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

ABSTRACT

Program verifiers 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.

CCS CONCEPTS

• **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:
Christopher Schuster, Sohum Banerjea, and Cormac Flanagan. 2018. ESVERIFY: Verifying Dynamically-Typed Higher-Order Functional Programs by SMT Solving. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3310232.3310240

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, a program verifier might statically verify that the result of a sorting routine is sorted and is a permutation of the input.

In this paper, we present ESVERIFY, a program verification system for JavaScript. JavaScript, a dynamically-typed scripting language, was chosen as target because its broad user base suggests many beneficial use cases for static analysis, and because its availability in browsers enables accessible online demos without local installation.

JavaScript programs often include idioms and patterns that do not adhere to standard typing rules. For instance, the latest edition of the JavaScript/ECMAScript standard [ECMA-262 2017] introduces *promises* such that a promise can be composed with other promises and with arbitrary objects, as long as these objects have a `"then"` method. Since ESVERIFY does not rely on static types, it can easily accommodate these idioms.

JavaScript programs also often use higher-order functions. In order to support verification of these functions, ESVERIFY introduces a new syntax to constrain function values in terms of their pre- and postconditions. Similarly, other JavaScript values such as numbers, strings, arrays and classes can be used in assertions, including invariants on array contents and class instances.

The implementation of ESVERIFY, including its source code[1] and a live demo[2] are available online. In summary, if a JavaScript program uses features unsupported by ESVERIFY, it will be rejected early; otherwise, verification conditions are generated based on annotations, and each verification condition is transformed according to a quantifier instantiation algorithm and then checked by an SMT solver.

In addition to describing the design and implementation or ESVERIFY, we formally define a JavaScript-inspired, statically verified but dynamically typed language, called λ^S. Functions in λ^S 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 λ^S 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 proposing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IFL 2018, September 5–7, 2018, Lowell, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310240

[1] Implementation Source Code: https://github.com/levjj/esverify/
[2] Online live demo of ESVERIFY: https://esverify.org/try

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 λ^S is sound, i.e. verifiable λ^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 [Vazou et al. 2014]. 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 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 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 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 λ^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, esverify, targets a subset of ECMAScript/-JavaScript. By supporting a dynamically-typed scripting language, 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. Instead, the goal is to support both functional as well as object-oriented programming paradigms with an emphasis on functional JavaScript programs with higher-order functions.

2.1 Annotating JavaScript with Assertions

esverify extends JavaScript with source code annotations such as functions pre- and postconditions, loop invariants and statically-checked assertions. These are written as *pseudo function calls* with standard Javascript syntax. While some program verification systems specify these in comments such as ESC/Java [Flanagan et al. 2002], this approach enables a better integration with existing tooling support such as refactoring tools and syntax highlighters.

[3]Formal definitions and proofs in Lean: https://github.com/levjj/esverify-theory/

```
1  function max(a, b) {
2    requires(typeof(a) === 'number');
3    requires(typeof(b) === 'number');
4    ensures(res => res >= a);
5    ensures(res => res >= b); // does not hold
6    if (a >= b) {
7      return a;
8    } else {
9      return a; // bug
10   }
11 }
```

Listing 1: A JavaScript function `max` annotated with pre- and postconditions.

```
1  function sumTo (n) {
2    requires(Number.isInteger(n) && n >= 0);
3    ensures(res => res === (n + 1) * n / 2);
4    let i = 0;
5    let s = 0;
6    while (i < n) {
7      invariant(Number.isInteger(i) && i <= n);
8      invariant(Number.isInteger(s));
9      invariant(s === (i + 1) * i / 2);
10     i++;
11     s = s + i;
12   }
13   return s;
14 }
```

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 a subset of JavaScript. It does not support all of JavaScript's semantics. In particular, it is restricted to pure expressions that do not contain function definitions.

2.2 `max`: A Simple Example

Listing 1 shows an example of an annotated JavaScript program. The calls to `requires` and `ensures` in lines 2–5 are only used for verification purposes and excluded from evaluation. Instead of introducing custom type annotations, the standard JavaScript `typeof` operator is used to constrain the possible values passed as function arguments. Due to a bug in line 9, the `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, static analysis can check various correctness properties precisely. However, the potential behavior of programs with loops or recursion cannot be determined statically. esverify "overapproximated" the behavior of the program, i.e. 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 of iterations or recursive function calls.

```
1  function inc (x) {
2    requires(Number.isInteger(x));
3    ensures(y => Number.isInteger(y) && y > x);
4    // implicit: ensures(y => y === x + 1);
5    return x + 1;
6  }
7  function twice (f, n) {
8    requires(spec(f, (x)   => Number.isInteger(x),
9                     (x,y) => Number.isInteger(y) &&
10                              y > x));
11   requires(Number.isInteger(n));
12   ensures(res => res >= n + 2);
13   return f(f(n));
14 }
15 const n = 3;
16 const m = twice(inc, n); // 'inc' satisfies spec
17 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`.

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[4]. Without these loop invariants, the state of `i` and `s` would be unknown in line 13 except for the fact that `i < n` is false. However, when combined with the loop invariants, the equality $i == n$ can be inferred after the loop and thereby the postcondition in line 3 can be verified. ESVERIFY internally uses standard SMT theorems for integer arithmetic to establish that the invariants are maintained for each iteration of the loop.

There is extensive prior work on automatically inferring loop invariants [Furia and Meyer 2010]. Recent research suggest that automatic inference can also be extended to program invariants [Ernst et al. 2001] and specifications [Henkel and Diwan 2003]. 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 the call `twice(inc,n)` in line 17 requires ESVERIFY to compare the pre- and postconditions of `inc` with pre- and postconditions in lines 8–10. It is important to note that ESVERIFY implicitly strengthens the stated postcondition of `inc` by inlining its function body `x + 1`. Recursive functions are only inlined by one level, so these need to be explicitly annotated with adequate pre- and postconditions for verification purposes, similarly to loop invariants.

[4]Here, `Number.isInteger(i)` ensures that `i` is an actual integer, while **`typeof`**`(i) === 'number'` is also true for floating point numbers.

