ESVERIFY: Verifying Dynamically-Typed Higher-Order Functional Programs by SMT Solving

Christopher Schuster
University of California, Santa Cruz
cschuste@ucsc.edu

Sohum Banerjea
University of California, Santa Cruz
sobanerj@ucsc.edu

Cormac Flanagan
University of California, Santa Cruz
cormac@ucsc.edu

1 INTRODUCTION

The goal of program verification is to statically check correctness properties of programs with annotations such as assertions and pre- and postconditions. Recent advances in SMT solving make it applicable to a wide range of domains, including program verification. In this paper, we describe ESVERIFY, a program verifier for JavaScript based on SMT solving, supporting functional correctness properties comparable to languages with refinement and dependent function types. ESVERIFY supports both higher-order functions and dynamically-typed idioms, enabling verification of programs that static type systems usually do not support. To verify these programs, we represent functions as universal quantifiers in the SMT logic and function calls as instantiations of these quantifiers. To ensure that the verification process is decidable and predictable, we describe a bounded quantifier instantiation algorithm that prevents matching loops and avoids ad-hoc instantiation heuristics. We also present a formalism and soundness proof of this verification system in the Lean theorem prover and a prototype implementation.

12

Theory of computation → Pre- and post-conditions; Logic and verification; • Software and its engineering → Language types; Functional languages;

KEYWORDS

program verification, functional programming, SMT solving

ACM Reference Format:

1 INTRODUCTION

The goal of program verification is to statically check programs for properties such as robustness, security and functional correctness across all possible inputs. For example, program verifiers might statically verify that the result of a sorting routine is both sorted and/or a fee. Request permissions from permissions@acm.org.

If a JavaScript program uses features unsupported by ESVERIFY, it will be rejected early; otherwise, verification conditions are generated based on the source code and annotations. Each verification condition is transformed according to a quantifier instantiation algorithm and then checked by an SMT solver. For refuted verification conditions, a dynamic test is synthesized based on the counterexample from the SMT solver and executed to improve error reporting with concrete witnesses. The implementation, including source code1 and a live demo2 are available online.

Additionally, we formally define a JavaScript-inspired, statically verified but dynamically typed language, called λS. Functions in λS are annotated with pre- and postconditions, rendered as logical propositions. These propositions can include operators, refer to variables in scope, denote function results with uninterpreted function calls, and constrain the pre- and postconditions of function values. The verification rules for λS involve checking verification conditions for validity. This checking is performed by an SMT solver augmented with decidable theories for linear integer arithmetic, equality, data types and uninterpreted functions. The key difficulty is that verification conditions can include quantifiers, as function definitions in the source program correspond to universally quantified formulas in verification conditions. Unfortunately, SMT solvers may not always perform the right instantiations, and therefore quantifiers imperil the decidability of the verification process [Ge and de Moura 2009; Reynolds et al. 2013]. We ensure that the verification process remains decidable and predictable, by

1Implementation Source Code: https://github.com/levjj/esverify/
2Online live demo of ESVERIFY: https://esverify.org/try
proposing a bounded quantifier instantiation algorithm such that function calls in the source program act as hints ("triggers") that instantiate these quantifiers. The algorithm only performs a bounded number of trigger-based instantiations and thereby avoids brittle instantiation heuristics and matching loops. Using this decision procedure for verification conditions, we show that verification of \( \lambda^S \) is sound, i.e. verifiable \( \lambda^S \) programs do not get stuck. The proof is formalized in the Lean Theorem Prover and available online\(^3\).

To evaluate the expressiveness of this approach, we also include a brief comparison with static refinement types [Freeman and Pfennig 1991]. While a formalization and proof is beyond the scope of this paper, refined base type and dependent function types can be translated to assertions such that the resulting program is verifiable if the original program is well-typed. This suggests that \texttt{esverify} is at least as expressive as a language with refinement types.

To summarize, the main contributions of this paper are

(1) an approach for verifying dynamically-typed, higher-order JavaScript programs,
(2) a bounded quantifier instantiation algorithm that enables trigger-based instantiations without heuristics or matching loops,
(3) a prototype implementation called \texttt{esverify}, and
(4) a formalization of the verification rules and a proof of soundness in the Lean theorem prover.

The structure of the rest of the paper is as follows: Section 2 illustrates common use cases and relevant features of \texttt{esverify}, Section 3 outlines the verification process and the design of the implementation, Section 4 formally defines quantifier instantiation, as well as the syntax, semantics, verification rules and a soundness theorem for a core language \( \lambda^S \), Section 5 compares the program verification approach to refinement type systems, Section 6 discusses related work, and finally Section 7 concludes the paper.

2 VERIFYING JAVASCRIPT PROGRAMS

Our program verifier, \texttt{esverify}, targets a subset of ECMAScript/-JavaScript. By supporting a dynamically-typed scripting language, \texttt{esverify} is unlike existing verifiers for statically-typed programming languages. We do not aim to support complex and advanced JavaScript features such as prototypical inheritance and metaprogramming, leaving these extensions for future work. We do aim to support both functional as well as object-oriented programming paradigms, as commonly present in JavaScript. This paper, however, focuses on pure functional JavaScript programs with higher-order functions.

2.1 Annotating JavaScript with Assertions

\texttt{esverify} extends JavaScript with syntax for annotating functions with pre- and postconditions, as well as loop invariants and static assertions. We write these with as pseudo function calls with standard JavaScript syntax. While some program verification systems specify these in comments [Flanagan et al. 2002], this approach enables a better integration with existing tooling support such as refactoring tools and syntax highlighters.

---

Christopher Schuster, Sohum Banerjea, and Cormac Flanagan

---

function \texttt{max}(a, b)
\begin{verbatim}
  \texttt{requires} (typeof \(a\) === ‘number’);
  \texttt{requires} (typeof \(b\) === ‘number’);
  \texttt{ensures} (\(a >= b\));
  \texttt{ensures} (\(a >= b\));
  // does not hold
  if (a >= b) {
    return a;
  } else {
    return b; // bug
  }
\end{verbatim}

Listing 1: A JavaScript function \texttt{max} annotated with pre- and postconditions.

function \texttt{sumTo} (n)
\begin{verbatim}
  \texttt{requires} (Number.isInteger(n) && n >= 0);
  \texttt{ensures} (\(\sum_{i=0}^{n} i = \frac{(n+1) \cdot n}{2}\));
  \texttt{let} i = 0;
  \texttt{let} s = 0;
  \texttt{while} (i < n) {
    \texttt{invariant} (Number.isInteger(i) && i <= n);
    \texttt{invariant} (Number.isInteger(s));
    \texttt{invariant} (s = (i + 1) \* i / 2);
    i++;
    s = s + i;
  }
  \texttt{return} s;
\end{verbatim}

Listing 2: A JavaScript function that shows \(\sum_{i=0}^{n} i = \frac{(n+1) \cdot n}{2}\). Loop invariants are not inferred and need to be specified explicitly for all mutable variables in scope.

