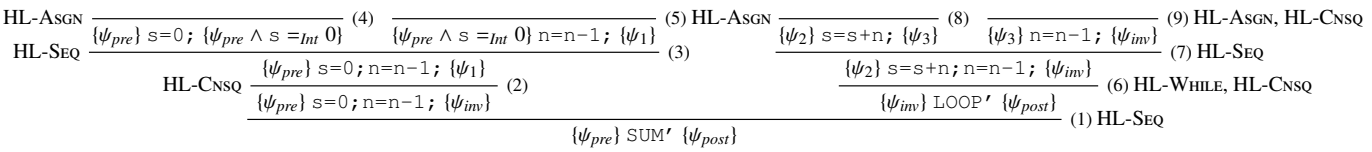
$$\frac{\models \psi'_1 \rightarrow \psi_1 \qquad \{\psi_1\}\ \ S\ \ \{\psi_2\} \qquad \models \psi_2 \rightarrow \psi'_2}{\{\psi'_1\}\ \ S\ \ \{\psi'_2\}} \quad \text{(HL-CNSQ)}$$

The proof can be derived as shown in Figure 6a. 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 $\psi_{inv} \wedge$ `n` $\neq_{Int} 0$. The rest follows by two applications of HL-ASGN at (8,9), followed by an HL-SEQ which concludes the proof.

4.3 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 4.2 because the Hoare logic proof rule for `while` assumes there are no side-effects in the condition.

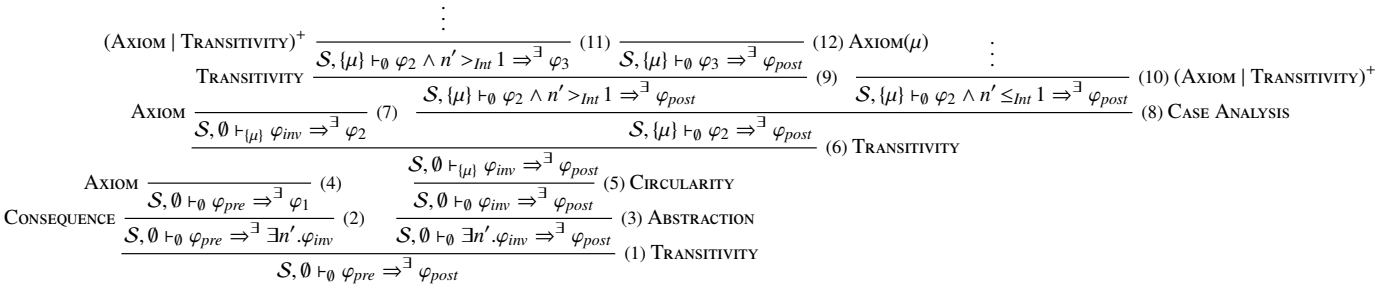
Let $\mathcal{S}$ be the reachability logic system in Figure 5, where each rule is regarded as a one-path rule as explained in Section 4.1. The reachability logic rule stating the correctness of `SUM` is $\varphi_{pre} \Rightarrow^{\exists} \varphi_{post}$, which can be derived as shown in Figure 6c. 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



(a) Hoare logic proof of `SUM′`

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

(b) Notations for Hoare logic proof



(c) Reachability logic proof of `SUM`

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

(d) Notations for reachability logic proof

Figure 6: Hoare logic and 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 4.2 and 4.3.

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 $\varphi_2 \wedge (n' \leq_{Int} 1 \vee n' >_{Int} 1)$).

4.4 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 [25] 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.