```
1  function f (a) {
2    requires(a instanceof Array);
3    requires(a.every(e => e > 3));
4    requires(a.length >= 2);
5    assert(a[0] > 2); // holds
6    assert(a[1] > 4); // fails as a[1] might be 4
7    assert(a[2] > 1); // a may have only 2 elements
8  }
```

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

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. Specifically, mutation of arrays and objects is not currently supported and objects have to be either immutable *dictionaries* that map string keys to values or instances of user-defined classes with a fixed set of fields 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. To simplify reasoning, the pseudo call **`pure`**`()` in the postcondition ensures the absence of side effects. 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

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 select and return portions of the current webpage.

Even standard JavaScript objects use 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 Shrira 1988] and specifies a polymorphic `Promise.resolve()` function. This function behaves differently depending on whether it is called with a promise, an arbitrary non-promise object with a method called `"then"`, or a non-promise object without such a method. ESVERIFY can accurately express these kinds of specifications in pre- and postconditions as

```
class List {
  constructor (head, tail, each) {
    this.head = head; this.tail = tail; this.each = each;
  }
  invariant () {
    // this.each is a predicate that is true for this element and the rest of the list
    return spec(this.each, x => true, (x, y) => pure() && typeof(y) === 'boolean') &&
          (true && this.each)(this.head) && // same as 'this.each(this.head)' but without binding 'this'
          (this.tail === null || (this.tail instanceof List && this.each === this.tail.each));
  }
}
function map (f, lst, newEach) {
  // newEach needs to be a predicate
  // (a pure function without precondition that returns a boolean)
  requires(spec(newEach, x => true, (x, y) => pure() && typeof(y) === 'boolean'));
  // the current predicate 'this.each' must satisfy the precondition of 'f'
  // and the return value of 'f' needs to satisfy the new predicate 'newEach'
  requires(lst === null || spec(f, x => (true && lst.each)(x), (x, y) => pure() && newEach(y)));
  requires(lst === null || lst instanceof List);
  ensures(res => res === null || (res instanceof List && res.each === newEach));
  ensures(pure()); // necessary as recursive calls could otherwise invalidate the class invariant
  return lst === null ? null : new List(f(lst.head), map(f, lst.tail, newEach), newEach);
}
```

Listing 5: Custom linked list class with a field `each` which is a function that is true for all elements. Mapping over the list results in a new list whose elements satisfy a new predicate analogous to a map function in a parametrized type system.

```
class Promise {
  constructor (value) { this.value = value; }
}
function resolve (fulfill) {
  // "fulfill" is promise, then-able or
  // a value without a "then" property
  requires(fulfill instanceof Promise ||
           spec(fulfill.then, () => true,
                              () => true) ||
           !('then' in fulfill));
  ensures(res => res instanceof Promise);
  if (fulfill instanceof Promise) {
    return fulfill;
  } else if ('then' in fulfill) {
    return new Promise(fulfill.then());
  } else {
    return new Promise(fulfill);
  }
}
```

Listing 6: The standard `Promise.resolve()` function in JavaScript has complex polymorphic behavior. This simplified mock definition illustrates how ESVERIFY enables such dynamic programming idioms.

shown in Listing 6, while standard type systems need to resort to code changes, such as 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. Interestingly, about 48 out of a total 99 lines are verification annotations, including invariants, pre- and postconditions and the predicate function `isSorted`. `isSorted` is primarily used in specifications, but the implementations of `merge` and `sort` also include calls to it. These calls are used as *triggers*, hints to the underlying SMT solver that do not contribute to the result. In other verified languages such as Dafny [Leino 2013], `isSorted` would correspond to a "ghost function", but 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 `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].

[5]The source code of a MergeSort algorithm in ESVERIFY is available at https://esverify.org/try#msort.

```
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);
  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_fib_inc (n) {
  requires(Number.isInteger(n) && n >= 0);
  ensures(fib(n) <= fib(n + 1));
  ensures(pure());
  fib(n); // unfolds fib at n
  fib(n + 1);
  if (n > 0) {
    fib(n - 1);
    proof_fib_inc(n - 1); // I.H.
  }
  if (n > 1) {
    fib(n - 2);
    proof_fib_inc(n - 2); // I.H.
  }
}
function proof_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_mono(fib, proof_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].

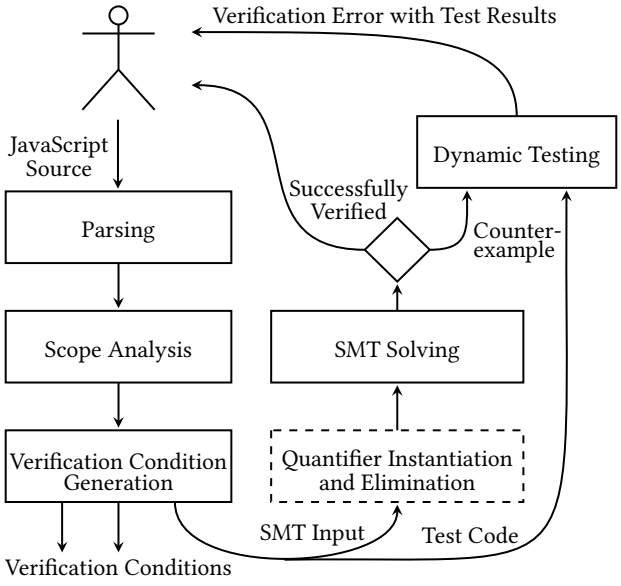

Figure 1: The basic verification workflow: ESVERIFY generates and statically checks verification conditions by SMT solving.

3 IMPLEMENTATION

The ESVERIFY prototype implementation[6] is available online. Because the implementation itself is written in TypeScript, a dialect of JavaScript, it can be used in a browser. Indeed, there is a browser-based editor with ESVERIFY checking[7]. Alternative integrations such as extensions for Vim and Emacs also exist.