The assertion language is embedded in JavaScript but does not support all of JavaScript’s semantics. In particular, it is restricted to pure expressions and cannot define new functions.

2.2 \texttt{max}: A Simple Example

Listing 1 shows an example of an annotated JavaScript program. The calls to \texttt{requires} and \texttt{ensures} in lines 2–5 are only used for verification purposes and excluded from evaluation. Instead of introducing custom type annotations, the values of function arguments are constrained using the standard JavaScript \texttt{typeof} operator.

Due to a bug in line 9, the \texttt{max} function does not return the maximum of the arguments if \(b\) is greater than \(a\), violating the postcondition in line 5.

2.3 Explicit Loop Invariants

For programs without loops or recursion, any correctness property can be checked precisely. However, the potential behavior of programs with loops or recursion cannot be determined statically and is “overapproximated” by \texttt{esverify}. Therefore, correct programs may be rejected if the program lacks a sufficiently strong loop invariant or pre- or postcondition, but verified programs are guaranteed to not violate an assertion regardless of the number

---

function inc (x) {
  requires(Number.isInteger(x));
  ensures(y ⇒ Number.isInteger(y) && y > x);
  // implicit: ensures(y ⇒ y == x + 1);
  return x + 1;
}

function twice (f, n) {
  requires(spec((x, y) ⇒ Number.isInteger(y) && y > x));
  requires(Number.isInteger(n));
  ensures(res ⇒ res >= n + 2);
  return f(f(n));
}

const n = 3;
const m = twice(inc, n); // 'inc' satisfies spec
assert(m > 4); // statically verified

Listing 3: The higher-order function twice restricts its function argument f with a maximum precondition and a minimum postcondition. The function inc has its body as implicit postcondition and therefore satisfies this spec.

of iterations or function calls. Listing 2 shows a JavaScript function that computes the sum of the first n natural numbers with a while loop. The loop requires annotated invariants for mutable variables including their types and bounds\(^1\). esverify internally uses standard SMT theorems for integer arithmetic to establish \(\sum_{i=0}^{n-1} i = \frac{(n+1)n}{2}\) at each iteration and thereby verify the whole function.

There is extensive prior work on automatically inferring loop invariants [Furia and Meyer 2010]. However, this topic is orthogonal to the program verification approach presented in this paper.

### 2.4 Higher-order Functions

In order to support function values as arguments and results, esverify introduces a spec construct in pre-, postconditions and assertions. Listing 3 illustrates this syntax in lines 8–10 of the higher-order twice function. The argument f needs to be a function that satisfies the given constraints, and therefore esverify must compare the pre- and postconditions of inc with this spec at the call-site twice(inc, n) in line 16. The explicitly stated postcondition of inc does not satisfy this spec, but esverify implicitly strengthens the postcondition of inc by inlining its function body n + 1. It is important to note that recursive functions are only inlined by one level. This means that recursive functions, similarly to loop invariants, need to be explicitly annotated with adequate pre- and postconditions.

### 2.5 Arrays and Objects

In addition to floating point numbers and integers, esverify also supports other standard JavaScript values such as boolean values, strings, functions, arrays and objects. However, esverify restricts how objects and arrays can be used. In particular, mutation of arrays and objects is not currently supported. Additionally, objects

\(^1\)Here, Number.isInteger(i) ensures that i is an actual integer, while typeof(i) === ‘number’ is also true for floating point numbers.

function f (a) {
  requires(a instanceof Array);
  requires(a.every(e ⇒ e > 3));
  requires(a.length >= 2);
  assert(a[0] > 2); // holds
  assert(a[1] > 4); // does not hold
  assert(a[2] > 0); // does not hold
}

Listing 4: esverify includes basic support for immutable arrays. The elements of an array can be described with every.

can either be immutable dictionaries that map string keys to values or instances of a user-defined class with a fixed set of fields and without inheritance.

The elements of an array can be described with a quantified proposition, corresponding to the standard array method every. This is illustrated in Listing 4.

Despite these restrictions, it is possible to express complex recursive data structures. For example, Listing 5 shows a user-defined linked list class that is parameterized by a predicate. Here, the each field is actually a function that returns true for each element in the linked list. Mapping over the elements of the list with a function f requires that f can be invoked with elements that satisfy this.each and that return values of f satisfy the new predicate newEach. This demonstrates how generic data structures can be used to verify correctness in a similar way to parameterized types. It is important to note that function calls in an assertion context are uninterpreted, so the call newEach(y) in line 19 only refers to the function return value but does not actually invoke the function.

### 2.6 Dynamic Programming Idioms: Promises

JavaScript programs often include functions that have polymorphic calling conventions. A common example is the jQuery library which provides a function \(\$\) whose behavior varies greatly depending on the arguments: given a function argument, the function is scheduled for deferred execution, while other argument types find and return portions of the current webpage.

Even the JavaScript standard itself uses dynamic programming idioms to provide a more convenient programming interface. For example, the latest edition of the ECMAScript standard [ECMA-262 2017] includes Promises [Liskov and Shiria 1988] and specifies a polymorphic Promise.resolve() function. It performs different actions depending on whether it is called with a promise, an arbitrary non-promise object with a method called “then”, or any other arbitrary non-promise object. esverify can accurately express these kinds of specifications in pre- and postconditions as shown in Listing 6, while standard type systems need to resort to code changes (for example, sum types and injections).

### 2.7 Complex Programs: MergeSort

We also demonstrate non-trivial programs such as MergeSort and verify their functional correctness\(^5\). The implementation is purely functional and uses a linked list data type that is defined as a class.

\(^5\)The source code of a MergeSort algorithm in ESVERIFY is available at https://esverify.org/try#msort.
JavaScript has complex polymorphic behavior. This simplifies languages such as Dafny \[1\] and LiquidHaskell \[2\]. These enable such dynamic programming idioms.