5. Implementation

We discuss our novel implementation of the $\mathbb{K}$ verification infrastructure, depicted in Figure 1, based on the language-independent proof system in Figure 4. Our framework takes an operational semantics defined in $\mathbb{K}$ [44] 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 3.2, 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 evaluation order). Section 2 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 C of rules between conjunctive patterns, it uses the following algorithm to derive

$$\mathcal{S}, \emptyset \vdash_C \varphi \Rightarrow^{\forall} \varphi'$$

for each $\varphi \Rightarrow^{\forall} \varphi' \in C$:

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

where *successors*(φ) returns, as a set, the disjunction of conjunctive patterns representing the one-step successors of φ (see Section 5.1), σ is a substitution, and *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 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 4. 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 C to $\mathcal{A}$. Consequence and Abstraction simplify a pattern before adding it to Q. We use Circularity on the set 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 C with Axiom in the body of the loop in line 6. Thus, we can conclude that all the rules in C hold.

Our verification infrastructure is implemented in Java, and uses Z3 [15]. It consists of approximately 30,000 non-blank lines of code, and it took about 2.5 man-years to complete.

5.1 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 [18]), but it is unifiable with the left-hand-sides of both rules. We achieve symbolic execution by performing *narrowing* [2] (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 [12], 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 = `true` and b = `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.

5.2 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 3.1, 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.,

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

There is an extensive literature on such recursive definitions, especially in the context of separation logic [33, 35, 41].

We employ two heuristics. The first is similar to natural proofs [33, 41]. 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 [40] to work with data, and using them as additional lemmas for list segments on the left-hand-side of an implication, e.g.,

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

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 2, we use equationally constrained function and predicate symbols (like bst and tree_keys); e.g.,

$$\text{height}(\text{node}(\_, t_l, t_r)) = 1 + \max(\text{height}(t_l), \text{height}(t_r))$$
$$\text{height}(\text{leaf}) = 0$$
$$\text{height}(\_) \geq 0 = \text{true}$$

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 5.3).

5.3 Integration with Z3

We use Z3 [15] 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 5.2, 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 [14]. 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 [16]. 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.

5.4 Example

Let us discuss how this algorithm derives the reachability logic proof of the correctness of the `SUM` program in Figure 6c. Note that the algorithm derives the all-path version of the rule instead of the one-path version derived in Figure 6c. The two rules specify the same property, since `IMP` is deterministic.

In this case, 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 6c). 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 6c), 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 6c).

Next, we run the algorithm on $\varphi_{inv} \Rightarrow^{\forall} \varphi_{post}$ (corresponding to finding the sub-proof tree rooted at (5) in Figure 6c). Like before, we compute $successors(\varphi_{inv})$ in line 1, which returns φ_2 (corresponding to (7) in Figure 6c). 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 $successors(\varphi_2)$, which is

$$\langle\langle \texttt{if(}n' -_{Int} 1\texttt{)\{s = s + n; LOOP\} else \{\}}\rangle_{\text{code}}$$
$$\langle \texttt{n} \mapsto n',\ \texttt{s} \mapsto \Sigma_{n'}^{n -_{Int} 1}\rangle_{\text{state}}\rangle_{\text{cfg}} \wedge n' \geq_{Int} 1$$

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 6c):

$$\langle\langle \texttt{\{s = s + n; LOOP\}}\rangle_{\text{code}}$$
$$\langle \texttt{n} \mapsto n',\ \texttt{s} \mapsto \Sigma_{n'}^{n -_{Int} 1}\rangle_{\text{state}}\rangle_{\text{cfg}} \wedge n' \geq_{Int} 1 \wedge n' -_{Int} 1 \neq 0$$

and

$$\langle\langle \texttt{\{\}}\rangle_{\text{code}}$$
$$\langle \texttt{n} \mapsto n',\ \texttt{s} \mapsto \Sigma_{n'}^{n -_{Int} 1}\rangle_{\text{state}}\rangle_{\text{cfg}} \wedge n' \geq_{Int} 1 \wedge n' -_{Int} 1 = 0$$

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 6c), at which point we go through lines 6-7 (corresponding to (12) in Figure 6c). 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.

6. 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 simple toy 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 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 [18, 25], Java 1.4 [8], and JavaScript 5.1 [38] to obtain corresponding program verifiers. We evaluated these verifiers by proving the correctness of the same programs in Table 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 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.).