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 supported by ESVERIFY. Some of these restrictions may be lifted in future versions of ESVERIFY, such as support for regular expressions or functions with a variable number of arguments. However, other JavaScript features would involve immense complexity for accurate verification due to their dynamic character and their interactions with the rest of the program, such as metaprogramming with `eval` or `new Function()`. Additionally, ESVERIFY does not support features that 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 expressions and assertions. For example, `spec` can only 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 and references to unsupported global objects. In addition to user-provided definitions, it includes a whitelist of globals supported by ESVERIFY, such as `Array`, `Math` and `console`. The analysis also takes mutability into account. For example, mutable variables cannot be referenced in

[6] Implementation Source Code: https://github.com/levjj/esverify/
[7] Online live demo of ESVERIFY: https://esverify.org/try

$$\begin{array}{c} \forall a \forall b. \\ \texttt{typeof}(a) = \text{“}number\text{”} \\ \wedge\ \texttt{typeof}(b) = \text{“}number\text{”} \\ \wedge\ (a > b) \implies result = a \\ \wedge\ \neg(a > b) \implies result = a \\ \implies result \geq a \end{array}$$

Figure 2: A simplified verification condition for the postcondition of the `max` function in line 4 of Listing 1.

$$\begin{array}{ll} (\forall a, b. \texttt{max}(a, b) \geq a) & (\forall a, b. \texttt{max}(a, b) \geq a) \\ \implies \texttt{max}(3, 5) > 0 & \wedge\ \texttt{max}(3, 5) \geq 3 \\ & \implies \texttt{max}(3, 5) > 0 \end{array}$$

Figure 3: The proposition on the left has a universal quantifier. On the right, this quantifier is instantiated with concrete values of a and b, yielding an augmented proposition that can be verified with simple arithmetic.

class invariants, and the `old(x)` syntax in a postcondition requires `x` to be a mutable variable.

The main **verification** step is implemented as a traversal of the source program that generates verification conditions and maintains a *verification context*. Most notably, the verification context includes a logical proposition that acts as precondition and a set of variables with unknown values. Generated verification conditions combine this context with an assertion, such as a function postcondition. Figure 2 illustrates this process for a simple example. The verification condition checks whether the preconditions and the translated function body imply the postcondition. Section 4.4 describes the verification rules in more detail.

The verification condition is then transformed with a **quantifier instantiation** procedure. As illustrated by Figure 3, quantified propositions in verification conditions need to be instantiated with concrete values in order to determine satisfiability of the formula. Quantifiers are instantiated based on matching triggers and remaining quantifiers are then erased from the proposition[8]. The resulting quantifier-free proposition can be checked by SMT solving, ensuring that the verification process remains predicable. However, this approach to quantifier instantiation requires the programmer to provide explicit triggers as function calls. Alternatively, the trigger-based quantifier instantiation can be skipped and the proposition passed directly to the SMT solver, which internally performs instantiations based on heuristics.

The final 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 find a solution for the negated verification condition, i.e. if the solver cannot refute the proposition, 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 involves dynamic checking of assertions and serves two purposes. On the one hand, the test might not be able to 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].

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

$$\begin{array}{lll} \phi \in \text{Propositions} & ::= & \tau \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid pre_1(\otimes, \tau) \mid \\ & & pre_2(\oplus, \tau, \tau) \mid pre(\tau, \tau) \mid post(\tau, \tau) \mid \forall x.\phi \\ \tau \in \text{Terms} & ::= & v \mid x \mid \otimes\, \tau \mid \tau \oplus \tau \mid \tau(\tau) \\ \otimes \in \text{UnaryOperators} & ::= & \neg \mid isInt \mid isBool \mid isFunc \\ \oplus \in \text{BinaryOperators} & ::= & + \mid - \mid \times \mid / \mid \wedge \mid \vee \mid = \mid < \\ v \in \text{Values} & ::= & \text{true} \mid \text{false} \mid n \mid \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma \rangle \\ \sigma \in \text{Environments} & ::= & \emptyset \mid \sigma[x \mapsto v] \\ n \in \mathbb{N} & & f, x, y, z \in \text{Variables} \end{array}$$

Figure 4: Syntax of logical propositions used in the verifier.

4 FORMALISM

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

The verification rules of λ^S use verification conditions whose validity is checked with a custom decision procedure, so this section first formally defines logical propositions and axiomatizes their validity and then describes the decision procedure including quantifier instantiation. Finally, the syntax and semantics of λ^S 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[9].

4.1 Logical Foundation

Figure 4 formally defines the syntax of propositions, terms, values and environments. Propositions ϕ can use terms, connectives $\neg$, $\wedge$ and $\vee$, symbols $pre_1, pre_2, pre, post$ and universal quantifiers. Here, terms τ are either values, variables, unary or binary operations or uninterpreted function calls. Finally, values v include boolean and integer constants as well as *closures* which are opaque values that will be explained in Section 4.3.

Instead of defining validity of propositions in terms of an algorithm such as SMT solving, this formal development uses an axiomatization of the validity judgement $\vdash \phi$.

Standard axioms for logical connectives and quantifiers are omitted here for brevity but can be found Lean proof. Most noteworthy, axioms about unary and binary operators that are specified in terms of a partial function δ, e.g. $\delta(+, 2, 3) = 5$.

Axiom 1. Iff $\delta(\otimes, v_x) = v$ then $\vdash v = \otimes v_x$. $\boxed{\vdash \phi}$

[9]Formal definitions and proofs in Lean: https://github.com/levjj/esverify-theory/

Axiom 2. Iff $\delta(\oplus, v_x, v_y) = v$ then $\vdash v = v_x \oplus v_y$.

The propositions $pre_1(\otimes, v_x)$ and $pre_2(\otimes, v_x, v_y)$ can be used to reason about the domain of operators.

Axiom 3. If $\vdash pre_1(\otimes, v_x)$ then $(\otimes, v_x) \in dom(\delta)$.

Axiom 4. If $\vdash pre_2(\oplus, v_x, v_y)$ then $(\oplus, v_x, v_y) \in dom(\delta)$.

Similarly, the constructs $pre(f, x)$ and $post(f, x)$ in propositions denote the pre- and postcondition of a function f when applied to a given argument x. However, in contrast to pre_1 and pre_2, the logical foundations do not contain axioms for pre and $post$. The interpretation of pre and $post$ is instead determined by their use in the generated verification condition.

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

Axiom 5. If x is free in ϕ and $\vdash \phi$ then $\vdash \forall x.\ \phi$.