Interestingly, about 48 out of a total 99 lines are verification annotations, including invariants, pre- and postconditions and the predicate function \(\text{isSorted}\). \(\text{isSorted}\) is primarily used in specifications, but the implementations of \texttt{merge} and \texttt{sort} also include calls to it. These calls are used as \texttt{triggers}, hints to the underlying SMT solver that do not contribute to the result. In other verified languages such as Dafny [Leino 2013], \(\text{isSorted}\) would correspond to a “ghost function”, but \texttt{esverify} does not currently differentiate between verification-only and regular implementation functions.

### 2.8 JavaScript as Theorem Prover

A simple induction proof over natural numbers can be written as a while loop as previously shown in Listing 2. This idea can be generalized by using the \texttt{spec} construct to reify propositions.

In particular, the postcondition of a function need not only describe its return value; it can also state a proposition such that a value that satisfies the function specification acts as proof of this proposition – analogous to the Curry-Howard isomorphism. Such a function can then be supplied as argument to higher-order functions to build up longer proofs. For an example, Listing 7 includes a proof written in JavaScript showing that any locally increasing integer-ranged function is globally increasing. This example was previously used to illustrate refinement reflection in LiquidHaskell [Vazou et al. 2018].

### 3 IMPLEMENTATION

The sources of the \texttt{esverify} prototype implementation\(^6\) are available online. Because the implementation itself is written in TypeScript, a dialect of JavaScript, it can be used in a browser. In fact, we also implemented a simple browser-based editor with \texttt{esverify} checking. Its source code\(^7\) as well as a live demo\(^8\) are available online. Alternative user interfaces such as integrations with Vim and Emacs also exist.

\(^{6}\)Implementation Source Code: https://github.com/levjj/esverify/

\(^{7}\)Browser-based ESVERIFY Editor Source: https://github.com/levjj/esverify-web/

\(^{8}\)Online live demo of ESVERIFY: https://esverify.org/try

---

function proof_f_mono (f, proof_f_inc, n, m) {
    // f is a function from non-negative int to int
    requires(spec(f),
        (x) => Number.isInteger(x) && x >= 0,
        (x, y) => Number.isInteger(y) && pure());
    // proof_f-inc states that f is increasing
    requires(spec(proof_f_inc, x => Number.isInteger(x) && x >= 0, x => f(x) <= f(x + 1) && pure());
    requires(Number.isInteger(n) && n >= 0);
    requires(Number.isInteger(m) && m >= 0);
    requires(n < m);
    // show that f is increasing for arbitrary n,m
            ensures(f(n) <= f(m));
    ensures(pure()); // no side effects
    proof_f-inc(n); // instantiate proof for n
    if (n + 1 < m) {
        // invoke induction hypothesis (I.H.)
        proof_f_mono(f, proof_f_inc, n + 1, m);
    }
}

function fib (n) {
    requires(Number.isInteger(n) && n >= 0);
    requires(Number.isInteger(n) && n >= 0);
    ensures(res => Number.isInteger(res));
    ensures(pure());
    if (n <= 1) {
        return 1;
    }
    else {
        return fib(n - 1) + fib(n - 2);
    }
}

// A proof that fib is increasing
function proof_f-fib-inc (n) {
    requires(Number.isInteger(n) && n >= 0);
    requires(Number.isInteger(n) && n >= 0);
    ensures(fib(n) <= fib(n + 1));
    ensures(pure());
    fib(n); // unfolds fib at n
    fib(n + 1); // I.H.
    if (n > 0) {
        if (n > 0) {
            fib(n - 1);
            proof_f-fib-inc(n - 1); // I.H.
        }
        if (n > 0) {
            fib(n - 2);
            proof_f-fib-inc(n - 2); // I.H.
        }
    }
}

function proof_f-fib-mono (n, m) {
    requires(Number.isInteger(n) && n >= 0);
    requires(Number.isInteger(m) && m >= 0);
    requires(n < m);
    ensures(fib(n) <= fib(m));
    ensures(pure());
    proof_f-fib-mono(fib, proof_f-fib-inc, n, m);
}

Listing 7: A proof about monotonous integer functions in JavaScript and an instantiation for fib. This example was previously used to illustrate refinement reflection in the statically-typed LiquidHaskell system [Vazou et al. 2018].

The basic verification process and overall design of esverify is depicted in Figure 1.

The first step of the process involves parsing the source code and restricting the input language to a subset of JavaScript that is supported by esverify. Some of these restrictions may be lifted in future versions of esverify, such as support for regular expressions or a variable number of function arguments. However, some JavaScript features would require immense complexity for accurate verification due to their dynamic character and interactions with the rest of the program, such as metaprogramming with eval or new Function(), or have been deprecated in newer versions of strict mode JavaScript such as arguments.callee, this outside of functions or the with statement.

The parser also differentiates between the syntax of expressions and assertions. For example, spec can only be used in assertions while function definitions can only appear in the actual program implementation.

During the second step, scope analysis determines variable scopes and rejects programs with scoping errors or references to unsupported global objects. In addition to user-provided definitions, it includes a whitelist of supported globals such as Array, Math and console. The analysis also takes mutability into account. For example, mutable variables cannot be referenced in class invariants, and the old(x) syntax in a postcondition requires x to be mutable.

The main verification step then traverses the entire source program. At each statement and expression, the current verification context is used to generate verification conditions and augment the context for subsequent statements and expressions. Specifically, the verification context includes a logical proposition that acts as...
precondition, a synthesized unit test with holes, and a set of variables with unknown values. Generated verification conditions combine this context with an assertion, such as a function postcondition. Section 4.4 describes the verification rules in more detail.

The proposition of a verification condition is then transformed with a quantifier instantiation procedure. Quantifiers are instantiated based on triggers and remaining quantifiers erased. The resulting quantifier-free proposition can be checked precisely by SMT solving, ensuring that the verification process remains predicative. However, this approach to quantifier instantiation requires the programmer to provide explicit triggers in terms of function calls. Given this tradeoff, the step is optional and can be skipped.

The next step of the verification process involves checking the verification condition with an SMT solver such as z3 [de Moura and Bjørner 2008] or CVC4 [Barrett and Berezin 2004; Barrett et al. 2011]. If the solver cannot refute the proposition, the verification succeeded. Otherwise, the returned model includes an assignment of free variables that acts as a counterexample.