6.1 Verification Experiments

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 [34, 35, 39, 47]. 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 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 operations 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 [39], 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 1 suggests that our approach is practical and competitive with the state-of-the-art on such data-structures.

6.2 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 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 2), and of making changes to the semantics. Most of changes to the semantics are bug fixes (see Section 6.3), but some are performance improvements or simplifications. The development

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 1: Summary of verification experiments: 'Execution' shows time (seconds) and number of operational semantic steps for symbolic execution (Section 5.1); 'Reasoning' shows time (seconds) and number of Z3 queries for reasoning (Section 5.2 & 5.3).

	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 2: The development costs

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 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# [1]. We believe that writing such a translator takes considerably more effort than we reported for our approach in Table 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 (see Section 6.3), 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 [38].

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 5.2 are shared by all sorting-related programs in all languages).

6.3 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 [8, 18, 25, 38].

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 STEP in Figure 4). 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.

We proposed fixes for the bugs we found and the semantics' authors accepted them. This indicates the existing methodology to validate semantics needs improvement.

7. Related Work

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 IVL and do verification at the IVL level. This results in some re-usability, 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 [4] is a popular IVL integrated with Z3. There are several verifiers built on top of Boogie, including VCC [11], HAVOC [28], Spec# [5], Dafny [30], and Chalice [31]. VCDrayd [39] is a separation logic based verifier built on top of VCC. Why3 [21] is another IVL, also integrated with SMT solvers (and other provers). Tools built on top of Why3 include Frama-C [21] and Krakatoa [20]. There are many other practical VC generation based tools (with or without an IVL), including Verifast [27] and jStar [17]. 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 [23]. A semantics based-approach to translation to Horn clauses for a fragment of C is presented in [13], 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 [9], 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 [3, 22, 26, 32, 36], 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 [10] 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 [24] adds modal operators to FOL to embed program fragments within specifications, but still requires language-specific proof rules (e.g., invariant rules). KeY [6] 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.

Operational semantics-based verification. A first version of a language-independent proof system for reachability is given in [46], and [45] shows a mechanical translation of Hoare logic proof derivations for IMP to it. The CIRCULARITY proof rule was introduced in [47]. Support for operational semantics using conditional rules is introduced in [43], and support for non-determinism in [12]. These previous results are mostly theoretical, with MATCHC a prototype hand-crafted for KERNELC mixing language-independent reasoning with the operational semantics of KERNELC.

8. Conclusion, Limitations, Future Work

This paper introduces a language-independent verification infrastructure that takes as input an operational semantics and automatically turns it into a correct-by-construction program verifier. The framework is instantiated with the semantics of C, JAVA, and JAVASCRIPT, which were developed independently. The generated verifiers successfully check the functional correctness of challenging programs that implement the same algorithms in all three languages.

The language-independent verification approach presented in this paper comes with several limitations. (1) Performance of symbolic execution depends on the granularity of the semantics. In our evaluation, the more granular semantics of JAVASCRIPT is twice as slow as that of JAVA. (2) Specifications are reachability rules between program configurations, which can be verbose. The reachability rules could be generated from in-code annotations (pre/post conditions, loop invariants, class invariants, etc), the same way that Hoare triples can be generated from in-code annotations. However, our tool does not support this yet. (3) Non-determinism is handled by exhaustive interleaving. This works for the non-deterministic evaluation of C expressions, but is currently infeasible for threads. (4) The heap abstractions currently need to be defined for each language separately, leading to boilerplate code.

Currently, our trusted code base is the operational semantics of the language, the K verification infrastructure, and Z3. The proof system has been formalized in Coq previously. There is ongoing work on implementing a Coq backend for $\mathbb{K}$ (support for embedding a $\mathbb{K}$ definition in Coq). Once completed, it would be easy to make the verification infrastructure generate a Coq proof certificate containing the proof steps used, and thus reduce our trusted code base to only the language definition (given as a $\mathbb{K}$ semantics).