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 1 (Satisfiability)**.** $Sat(\phi)$ denotes that the SMT solver found a model that satisfies ϕ.

$$\boxed{Sat(\phi)}$$

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

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 above, verification of λ^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 infinite matching loops. This section describes a bounded quantifier instantiation algorithm that avoids matching loops and brittle heuristics, thus enabling a predictable decision procedure for verification conditions.

$$\begin{aligned} P \in \text{VerificationConditions} ::= \\ \tau \mid \neg P \mid P \wedge P \mid P \vee P \mid pre_1(\otimes, \tau) \mid pre_2(\oplus, \tau, \tau) \mid \\ pre(\tau, \tau) \mid post(\tau, \tau) \mid call(\tau) \mid \forall x.\{call(x)\} \Rightarrow P \mid \exists x.P \end{aligned}$$

The syntax of verification conditions P used in verification rules is similar to the syntax for propositions but universal quantifiers in verification conditions (VCs) have explicit matching patterns to indicate that instantiation requires a *trigger*. Accordingly, the construct $call(x)$ is introduced to act as an instantiation trigger that does not otherwise affect validity of propositions, i.e. $call(x)$ can always assumed to be true. Intuitively, $call(x)$ represents a function call or an asserted function specification while $\forall x.\{call(x)\} \Rightarrow P$ corresponds to a function definition or an assumed function specification. For the trigger $call(x)$, x denotes the argument of the call; the callee is omitted as triggers are matched irrespective of their callees by the instantiation algorithm.

The complete decision procedure for VCs including quantifier instantiation is shown in Figure 5.

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

The procedure $lift^+$ matches universal quantifiers in positive and existential quantifiers in negative positions. In both cases, an equivalent VC without the quantifier can be obtained by renaming the quantified variable to a fresh variable that is implicitly universally quantified. It is important to note that the matching pattern $call(y)$ of a universal quantifier now becomes a part of an implication thereby available to instantiate further quantifiers. The lifting is repeated until no more such quantifiers can be found.

The procedure $instantiateOnce^-$ performs one round of trigger-based instantiation such that each universal quantifier with negative polarity is instantiated with all available triggers in negative position. All such instantiations are conjoined with the original quantifier.

Both lifting and instantiation are repeated for multiple iterations by the recursive $instantiate^-$ procedure. As a final step, $erase^-$ 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 considered valid if SMT solving cannot refute the resulting proposition.

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 verifier 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 $lift^+$ function eliminates a quantifier during each recursive call and therefore terminates when there are no more matching quantifiers in the formula. $instantiateOnce^-$ and $erase^-$ are 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 also terminates. □

It is now possible to compare the axiomatized validity judgement for propositions $\vdash \phi$ with the decision procedure for verification conditions $\langle P \rangle$ by translating the VC P to a proposition ϕ without triggers or matching patterns.

Definition 2 (Proposition Translation)**.** $prop(P)$ denotes a proposition such that triggers and matching patterns in P are removed and existential quantifiers $\exists x.\ P$ translated to $\neg\forall x.\ \neg prop(P)$.

$$P^+[\circ] ::= \quad \circ \mid \neg P^-[\circ] \mid P^+[\circ] \wedge P \mid P \wedge P^+[\circ] \mid P^+[\circ] \vee P \mid P \vee P^+[\circ] \qquad calls^+(P) \stackrel{\text{def}}{=} \{\, call(\tau) \mid P = P^+[call(\tau)] \,\}$$

$$P^-[\circ] ::= \quad \neg P^+[\circ] \mid P^-[\circ] \wedge P \mid P \wedge P^-[\circ] \mid P^-[\circ] \vee P \mid P \vee P^-[\circ] \qquad calls^-(P) \stackrel{\text{def}}{=} \{\, call(\tau) \mid P = P^-[call(\tau)] \,\}$$

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

$$instantiateOnce^-(P) \stackrel{\text{def}}{=} P\left[P^-\left[(\forall x.\{call(x)\} \Rightarrow P')\right] \mapsto P^-\left[(\forall x.\{call(x)\} \Rightarrow P') \wedge \bigwedge_{call(\tau) \in calls^-(P)} P'[x \mapsto \tau]\right]\right]$$

$$erase^-(P) \stackrel{\text{def}}{=} P\left[P^-\left[(\forall x.\{call(x)\} \Rightarrow P'')\right] \mapsto P^-\left[\text{true}\right],\ P^+\left[call(\tau)\right] \mapsto P^+\left[\text{true}\right],\ P^-\left[call(\tau)\right] \mapsto P^-\left[\text{true}\right]\right]$$

$$instantiate^-(P, n) \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } erase^-(lift^+(P)) \text{ else } instantiate^-(instantiateOnce^-\left(lift^+(P)\right), n-1)$$

$$\langle P \rangle \stackrel{\text{def}}{=} \text{let } n = \text{maximum level of quantifier nesting of } P \text{ in } \neg Sat\left(\neg instantiate^-(P, n)\right) \qquad \boxed{\langle P \rangle}$$

Figure 5: The decision procedure lifts, instantiates and finally eliminates quantifiers. The number of iterations is bounded by the maximum level of quantifier nesting.

$$\begin{array}{rl} e \in \text{Expressions} & ::= \\ & \text{let } x = \text{true in } e \mid \text{let } x = \text{false in } e \mid \text{let } x = n \text{ in } e \mid \\ & \text{let } f(x) \text{ req } R \text{ ens } S = e \text{ in } e \mid \text{let } y = \otimes x \text{ in } e \mid \\ & \text{let } z = x \oplus y \text{ in } e \mid \text{let } y = f(x) \text{ in } e \mid \text{if } (x)\ e \text{ else } e \mid \text{return } x \\ R, S \in \text{Specs} & ::= \tau \mid \neg R \mid R \wedge R \mid R \vee R \mid \text{spec } \tau(x) \text{ req } R \text{ ens } S \\ \kappa \in \text{Stacks} & ::= (\sigma, e) \mid \kappa \cdot (\sigma, \text{let } y = f(x) \text{ in } e) \end{array}$$

Figure 6: Syntax of λ^S programs. Function definitions have pre- and postconditions written as simple logical propositions with the *spec* syntax for higher-order functions.