Given a counterexample and a synthesized unit test with holes, the verification condition can be dynamically tested. This serves two purposes. On the one hand, the test might not be able reproduce an error or assertion violation. This indicates that the static analysis did not accurately model the actual program behavior due to a loop invariant or assertion not being sufficiently strong. In this case, the programmer can use the variable assignment in the counterexample to better understand the shortcomings of the analysis and improve those annotations. On the other hand, the test might lead to an error or assertion violation. In this case, the programmer is presented with both a verification error message as well as a concrete witness that assists in the debugging process similar to existing test generators [Tillmann and de Halleux 2008].

4 FORMALISM

In order to reason about esverify, this section introduces a formal development of λS, a JavaScript-inspired, statically verified but dynamically typed language, and shows that its verification is sound.

The verification rules of λS use verification conditions whose validity is checked according to a custom decision procedure, so this section first defines a language for logical propositions and axiomatizes their validity. Then, the decision procedure including quantifier instantiation is given and proven to be sound. Finally, the syntax and semantics of λS are defined and its verification rules are shown to be sound.

The definitions, axioms and theorems in this section are also formalized in the Lean theorem prover and available online\footnote{Formal Definitions and Proofs in Lean: https://github.com/levji/esverify-theory/}.

4.1 Logical Foundation

Figure 2 formally defines the syntax of propositions, terms, values and environments. Propositions can use terms, connectives \(\neg, \land\) and \(\lor\), symbols \(\text{pre1}, \text{pre2}, \text{pre}, \text{post}\) and universal quantifiers. Here, terms are either values, variables, unary or binary operations or uninterpreted function calls. Finally, values include boolean and integer constants as well as closures which are opaque values that will be explained in section 4.3.

The formal definition of the quantifier instantiation algorithm is given in Section 4.2.

\[
\phi \in \text{Propositions} \quad ::= \quad \tau \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \text{pre1}(\emptyset, \tau) \mid \\
\text{pre2}(\emptyset, \tau, \tau) \mid \text{pre}(\tau, \tau) \mid \text{post}(\tau, \tau) \mid \forall x. \phi
\]

\[
\tau \in \text{Terms} \quad ::= \quad v \mid x \mid \otimes \tau \mid \tau \circ \tau \mid \tau(r)
\]

\[
\otimes \in \text{UnaryOperators} \quad ::= \quad \neg | \text{isInt} | \text{isBool} | \text{isFunc}
\]

\[
\oplus \in \text{BinaryOperators} \quad ::= \quad + | \|- | \times | | | \lor | \land | \lor | = | <
\]

\[
v \in \text{Values} \quad ::= \quad \text{true} | \text{false} | n | (f(x) \text{ req Ens S } [\sigma] \sigma)
\]

\[
\sigma \in \text{Environments} \quad ::= \quad \emptyset | \sigma[x \mapsto v]
\]

\[
n \in \mathbb{N} \quad ::= \quad f, x, y, z \in \text{Variables}
\]

Figure 2: Syntax of logical propositions used in the verifier.

The validity judgement about propositions \(\vdash \phi\) is axiomatized as follows.

\[
\vdash \phi
\]

Axiom 1. \(\vdash \text{true}\).

Validity of conjunctions and disjunctions follows standard inference rules.

Axiom 2. \(\text{If}\) both \(\vdash \phi_1\) and \(\vdash \phi_2\) then \(\vdash \phi_1 \land \phi_2\).

Axiom 3. \(\text{If}\) \(\vdash \phi_1\) then \(\vdash \phi_1 \lor \phi_2\).

Axiom 4. \(\text{If}\) \(\vdash \phi_2\) then \(\vdash \phi_1 \lor \phi_2\).

Axiom 5. \(\text{If}\) \(\vdash \phi_1 \lor \phi_2\) then \(\vdash \phi_1\) or \(\vdash \phi_2\).

For negation, we assume that valid propositions do not include contradictions and we also assume the law of excluded middle.

Axiom 6. \(\vdash \phi \land \neg \phi\) is not true.

Axiom 7. \(\vdash \phi \lor \neg \phi\).

For convenience, we include a notation for implication based on disjunction and negation.

Notation 1 (Implication). \(\phi_1 \Rightarrow \phi_2 \overset{\text{def}}{=} \neg \phi_1 \lor \phi_2\).

Instead of defining evaluation of terms, we include a few axioms that suffice for the remaining formal development. For example, we assume that a term is valid if it evaluates to the value "true" and thereby satisfies the following equality.

Axiom 8. \(\text{If}\) \(\vdash \tau\) then \(\vdash \tau = \text{true}\).

The meaning of unary and binary operators is specified in terms of a partial function \(\delta\), e.g. \(\delta(\times, 2, 3) = 5\).

Axiom 9. \(\text{If}\) \(\delta(\emptyset, v_x) = v\) then \(\vdash v = \emptyset \circ v_x\).

Axiom 10. \(\text{If}\) \(\delta(\emptyset, v_x, v_y) = v\) then \(\vdash v = v_x \oplus v_y\).

The propositions \(\text{pre1}(\emptyset, v_x, v_y)\) and \(\text{pre2}(\emptyset, v_x, v_y)\) can be used to reason about the domain of operators.

Axiom 11. \(\text{If}\) \(\vdash \text{pre1}(\emptyset, v_x, v_y)\) then \(\emptyset, v_x, v_y \in \text{dom}(\delta)\).

Axiom 12. \(\text{If}\) \(\vdash \text{pre2}(\emptyset, v_x, v_y)\) then \(\emptyset, v_x, v_y \in \text{dom}(\delta)\).

The constructs \(\text{pre}(f, x)\) and \(\text{post}(f, x)\) in propositions denote the pre- and postcondition of a function \(f\) when applied to a given argument \(x\) but are left uninterpreted for now.
Definition 1 (Substitution). $\phi[x \mapsto v]$ denotes the proposition in which free occurrences of $x$ in $\phi$ are replaced by $v$.

A universally quantified proposition is true for all values and can be instantiated with any term.

Axiom 13. If $\vdash \phi[x \mapsto v]$ for all values $v$, then $\vdash \forall x. \phi$.

Axiom 14. If $\vdash \forall x. \phi$ then $\vdash \phi[x \mapsto \tau]$ for all terms $\tau$.

A valid proposition is not necessarily closed. In fact, a free variable in a valid proposition is assumed to be implicitly universally quantified.