Several future directions look interesting. In this work we have focused on data-structure verification. We plan to look at a larger, real-world code base next. There are no conceptual issues caused by our language-independent proof system that need addressing. However, there are several areas where we plan to make improvements. First, we will add support for compact and intuitive in-code annotations (similar to VCC and other program verifiers). Further, we will look into using invariant inference techniques to reduce the annotation burden on the user. Finally, we will connect our infrastructure with the Coq proof assistant in order to allow the user to interactively prove the formulae that our verification infrastructure cannot prove fully automatically. Also, all the programs we verified are single-threaded. We would like to extend our framework to support modular reasoning about multi-threaded programs.

Acknowledgments

We thank Brandon Moore, Cosmin Rădoi, and all the members of the $\mathbb{K}$ team (`http://k-framework.org`) for the many insightful discussions about the work presented in this paper. We also thank the anonymous reviewers for their valuable comments on previous versions of this paper. The work in this paper was supported in part by the NSF grants CCF-1218605, CCF-1318191, and CCF-1421575, by the DARPA HACMS grant FA8750-12-C-0284, and by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2015-2016.

References

[1] VCC: A verifier for concurrent C. `http://vcc.codeplex.com`. Accessed: October 5, 2016.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[3] A. W. Appel. Verified software toolchain. In *ESOP*, volume 6602 of *LNCS*, pages 1–17, 2011.

[4] 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)*, volume 4111 of *LNCS*, pages 364–387, 2006.

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[6] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, 2007.

[7] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[8] D. Bogdanas and G. Rosu. K-Java: A complete semantics of Java. In *POPL*, pages 445–456. ACM, 2015.

[9] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS*, pages 239–254. ACM, 2014.

[10] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.

[11] 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*, volume 5674 of *LNCS*, pages 23–42, 2009.

[12] A. Ştefănescu, S. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *RTA*, volume 8560 of *LNCS*, pages 425–440, July 2014.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *PPDP*, pages 91–102. ACM, 2015.

[14] L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, volume 4603 of *LNCS*, pages 183–198, 2007.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[16] L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE, 2009.

[17] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.

[18] C. Ellison and G. Roșu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.

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

[20] J. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

[21] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128, 2013.

[22] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. Prentice Hall, 1995.

[23] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.

[24] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.

[25] C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. In *PLDI*, pages 336–345. ACM, 2015.

[26] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *J. Logic and Algebraic Programming*, 58(1-2):61–88, 2004.

[27] 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)*, volume 6617 of *LNCS*, pages 41–55, 2011.

[28] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.

[29] D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with VCC. In *FM*, volume 5850 of *LNCS*, pages 806–809, 2009.

[30] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.

[31] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426, 2010.

[32] H. Liu and J. S. Moore. Java program verification via a JVM deep embedding in ACL2. In *TPHOLs*, volume 3223 of *LNCS*, pages 184–200, 2004.

[33] P. Madhusudan, X. Qiu, and A. Ştefănescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136. ACM, 2012.

[34] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.

[35] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, volume 4349 of *LNCS*, pages 251–266, 2007.

[36] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[37] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.

[38] D. Park, A. Stefanescu, and G. Rosu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356. ACM, 2015.

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

[40] J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011.

[41] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242. ACM, 2013.

[42] G. Roşu. Matching logic — extended abstract. In *RTA*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[43] G. Roşu, A. Ştefănescu, S. Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS*, pages 358–367. IEEE, 2013.

[44] G. Roșu and T.-F. Șerbănuță. An overview of the K semantic framework. *J. Logic and Algebraic Programming*, 79(6):397–434, 2010.

[45] G. Roșu and A. Ștefănescu. From Hoare logic to matching logic reachability. In *FM*, volume 7436 of *LNCS*, pages 387–402, 2012.

[46] G. Roșu and A. Ștefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP*, volume 7392 of *LNCS*, pages 351–363, 2012.

[47] G. Roșu and A. Ștefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.

[48] G. Roșu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, volume 6486 of *LNCS*, pages 142–162, 2010.