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

Proof. By Axiom 5, $list^+$ preserves equisatisfiability. Note that any conjuncts inserted by $instantiateOnce^-$ could also be obtained via classical (not trigger-based) instantiation. Furthermore, since $erase^-$ 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[10]. □

4.3 Syntax and Operational Semantics of λ^S

Figure 6 defines the syntax of λ^S. 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 let $f(x)$ req R ens S = e_1 in e_2 is annotated with a precondition R and a postcondition S. These specifications can include terms τ such as constants, program variables, and uninterpreted function application $\tau(\tau)$ as well as logical connectives, and a special syntax "spec $\tau(x)$ req R ens S" that describes the pre- and postcondition of a function value.

The operational semantics of λ^S is specified by a small-step evaluation relation over stack configurations κ, as shown in Figure 7. Most noteworthily, 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 completion configuration.

Definition 3 (Evaluation Finished)**.** A stack κ has terminated successfully, abbreviated with $terminated(\kappa)$, if there exists σ and x such that $\kappa = (\sigma, \text{return } x)$ and $x \in \sigma$.

4.4 Program Verification

The verification rules of λ^S are inductively defined in terms of a verification judgement $P \vdash e : Q$ as shown in Figure 8. Given a known precondition P and an expression e, a verification rule checks potential verification conditions and generates a postcondition Q. This postcondition contains a hole $\bullet$ for the evaluation result of e. Since λ^S is purely functional, P still holds after evaluating e, so we call Q the *marginal postcondition* and $P \wedge Q[\bullet]$ the *strongest postcondition*.

As an example, a unary operation such as let $y = \otimes x$ in e is verified with the rule 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 $\langle P \Rightarrow pre(\otimes, x) \rangle$ needs to be valid for all assignments of free variables (such as x). This check ensures that the value of x is in the domain of the operator $\otimes$. The rules VC-BINOP, VC-IF, etc. follow analogously.

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

[10] A complete proof is available at: https://github.com/levjj/esverify-theory/

$$\boxed{\kappa \hookrightarrow \kappa}$$

$$\begin{array}{rll}
(\sigma, \text{let } x = v \text{ in } e) \hookrightarrow (\sigma[x \mapsto v], e) & \text{where } v \in \{\text{true}, \text{false}, n\} & \text{[E-VAL]} \\
(\sigma, \text{let } f(x) \text{ req } R \text{ ens } S = e_1 \text{ in } e_2) \hookrightarrow (\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e_1\}, \sigma\rangle], e_2) & & \text{[E-CLOSURE]} \\
(\sigma, \text{let } y = \otimes x \text{ in } e) \hookrightarrow (\sigma[y \mapsto v], e) & \text{where } v = \delta(\otimes, \sigma(x)) & \text{[E-UNOP]} \\
(\sigma, \text{let } z = x \oplus y \text{ in } e) \hookrightarrow (\sigma[z \mapsto v], e) & \text{where } v = \delta(\oplus, \sigma(x), \sigma(y)) & \text{[E-BINOP]} \\
(\sigma, \text{let } z = f(y) \text{ in } e) \hookrightarrow (\sigma_f[g \mapsto \sigma(f), x \mapsto \sigma(y)], e_f) \cdot (\sigma, \text{let } z = f(y) \text{ in } e) & & \text{[E-CALL]} \\
& \text{where } \sigma(f) = \langle g(x) \text{ req } R \text{ ens } S\ \{e_f\}, \sigma_f\rangle & \\
(\sigma, \text{return } z) \cdot (\sigma_2, \text{let } y = f(x) \text{ in } e_2) \hookrightarrow (\sigma_2[y \mapsto \sigma(z)], e_2) & & \text{[E-RETURN]} \\
(\sigma, \text{if } (x)\ e_1 \text{ else } e_2) \hookrightarrow (\sigma, e_1) & \text{if } \sigma(x) = \text{true} & \text{[E-IF-TRUE]} \\
(\sigma, \text{if } (x)\ e_1 \text{ else } e_2) \hookrightarrow (\sigma, e_2) & \text{if } \sigma(x) = \text{false} & \text{[E-IF-FALSE]} \\
\kappa \cdot (\sigma, \text{let } y = f(x) \text{ in } e) \hookrightarrow \kappa' \cdot (\sigma, \text{let } y = f(x) \text{ in } e) & \text{if } \kappa \hookrightarrow \kappa' & \text{[E-CONTEXT]}
\end{array}$$

Figure 7: Operational semantics

$$\boxed{P \vdash e : Q}$$

$$\begin{array}{rcl}
Q[\bullet] \in \text{PropositionContexts} & ::= & P \mid \eta[\bullet] \mid \neg Q[\bullet] \mid Q[\bullet] \wedge Q[\bullet] \mid Q[\bullet] \vee Q[\bullet] \mid pre_1(\otimes, \eta[\bullet]) \mid pre_2(\oplus, \eta[\bullet], \eta[\bullet]) \mid \\
& & pre(\eta[\bullet], \eta[\bullet]) \mid post(\eta[\bullet], \eta[\bullet]) \mid call(\eta[\bullet]) \mid \forall x.\{call(x)\} \Rightarrow Q[\bullet] \mid \exists x.\ Q[\bullet] \\
\eta[\bullet] \in \text{TermContexts} & ::= & \bullet \mid \tau \mid\ \otimes \eta[\bullet] \mid \eta[\bullet] \oplus \eta[\bullet] \mid \eta[\bullet](\eta[\bullet])
\end{array}$$

$$\frac{x \notin FV(P) \quad v \in \{\text{true}, \text{false}, n\} \quad P \wedge x = \text{v} \vdash e : Q}{P \vdash \text{let } x = v \text{ in } e : \exists x.\ x = v \wedge Q}\ \text{VC-VAL} \qquad \frac{x \in FV(P) \quad y \notin FV(P) \quad \langle P \Rightarrow pre(\otimes, x)\rangle \quad P \wedge y = \otimes x \vdash e : Q}{P \vdash \text{let } y = \otimes x \text{ in } e : \exists y.\ y = \otimes x \wedge Q}\ \text{VC-UNOP}$$