Axiom 15. If $x$ is free in $\phi$ and $\vdash \phi$ then $\vdash \forall x. \phi$.

An environment $\sigma$ mapping variables to values can be used as a substitution for terms and propositions.

Definition 2 (Lookup). $\sigma(x)$ looks up $x$ is the environment $\sigma$.

Definition 3 (Substitution with Environment). $\sigma(\tau)$ and $\sigma(\phi)$ substitute free variables in $\tau$ and $\phi$ with values according to $\sigma$.

An environment $\sigma$ is a model for a proposition if the substituted proposition is valid. Note that this definition of models is unconventional but it facilitates our subsequent formal development.

Notation 2 (Model). $\sigma \models \phi$ $\iff$ $\sigma(\phi)$.

It is important to note that the validity judgement may not be decidable for all propositions due to the use of quantifiers, so in addition to this (undecidable) validity judgement, we also introduce a notion of satisfiability by an SMT solver.

Definition 4 (Satisfiability). $\text{Sat}(\phi)$ denotes that the SMT solver found a model that satisfies $\phi$.

Theorem 1. If $\phi$ is quantifier-free, then $\text{Sat}(\phi)$ terminates and $\text{Sat}(\phi)$ iff $\sigma \models \phi$ for some model $\sigma$.

Proof. SMT solving is not decidable for arbitrary propositions but the QF-UFLIA fragment of quantifier-free formulas with equality, linear integer arithmetic and uninterpreted function is known to be decidable [Christ et al. 2012; Nelson and Oppen 1979].

4.2 Quantifier Instantiation Algorithm and Decision Procedure

As described in the previous sections, verification of $\lambda^S$ involves checking the validity of verification conditions that include quantifiers. Quantifier instantiation in SMT solvers is an active research topic [Ge and de Moura 2009; Reynolds et al. 2013] and often requires heuristics or explicit matching triggers. However, heuristics can cause unpredictable results and trigger-based instantiation might lead to matching loops. This section describes a bounded quantifier instantiation algorithm that avoids matching loops and brittle heuristics, thus providing us with a predictable decision procedure for verification conditions.

The syntax of verification conditions (VCs) is shown in Figure 3. VCs used in verification rules are similar to propositions introduced in the previous section. However, universal quantifiers have explicit matching patterns to indicate that instantiation requires a trigger. Accordingly, the construct $\text{call}(x)$ acts as an instantiation trigger that does not otherwise affect validity of propositions, i.e. $\text{call}(x)$ can always assumed to be true. Intuitively, $\text{call}(x)$ represents a function call or an asserted function specification while $\forall x.\text{call}(x)$ ⇒ $P$ corresponds to a function definition or an assumed function specification.

The complete decision procedure for VCs including quantifier instantiation is shown in Figure 4.

To make the definition more concise, we first define contexts $P^+[\sigma]$ and $P^-[\sigma]$ for a VC with positive and negative polarity regarding negation. Using this definition, the sets of call triggers in positive and negative positions can be defined as triggers for which there exists a context of the right polarity.

The procedure $\text{instantiateOnce}^-$ performs one round of trigger-based instantiation, within which each universal quantifier with negative polarity is instantiated with all triggers in negative positions. It is important to note that the matching pattern $\text{call}(y)$ now becomes available to instantiate other quantifiers. The lifting is repeated until no more such quantifiers can be found.

The procedure $\text{instantiateOnce}^+$ performs one round of trigger-based instantiation, within which each universal quantifier with positive polarity is instantiated with all triggers in negative position. All such instantiations are conjoined with the original quantifier.

Both lifting and instantiations are repeated for $n$ rounds by the recursive $\text{instantiate}^+$ procedure. During the final step, $\text{erase}^-$ removes all remaining triggers and quantifiers in negative positions.

The overall decision procedure $\langle P \rangle$ performs $n$ rounds of instantiations where $n$ is the maximum level of quantifier nesting. The original VC $P$ is valid if the resulting proposition cannot be refuted by SMT solving.

VCs $P$ can syntactically include both existential and universal quantifiers in both positive and negative positions. However, we can show that VCs generated by the quantifier have existential quantifiers only in negative positions.

Theorem 2 (Decision Procedure Termination). If $P$ does not contain existential quantifiers in negative positions, the decision procedure $\langle P \rangle$ terminates.

Proof. The $\text{lift}^+$ function eliminates a quantifier during each recursive call and therefore terminates when there are no more matching quantifiers in the formula. $\text{instantiateOnce}^-$ and $\text{erase}^-$ are both non-recursive and trivially terminate. Since the maximum level of nesting is finite, $\langle P \rangle$ performs only a finite number of instantiations. With existential quantifiers only in negative positions, the erased and lifted result is quantifier-free, so according to Lemma 1, the final SMT solving step terminates and so does the whole decision procedure.
\[ P^+[\alpha] := o \mid \neg P^-[\alpha] \mid P^+[\alpha] \land P \mid P \land P^+[\alpha] \mid P^+[\alpha] \lor P \mid P \lor P^+[\alpha] \]
\[ P^-[\alpha] := \neg P^-[\alpha] \mid P^-[\alpha] \land P \mid P \land P^-[\alpha] \mid P^-[\alpha] \lor P \mid P \lor P^-[\alpha] \]

\[ \text{lift}^+(P) \overset{\text{def}}{=} \begin{cases} \text{match } P \text{ with} \\ P^+[\forall x. \text{call}(x)] \Rightarrow P' \rightarrow \text{lift}^+(P^+[\text{call}(y) \Rightarrow P'[x \mapsto y])] & (y \text{ fresh}) \\ P^-[\exists x. P^+] \rightarrow \text{lift}^+(P^-[P'[x \mapsto y]]) & (y \text{ fresh}) \\ \text{otherwise} \rightarrow P \\ \end{cases} \]

\[ \text{instantiate}_\text{Once}^-(P) \overset{\text{def}}{=} \begin{cases} \text{let } P \text{ with} \\ P^+[\forall x. \text{call}(x)] \Rightarrow P' \rightarrow P^- \left[ (\forall x. \text{call}(x)) \Rightarrow P' \land \bigwedge_{\text{call}(\tau) \in \text{calls}(P)} P'[x \mapsto \tau] \right] \\ \end{cases} \]

\[ \text{erase}^-(P) \overset{\text{def}}{=} \begin{cases} P^- \left[ (\forall x. \text{call}(x)) \Rightarrow P'' \right] \rightarrow P^- \left[ \text{true} \land P^+[\text{call}(\tau)] \Rightarrow \text{true} \land P^+[\text{call}(\tau)] \Rightarrow P^- \left[ \text{true} \right] \right] \\ \end{cases} \]

\[ \langle P \rangle \overset{\text{def}}{=} \text{let } n = \text{maximum level of quantifier nesting of } P \text{ in } \neg \text{Sat} \left( \neg \text{instantiate}_\text{Once}^-(P, n) \right) \]

Figure 4: The decision procedure lifts, instantiates and finally eliminates quantifiers. The number of iterations is bounded by the maximum level of quantifier nesting.

\[ e \in \text{Expressions} := \]
\[ \text{let } x = \text{true} \in e \mid \text{let } x = \text{false} \in e \mid \text{let } x = n \in e \mid \]
\[ \text{let } f(x) \text{ req } R \text{ ens } S = e \in e \mid \text{let } y = \theta x \in e \mid \]
\[ \text{let } z = x \oplus y \in e \mid \text{let } y = f(x) \text{ in } e \mid \text{if } (x) e \text{ else } e \mid \text{return } x \]
\[ R, S \in \text{Specs} := \tau \mid \neg R \mid R \land R \mid R \lor R \mid \text{spec } \tau(x) \text{ req } R \text{ ens } S \]
\[ \kappa \in \text{Stacks} := (\sigma, e) \mid \kappa \cdot (\sigma, \text{let } y = f(x) \in e) \]

Figure 5: Syntax of \( \lambda^S \) programs. Function definitions have pre- and postconditions written as simple logical propositions with the \textit{spec} syntax for higher-order functions.

Definition 5 (Proposition Translation). \( \text{prop}(P) \) denotes a proposition such that triggers and matching patterns in \( P \) are removed and existential quantifiers \( \exists x. P \) translated to \( \neg \forall x. \neg \text{prop}(P) \).

Theorem 3 (Quantifier Instantiation Soundness). If \( P \) has no existential quantifiers in negative positions, then \( \langle P \rangle \) implies \( \text{prop}(P) \).

Proof. By Axiom 15, \( \text{list}^+ \) preserves equisatisfiability. Note that any conjuncts inserted by \text{instantiate}_\text{Once}^- \) could also be obtained via classical (not trigger-based) instantiation. Furthermore, since \text{erase}^- \) only removes quantifiers in negative positions and (inconsequential) triggers, the resulting propositions are implied by the original non-erased VC. Finally, with existential quantifiers only in negative positions, the erased and lifted result is quantifier-free. Therefore, Axiom 1 can be used to show that a valid VC according to the decision procedure is also valid without trigger-based instantiation\(^{11}\).

4.3 Syntax and Operational Semantics of \( \lambda^S \)

Figure 5 defines the syntax of \( \lambda^S \) programs. Programs are assumed to be in A-normal form [Flanagan et al. 1993] and the dynamic semantics uses environments and stack configurations. This formalism avoids substitution in expressions and assertions in order to simplify subsequent proofs.

Here, a function definition \( \lambda x. P \) is annotated with a precondition \( R \) and a postcondition \( S \). These specifications can include terms \( \tau \) such as constants, program variables, and uninterpreted function application \( \tau(x) \) as well as logical connectives, and a special syntax \( \text{spec } \tau(x) \text{ req } R \text{ ens } S \) that describes the pre- and postcondition of a function value.

As an example, the JavaScript function \( \text{inc}(n) \) in Listing 3 could be expressed in \( \lambda^S \) as follows:

\[ \text{let } \text{inc}(x) \text{ req } \text{isInt}(x) \text{ ens } (\text{isInt}(\text{inc}(x)) \land (\text{inc}(x) > x)) = x + 1 \text{ in } \ldots \]

Since \( \lambda^S \) is a pure functional language, the postcondition can refer to the function result with an uninterpreted invocation such as \( \text{inc}(x) \) instead of introducing additional variables (as in \text{esverify}).

Values in \( \lambda^S \) are either constants, such as integers and boolean literals, or closures consisting of a function definition and an environment \( \sigma \) (see Figure 2).

The operational semantics of \( \lambda^S \) are specified by a small-step evaluation relation over stack configurations \( \kappa \), as shown in Figure 6. Most noteworthy, the callee function name is added to the environment at each call to enable recursion, and function pre- and postconditions are not checked or enforced during evaluation.

The evaluation of a stack configuration terminates either by getting stuck or by reaching a successful finishing configuration.

Definition 6 (Evaluation Finished). If \( \text{isFinished}(x) \) then there exists \( \sigma \) and \( x \) such that \( \kappa = (\sigma, \text{return } x) \) and \( x \in \sigma \).

4.4 Program Verification

The actual verification of \( \lambda^S \) expressions is defined in terms of the verification judgement \( P \vdash e : Q \) shown in Figure 7. For simplicity, we adopt the convention that the free variables of \( P \), denoted \( \text{FV}(P) \), must be exactly the set of variables in scope at \( e \).
Given a known precondition $P$ and a expression $e$, a verification rule checks potential VCs and generates a postcondition $Q$. This postcondition contains a hole $\kappa$ for the evaluation result of $e$. Since $\lambda^S$ is purely functional, $P$ still holds after evaluating $e$, so we call $Q$ the marginal postcondition. The strongest postcondition is actually $P \land Q[\kappa]$.

As an example, a unary operation such as let $y = \&x$ in $e$ is verified with the rule $\text{vc-unop}$. It requires $x$ to be a variable in scope, i.e. a variable that is free in the precondition $P$. To avoid name clashes, the result $y$ should not be free. Additionally, the VC ($P \Rightarrow \text{pre}(\emptyset, x)$) needs to be valid for all free variables (such as $x$). This check ensures that the value of $x$ is in the domain of the operator $\&$. The rules $\text{vc-binop}$, $\text{vc-if}$, etc. follow analogously.

For function applications $f(x)$, an additional $\text{call}(x)$ trigger is assumed in order to instantiate quantified formulas that correspond to the function definition or specification of the callee.

The most complex rule concerns the verification of function definitions, such as let $f(x)$ req $R$ ens $S = e_1$ in $e_2$. Here, the precondition $R$, the specification of $f$ and the marginal postcondition $Q_1[f(x)]$ together have to imply the annotated postcondition $S$. Any recursive calls of $f$ appearing in its function body will instantiate its own (non-recursive) specification, while subsequent calls of $f$ in $e_2$ will use a postcondition that is strengthened by the generated marginal postcondition. This corresponds to expanding or inlining the definition function by one level at each non-recursive call site.