$$\frac{x \in FV(P) \quad y \in FV(P) \quad z \notin FV(P) \quad \langle P \Rightarrow pre(\oplus, x, y)\rangle \quad P \wedge z = x \oplus y \vdash e : Q}{P \vdash \text{let } z = x \oplus y \text{ in } e : \exists z.\ z = x \oplus y \wedge Q}\ \text{VC-BINOP}$$

$$\frac{\begin{array}{c} f \notin FV(P) \quad x \notin FV(P) \quad f \neq x \quad x \in FV(R) \quad FV(R) \subseteq FV(P) \cup \{f, x\} \quad FV(S) \subseteq FV(P) \cup \{f, x\} \\ P \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e_1 : Q_1 \quad \langle P \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \wedge Q_1[f(x)] \Rightarrow S\rangle \\ P \wedge \text{spec } f(x) \text{ req } R \text{ ens } (Q_1[f(x)] \wedge S) \vdash e_2 : Q_2 \end{array}}{P \vdash \text{let } f(x) \text{ req } R \text{ ens } S = e_1 \text{ in } e_2 : \exists f.\ \text{spec } f(x) \text{ req } R \text{ ens } (Q_1[f(x)] \wedge S) \wedge Q_2}\ \text{VC-FUNC}$$

$$\frac{f \in FV(P) \quad x \in FV(P) \quad y \notin FV(P) \quad \langle P \wedge call(x) \Rightarrow isFunc(f) \wedge pre(f, x)\rangle \quad P \wedge call(x) \wedge post(x) \wedge y = f(x) \vdash e : Q}{P \vdash \text{let } y = f(x) \text{ in } e : \exists y.\ call(x) \wedge post(f, x) \wedge y = f(x) \wedge Q}\ \text{VC-APP}$$

$$\frac{x \in FV(P) \quad \langle P \Rightarrow isBool(f)\rangle \quad P \wedge x \vdash e_1 : Q_1 \quad P \wedge \neg x \vdash e_2 : Q_2}{P \vdash \text{if } (x)\ e_1 \text{ else } e_2 : (x \Rightarrow Q_1) \wedge (\neg x \Rightarrow Q_2)}\ \text{VC-ITE} \qquad \frac{x \in FV(P)}{P \vdash \text{return } x : x = \bullet}\ \text{VC-RETURN}$$

Figure 8: The judgement $P \vdash e : Q$ verifies the expression e given a known context P.

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 annotated 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 (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 function definition by one level at each non-recursive callsite.

The special syntax spec $\tau(x)$ req R ens S, as used in verification rules, user-provided pre- and postconditions, is a notation that desugars to a universal quantifier when appearing in a verification condition.

Notation 1 (Function Specifications)**.** spec $\tau(x)$ req R ens $S \equiv$ $isFunc(\tau) \wedge \forall x.\{call(x)\} \Rightarrow ((R \Rightarrow pre(\tau, x)) \wedge (post(\tau, x) \Rightarrow S))$

That is, if a function call instantiates this quantifier, the precondition R of the *spec* satisfies the precondition of f and the postcondition S of the *spec* 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 callsite.

4.5 Soundness

Based on the decision procedure and the verification rules described in the previous sections, it is possible to show that verified programs evaluate to completion without getting stuck. While annotated assertions are not directly enforced by the operational semantics, the preconditions of operators have to be satisfied and can

be arbitrarily complex. Therefore, this soundness property also ensures that annotated assertions, such as postconditions, hold during evaluation for concrete values of free variables.

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

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

Proof. All VCs in the verification rules shown in Figures 8 are implications of the form $\langle P'' \Rightarrow Q'' \rangle$. 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 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 propositions does not include evaluation and treats terms $\tau(\tau)$ as uninterpreted symbols rather than function calls. However, for a proof of verification soundness it is necessary to establish equalities about function application for a given closure and argument value.

Axiom 6. If $(\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x], e) \longrightarrow^* (\sigma', y)$ and $\sigma'(y) = v$ then $\vdash \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle(v_x) = v$.

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

Axiom 7. Iff $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x] \models R$ then $\vdash pre(\langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, v_x)$.

Axiom 8. If $\vdash \sigma : Q_1$ and $Q_1 \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e : Q_2[\bullet]$ and $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x] \models Q_2[f(x)] \wedge S$ then $\vdash post(\langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, v_x)$.

Axiom 9. If $\vdash \sigma : Q_1$ and $Q_1 \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e : Q_2[\bullet]$ and $\vdash post(\langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, v_x)$ then $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S\ \{e\}, \sigma\rangle, x \mapsto v_x] \models Q_2[f(x)] \wedge S$

Based on these axioms, definitions and verification rules, it can now be shown that verifiable expressions evaluate to completion without getting stuck, i.e. all reachable configurations either terminate normally or can be further evaluated.

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

Proof. Due to the complex quantifier instantiation of the decision procedure, we first show verification safety for a similar but undecidable verification judgement without quantifier instantiation. With theorem 3, soundness of this verification judgement also implies soundness with trigger-based quantifier instantiation. The verification safety proof for this alternate judgement uses a standard progress/preservation proof strategy, where the notion of verifiability is extended to stack configurations. Here, a given runtime stack is considered verifiable if, at each stack frame, the expression is verifiable with the translation of σ as precondition. A complete proof is available at: https://github.com/levjj/esverify-theory/. □

$$
\begin{array}{lll}
c \in \text{ClassNames} & fd \in \text{FieldNames} & this \in \text{Variables} \\
v \in \text{Values} & ::= \ldots \mid C(\bar{v}) & \\
e \in \text{Expressions} & ::= \ldots \mid \text{let } y = \text{new } C(\bar{x}) \text{ in } e \mid \text{let } y = x.fd \text{ in } e & \\
\tau \in \text{Terms} & ::= \ldots \mid \tau.fd \mid C(\bar{\tau}) & \\
P \in \text{VCs} & ::= \ldots \mid fd \text{ in } \tau \mid \tau \text{ instanceof } C \mid & \\
 & \quad access(\tau) \mid \forall x.\{access(x)\} \Rightarrow P & \\
D \in \text{ClassDefs} & ::= \text{class } C(\overline{fd}) \text{ inv } S &
\end{array}
$$