The special syntax $\text{spec}(\kappa)$ req $R$ ens $S$, as used in verification rules, user-provided pre- and post-conditions, is a notation that desugars to a universal quantifier.

Notation 3 (Function Specifications). $\text{spec}(\kappa)$ req $R$ ens $S \equiv isFunc(\kappa) \land \forall x.(\text{call}(x) \Rightarrow ((R \Rightarrow \text{pre}(\tau, x)) \land (\text{post}(\tau, x) \Rightarrow S)))$

That is, if a function call instantiates this quantifier, the precondition $R$ of the $\text{spec}$ satisfies the precondition of $f$ and the postcondition of $S$ is implied by the postcondition of $f$. For a concrete function call, this means that $R$ needs to be asserted by the calling context and $S$ can be assumed at the call site.

4.5 Soundness
Given the decision procedure and the verification rules described in the previous sections, it is possible to show that verified programs can be evaluated to completion without getting stuck.

First, it is important to note that quantifiers in VCs only appear in certain positions.

**Lemma 1.** If $P$ is a proposition with existential quantifiers only in positive positions, then each VC $P'$ used in the derivation tree of $P \Rightarrow e : Q$ has existential quantifiers only in negative positions.

**Proof.** All VCs in the verification rules shown in Figures 7 are implications $(P'' \Rightarrow Q'')$. In each of these implications, there are no existential quantifiers in $Q''$, as user-supplied postconditions $S$ have no existential quantifiers. Additionally, all propositions $P''$ on the left-hand side have existential quantifiers only in positive positions, since newly introduced existential quantifiers in marginal postconditions are always in positive positions.

From Lemma 1 and Theorem 2, it follows that verification always terminates, ensuring a predictable verification process.

As mentioned in section 4.1, the axiomatization of logical proposition does not include evaluation and treats terms $\tau(\tau)$ as uninterpreted function calls. However, for a proof of verification soundness, it can be shown that a given closed application to an argument value evaluates to a value, it will be necessary to establish facts about this function application.

**Axiom 16.** If $(\sigma[f \Rightarrow (f(x) \Rightarrow \text{req} \ens S[e], x \Rightarrow v_\kappa)] \Rightarrow e)$ \Rightarrow $(\sigma'[y] \Rightarrow v_\kappa)$

Similarly, axioms about $\text{pre}(f, x)$ and $\text{post}(f, x)$ can be added for concrete values of $f$ and $x$.

**Axiom 17.** If $\sigma[f \Rightarrow (f(x) \Rightarrow \text{req} \ens S[e], x \Rightarrow v_\kappa)] \Rightarrow R$ then $\vdash \text{pre}(f(x)) \Rightarrow \text{req} \ens S[e], v_\kappa$.

**Axiom 18.** If $\vdash \sigma : Q_1$ and $\text{spec}(f(x)) \Rightarrow \text{req} \ens S \land R \Rightarrow e : Q_2[\bullet]$ and $\sigma[f \Rightarrow (f(x)) \Rightarrow \text{req} \ens S[e], x \Rightarrow v_\kappa] \Rightarrow Q_1[f(x)] \land S$ then $\vdash \text{pre}(f(x)) \Rightarrow \text{req} \ens S[e], v_\kappa$.

**Axiom 19.** If $\vdash \sigma : Q_1$ and $\text{spec}(f(x)) \Rightarrow \text{req} \ens S \land R \Rightarrow e : Q_2[\bullet]$ and $\vdash \text{post}(f(x)) \Rightarrow \text{req} \ens S[e], v_\kappa$ then $\sigma[f \Rightarrow (f(x)) \Rightarrow \text{req} \ens S[e], x \Rightarrow v_\kappa] \Rightarrow Q_2[f(x)] \land S$.

We can now show that verifiable expressions according to the rules in Figure 7 can be evaluated without getting stuck, i.e. all reachable configurations finish normally or can be evaluated further.

**Theorem 4 (Verification Safety).** If true $\vdash e : Q$ and $(\emptyset, e) \Rightarrow^* \kappa$ then isFinished($\kappa$) or $\kappa \Rightarrow^* \kappa'$ for some $\kappa'$. 

**Figure 6: Operational semantics**
The core language $\lambda^S$ includes higher-order functions but does not address other language features supported by esverify, such as imperative programs and complex recursive data types.

Extending $\lambda^S$ for imperative programs would entail syntax, semantics and verification rules for allocating, mutating and referencing values stored in the heap. Most noteworthy, loops and recursion invalidate previous facts about heap contents and therefore require precise invariants. This issue can be addressed with segment linkage and dynamic frames [Smans et al. 2009].

Additionally, $\lambda^S$ can be extended to support “classes” as shown in Figure 8. These classes are immutable and more akin to recursive data types as they do not support inheritance. Each class definition consists of an ordered sequence of fields and an invariant $S$ that is specified in terms of a free variable $this$. The class invariant can be used to express complex recursive data structures such as the parameterized linked list shown in Section 2.5.

A complete proof is available at: https://github.com/levjj/esverify-theory/
of its fields.

An excerpt of such a language. Here, a type is either a dependent func-
tionally typed language.

Despite being dynamically typed, the verification rules shown in
Figure 7 resemble static typing rules. In this section, we provide a
brief comparison of this program verification approach with static
type checking.

An comprehensive formalization of refinement and dependent
type systems and a formal proof that examines their expressiveness
is beyond the scope of this paper. However, by describing a
translation of types to assertions and investigating concrete exam-
pies, we enable a comparison of esverify with systems such as Liq-
uidHaskell [Vazou et al. 2014] and conjecture that it is at least as
expressive.

First, we assume a language $\lambda^T$ similar to $\lambda^S$ but with type an-
notations instead of pre- and postconditions. Figure 9 shows an
excerpt of such a language. Here, a type is either a dependent func-
tion type or a refined base type where refinements $R$ are consistent
with specifications in $\lambda^S$.

Given such a language, the typing rule for function definitions
($\tau$-fn) checks the function body $t_1$ and compares its type $T_1$ with
the annotated return type $T$. This return type might refer to the
function argument in order to support dependent types. However, oth-
erwise free variables in refinements can break hygiene, so $\tau$-fn
restricts free variables in user-provided types $T_x$ and $T$ accordingly.