$$
\frac{x \in FV(P) \quad y \notin FV(P) \quad \langle P \Rightarrow fd \text{ in } x\rangle \quad P \wedge y = x.fd \vdash e : Q}{P \vdash \text{let } y = x.fd \text{ in } e : \exists y.\ y = x.fd \wedge Q}
$$

$$
\frac{\begin{array}{c}\overline{x \in FV(P)} \quad y \notin FV(P) \quad \text{class } C(\overline{fd}) \text{ inv } S \in \overline{D} \\ \langle P \wedge this = C(\bar{x}) \wedge this \text{ instanceof } C \Rightarrow S\rangle \\ P \wedge y = C(\bar{x}) \wedge y \text{ instanceof } C \vdash e : Q\end{array}}{P \vdash \text{let } y = \text{new } C(\bar{x}) \text{ in } e : \exists y.\ y = C(\bar{x}) \wedge y \text{ instanceof } C \wedge Q}
$$

Figure 9: Extending the verification rules of λ^S with simple immutable classes with class invariants.

4.6 Extensions

The core language λ^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 λ^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 segmentation logic and dynamic frames [Smans et al. 2009].

Additionally, λ^S can be extended to support "classes" as shown in Figure 9. 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.

The class invariant has to be instantiated for concrete instances of the class, so a trigger $access(x)$ is inserted into verification conditions at each field access, similarly to $call(x)$ trigger for function calls. However, unlike function definitions, class definitions $\overline{D}$ are global. Therefore, we augment verification conditions such that for each class $C(\overline{fd})$ inv $S \in \overline{D}$ the following quantifier is assumed:

$$\forall x.\{access(x)\} \Rightarrow \left(x \text{ instanceof } C \Rightarrow \left(\overline{x \text{ has } fd} \wedge S[this \mapsto x]\right)\right)$$

This quantifier is instantiated by an $access(x)$ trigger and the instantiated formula includes both the class invariant as well as a description of its fields.

5 COMPARISON WITH REFINEMENT TYPES

Despite being dynamically typed, the verification rules shown in Figure 8 resemble static typing rules. In this section, we provide a brief comparison of this program verification approach with static type checking.

$$
\begin{array}{rcl}
t \in \text{TypedExpressions} & ::= & \ldots \mid \text{let } f(x:T):T = t \text{ in } t \\
T \in \text{Types} & ::= & \{\, x : B \mid R \,\} \mid x:T \to T \\
B \in \text{BaseTypes} & ::= & \text{Bool} \mid \text{Int} \\
\Gamma \in \text{TypeEnvironments} & ::= & \emptyset \mid \Gamma, x:T
\end{array}
$$

$\boxed{\Gamma \vdash t : T}$

$$
\frac{\begin{array}{c} x, f \notin dom(\Gamma) \quad FV(T_x) \subseteq dom(\Gamma) \quad FV(T) \subseteq dom(\Gamma) \cup \{x\} \\ \Gamma, x:T_x, f:(x:T_x \to T) \vdash t_1 : T_1 \qquad \Gamma, x:T_x \vdash T_1 <: T \\ \Gamma, x:T_x, f:(x:T_x \to T) \vdash t_2 : T_2 \end{array}}{\Gamma \vdash \text{let } f(x:T):T = t_1 \text{ in } t_2 : T_2} \text{ T-FN}
$$

$\boxed{\Gamma \vdash T <: T}$

$$
\frac{B_1 = B_2 \qquad \langle [\![\Gamma]\!] \wedge R \Rightarrow S \rangle \qquad x \notin dom(\Gamma)}{\Gamma \vdash \{\, x : B_1 \mid R \,\} <: \{\, x : B_2 \mid S \,\}} \text{ ST-REF}
$$

$$
\frac{\Gamma \vdash T'_x <: T_x \qquad \Gamma, x:T'_x \vdash T <: T'}{\Gamma \vdash (x:T_x \to T) <: (x:T'_x \to T')} \text{ ST-FUN}
$$

$\boxed{[\![\Gamma]\!]}$

$$
\begin{array}{rcl}
[\![\emptyset]\!] & \stackrel{\text{def}}{=} & \text{true} \\
[\![\Gamma, x:T]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!] \wedge [\![x:T]\!]
\end{array}
$$

$\boxed{[\![\tau : T]\!]}$

$$
\begin{array}{rcl}
[\![\tau : \{x : \text{Bool} \mid R\}]\!] & \stackrel{\text{def}}{=} & isBool(\tau) \ \wedge \ R[x \mapsto \tau] \\
[\![\tau : \{x : \text{Int} \mid R\}]\!] & \stackrel{\text{def}}{=} & isInt(\tau) \ \wedge \ R[x \mapsto \tau] \\
[\![\tau : (x:T_x \to T)]\!] & \stackrel{\text{def}}{=} & \text{spec } \tau(x) \text{ req } [\![x:T_x]\!] \text{ ens } [\![\tau(x) : T]\!]
\end{array}
$$

Figure 10: Selected typing and subtyping rules of a statically typed language λ^T. Functions are annotated with types where refinements R are analogous to specifications in λ^S.

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 examples, we enable a comparison of ESVERIFY with systems such as LiquidHaskell [Vazou et al. 2014] and conjecture that it is at least as expressive.

First, we assume a language λ^T similar to λ^S but with type annotations instead of pre- and postconditions. Figure 10 shows an excerpt of such a language. Here, a type is either a dependent function type or a refined base type where refinements R are consistent with specifications in λ^S.

Given such a language, the typing rule for function definitions (T-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, other free variables in refinements can break hygiene, so T-FN restricts free variables in user-provided types T_x and T accordingly.

The subtyping relation is also shown in Figure 10. Most importantly, subtyping of refined base types requires checking an implication between the refinements and it requires translating the type environment Γ to a logical formula $[\![\Gamma]\!]$, where function types translate to the function specifications with the *spec* syntax.

Intuitively, the logical implication used for refinements also extends to translated function types, so if $\Gamma \vdash T <: T'$ then for all terms τ, $[\![\Gamma]\!] \wedge [\![\tau : T]\!]$ implies $[\![\tau : T']\!]$.

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

$$
\begin{array}{l}
\text{let } f\,(g : \ (x : \{x : \text{Int} \mid x > 3\} \to \{y : \text{Int} \mid y > 8\})) : \\
\qquad\qquad (x : \{x : \text{Int} \mid x > 4\} \to \{y : \text{Int} \mid y > 7\}) \ = \ g \text{ in } \ldots
\end{array}
$$

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

```
function f (g) {
  requires(  spec(g, x => x > 3, (x,y) => y > 8));
  ensures(r=>spec(r, x => x > 4, (x,y) => y > 7));
  return g;
}
```

This program is verifiable with the quantifier instantiation algorithm described in section 4.2. The second **spec** is translated to a universal quantifier in positive position that will be lifted, introducing a free global variable x. This also exposes a $call(x)$ trigger in negative position that now instantiates the quantifier in the antecedent. The resulting proposition can now be checked without further instantiations by comparing the argument and return propositions of both functions for all possible values of x.

This suggests that the translation of λ^T to λ^S programs preserves verifiability, i.e. well-typed λ^T programs translate to verifiable λ^S programs.

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

A formal proof of this conjecture needs to take quantifier instantiation and the quantifier nesting bound into account. This introduces immense complexity for the proof and goes beyond the scope of this paper but might be addressed in future work.

Coincidentally, a sound translation of types to annotations also enables seamless interweaving of statically-typed λ^T expressions with dynamically-typed λ^S programs in a sound way. This might be a step towards a full spectrum type system that bridges the gap between verification and type checking.

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 2001], JaVerT [Fragoso Santos et al. 2018], Dafny [Leino 2010, 2013, 2017] and LiquidHaskell [Vazou et al. 2017, 2014, 2018].

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 and instead focused on the utility of tools to find bugs.

JaVert [Fragoso Santos et al. 2018] is a more recent program verifier for JavaScript. It supports object-oriented programs but, in contrast to ESVERIFY, does not support higher-order functions. Other related work on static analysis of JavaScript include Loop-Sensitive Analysis [Park and Ryu 2015], the TAJS Type analyzer for JavaScript [Andreasen and Møller 2014], and type systems such as TypeScript, Flow and Dependent JavaScript [Chugh et al. 2012].

esverify follows as different approach as it relies on manually annotated assertions that are generally more expressive than types.

Dafny [Leino 2010] seeks to provide a full verification language with support for both functional and imperative programs. Dafny offers comprehensive support for verified programming, such as ghost functions and parameters, termination checking, quantifiers in user-supplied annotations, and reasoning about the heap. However, in contrast to esverify and LiquidHaskell, Dafny requires function calls in an assertion context to satisfy the precondition instead of treating these as uninterpreted calls. Therefore, Dafny does not support higher-order proof functions such as those shown in Section 2.8. Additionally, quantifier instantiation in Dafny is often implicit and based on heuristics, which often results in a brittle and unpredictable verification process.

In trying to find a compromise, with predictable checking but also a larger scope than traditional type systems, LiquidHaskell is most closely related to esverify. In fact, the refinement type system discussed in Section 5 loosely resembles its formalization by Vazou et. al. [Vazou et al. 2014]. More recently, LiquidHaskell introduced *refinement reflection* [Vazou et al. 2018], which enables external proofs in a similar way as the **spec** construct in esverify, and *proof by logical evaluation* which is a close cousin to the quantifier instantiation algorithm in Section 4.2 but is not based on triggering matching patterns. In contrast to LiquidHaskell, esverify is not based on static type checking and thus also supports dynamically-typed programming idioms such as dynamic type checks instead of injections.

Finally, trigger-based quantifier instantiation, as used by the decision procedure described in Section 4.2, 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 esverify 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 esverify, a program verifier for dynamically-typed JavaScript programs. esverify 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 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 esverify 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 regions or 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

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *POPL '11*.

Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *OOPSLA '14*.

Clark Barrett and Sergey Berezin. 2004. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *CAV'04*.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV'11*.

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *SPIN'12*.

Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent Types for JavaScript. In *OOPSLA '12*.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08*.

Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. 2016. Adding decision procedures to SMT solvers using axioms with triggers. *Journal of Automated Reasoning* (2016).

ECMA-262. 2017. *ECMAScript 2017 Language Specification* (6 / 2017 ed.).

M.D. Ernst, J. Cockrell, William G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on* 27, 2 (Feb 2001), 99–123. https://doi.org/10.1109/32.908957

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *PLDI'02*.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI '93*.

José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *POPL'18* (2018).

Carlo Alberto Furia and Bertrand Meyer. 2010. Inferring Loop Invariants Using Postconditions. In *Fields of Logic and Computation*.

Yeting Ge and Leonardo de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *CAV'09*.

Johannes Henkel and Amer Diwan. 2003. Discovering Algebraic Specifications from Java Classes. In *ECOOP 2003 Object-Oriented Programming*, Luca Cardelli (Ed.). Lecture Notes in Computer Science, Vol. 2743. Springer Berlin Heidelberg, 431–456. https://doi.org/10.1007/978-3-540-45070-2_19

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *TOPLAS* (2010).

K. Rustan M. Leino. 2001. Extended Static Checking: A Ten-Year Perspective. In *Informatics - 10 Years Back. 10 Years Ahead.*

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10).*

K. Rustan M. Leino. 2013. Developing Verified Programs with Dafny. In *International Conference on Software Engineering, (ICSE'13).*

K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* (2017).

K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *CAV'16*.

B. Liskov and L. Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *PLDI '88*.

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *TOPLAS* (1979).

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *ECOOP'15*.

Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. 2013. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *CADE'13*.

Damien Sereni and Neil D. Jones. 2005. Termination Analysis of Higher-order Functional Programs. In *APLAS'05*.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Workshop on Scheme and Functional Programming*.

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP 2009*.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *TAP'08*.

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq. *International Symposium on Haskell*.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *ICFP'14*.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan Scott, Ryan Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. In *POPL'18*.