Figure 9 also shows the subtyping relation. Most relevantly, sub-
typing of refined base types requires checking an implication be-
tween the refinements. Therefore, the typing rules also involve a
translation of the type environment $\Gamma$ to a logical formula, where
refinements become propositions and function types translate to the
spec syntax.

Intuitively, the implication used for refinements also extends to
translated function types, so if $\Gamma \vdash T' : T$ then for all terms $r$,
$[\Gamma] \land [r : T]$ implies $[r : T']$.

As an example, the following $\lambda^T$ expression is well-typed as the
return type is a subtype of the argument type:

\[
\begin{align*}
\text{let } f (g) (x) = \text{spec } (x : \text{Int } | x > 3) \to (y : \text{Int } | y > x)) \text{ in } g \end{align*}
\]

Translated into esverify, we obtain a program with spec in pre- and
postcondition:

\[
\begin{align*}
\text{function } f (g) \{ \\
\text{requires}(\text{spec } (g, x => x > 3, (x, y) => y > x)); \\
\text{ensures}(r => \text{spec } (r, x => x > 4, (x, y) => y >= x)); \\
\text{return } g; \}
\end{align*}
\]

This program is verifiable with the quantifier instantiation algo-

Conjecture 1 (Translated well-typed expressions are verifiable).
If $[t]$ is the translation of a $\lambda^T$ expression $t$ to $\lambda^S$, then $\Gamma \vdash t : T$
implies $[\Gamma] \land [t : T]$ for some $Q$.

A formal proof of this conjecture needs to take quantifier in-
stantiation and the quantifier nesting bound into account. This is
beyond the scope of this paper, but might be addressed in future
work.

Coincidentally, such a translation would also enable seamless
interweaving of statically-typed $\lambda^T$ expressions in dynamically-
typed $\lambda^S$ programs in a sound way. This might be a step towards
a full spectrum type system.

6 RELATED WORK

There have been decades of prior work on software verification.
In particular, static verification of general purpose programming
languages based on pre- and postconditions has previously been
explored in verifiers such as ESC/Java [Flanagan et al. 2002; Leino

\[
\begin{align*}
\text{function } f (g) \{ \\
\text{requires}(\text{spec } (g, x => x > 3, (x, y) => y > x)); \\
\text{ensures}(r => \text{spec } (r, x => x > 4, (x, y) => y >= x)); \\
\text{return } g; \}
\end{align*}
\]

This program is verifiable with the quantifier instantiation algo-

Conjecture 1 (Translated well-typed expressions are verifiable).
If $[t]$ is the translation of a $\lambda^T$ expression $t$ to $\lambda^S$, then $\Gamma \vdash t : T$
implies $[\Gamma] \land [t : T]$ for some $Q$.

A formal proof of this conjecture needs to take quantifier in-
stantiation and the quantifier nesting bound into account. This is
beyond the scope of this paper, but might be addressed in future
work.

Coincidentally, such a translation would also enable seamless
interweaving of statically-typed $\lambda^T$ expressions in dynamically-
typed $\lambda^S$ programs in a sound way. This might be a step towards
a full spectrum type system.

ESC/Java [Flanagan et al. 2002] proposed the idea of using undecidable but SMT-solvable logic to provide more powerful static checking than traditional type systems. Their proposed extended static checking gave up on soundness to do so, citing the utility of tools that on balance contributed to finding bugs.

JaVerT [Fragoso Santos et al. 2018] is a more recent program verifier for JavaScript. It supports object-oriented programs but, in contrast to EVERIFY, does not support higher-order functions.

Dafny [Leino 2010] pushed this further by seeking to provide a full verification language, with support for both functional and imperative programming. Dafny provides programmers with advanced constructs for verified programming, such as ghost functions and parameters, termination checking, quantifiers, and reasoning about the heap. However, Dafny also requires function calls in an assertion contexts to satisfy the precondition instead of treating these as uninterpreted calls. Therefore, Dafny does not support higher-order proofs such as those shown in Section 2.8. Additionally, quantifier instantiation in Dafny is often implicit and based on heuristics, which allows for brittle and unpredictable verification.

In trying to find a compromise, with predictable checking but also a larger scope than traditional type systems, LiquidHaskell is most closely related to EVERIFY. In fact, the refinement type system discussed in Section 5 loosely resembles its formalization by Vazou et al. [Vazou et al. 2014].

More recently [Vazou et al. 2018], LiquidHaskell introduced refinement reflection, which enables external proofs in a similar way as the spec construct in EVERIFY, and proof by logical evaluation which is a close cousin to our quantifier instantiation algorithm in Section 4.2 (though it is not based on triggering matching patterns). In contrast to LiquidHaskell, EVERIFY is not based on static type checking and thus also supports dynamically-typed programming idioms such as dynamic type checks instead of injections.

Finally, the decision procedure described in Section 4.2 involves trigger-based quantifier instantiation which has been studied by extensive prior work [Dross et al. 2016; Ge and de Moura 2009; Leino and Pit-Claudel 2016; Reynolds et al. 2013]. The instantiation in EVERIFY is specifically bounded in order to prevent matching loops, but further research could provide this kind of instantiation as a built-in feature of off-the-shelf SMT solvers.

7 CONCLUSION

This paper introduced EVERIFY, a program verifier for dynamically-typed JavaScript programs. EVERIFY supports both dynamic programming idioms as well as higher-order functional programs, and thus has an expressiveness comparable to and potentially greater than common refinement type systems.

Internally, the verifier relies on a bounded quantifier instantiation algorithm, thus avoiding brittle heuristics, and SMT solving, yielding concrete counterexamples for verification errors. We showed that this approach to program verification is sound by formalizing the quantifier instantiation algorithm and the verification rules in the Lean Theorem prover.

While EVERIFY enables verification of non-trivial programs such as MergeSort, it lacks termination checking and support for object-oriented programming. However, it would be possible to combine it with an external termination checker for total correctness [Sereni and Jones 2005], and to extend it with reasoning about the heap, such as dynamic frames [Smans et al. 2009].

Finally, while the approach presented in this paper is purely static, future work might use runtime checks similar to hybrid and gradual type checking [Ahmed et al. 2011; Knowles and Flanagan 2010; Siek and Taha 2006] to enable sound execution of programs that are not fully verified.

REFERENCES


