UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title

Towards Live Programming Environments for Statically Verified JavaScript

Permalink

https://escholarship.org/uc/item/6x85w65n

Author

Schuster, Christopher

Publication Date

2018

Copyright Information

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

SANTA CRUZ

**TOWARDS LIVE PROGRAMMING ENVIRONMENTS
FOR STATICALLY VERIFIED JAVASCRIPT**

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Christopher Schuster

December 2018

The Dissertation of Christopher Schuster
is approved:

_____

Professor Cormac Flanagan, Chair

_____

Professor Charlie McDowell

_____

Professor Robert Hirschfeld

_____

Gilad Bracha, Ph.D.

_____

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

vii

List of Tables

List of Listings

Abstract

Towards Live Programming Environments for Statically Verified JavaScript

by

Christopher Schuster

This dissertation includes contributions to both live programming and program verification and explores how programming environments can be designed to leverage benefits of both concepts in an integrated way.

Programming environments assist users in both writing program code and understanding program behavior. A fast feedback loop can significantly improve this process. In particular, live programming provides continuous feedback for live code updates of running programs. This idea can also be applied to program verification. In general, verifiers statically check programs based on source code annotations such as invariants, pre- and postconditions. However, verification errors are often hard to understand, so programming environment integration is crucial for supporting the development process.

The research for this dissertation involved the implementation of ESVERIFY, a program verifier for JavaScript, as well as prototype implementations of multiple programming environments. These implementations demonstrate potential benefits and limitations of proposed solutions and enable empirical evaluation with case and user studies. Additionally, the proposed designs were formally defined in order to explain the core idea in a concise way and to prove properties independent of concrete specifics of existing systems and programming languages.

The resulting systems represent possible solutions in a vast design space with various contributions. The research on live programming showed that a programming model that separates event handling from output rendering enables not only live code updates but also runtime version control and programming-by-example. For program verification, ESVERIFY represents a novel approach for static verification of both higher-order functional programs and dynamically-typed programming idioms. ESVERIFY can verify nontrivial algorithms such as MergeSort and a formal proof in the Lean theorem prover shows that its verification rules are sound. Finally, a programming environment based on ESVERIFY supports inspection and live edits of verification conditions including step-by-step debugging of automatically generated tests that serve as executable counterexamples. As part of a user study, participants used these features effectively to solve programming tasks and generally found them to be helpful or potentially helpful.

Acknowledgments

I owe my deepest gratitude to my advisor, Cormac Flanagan, for his ongoing support, guidance, and for always being available for discussions. I am proud to be one of his students.

I am also very thankful to my mentor, Charlie McDowell, and his helpful advice – not only for my research but also for teaching – and I want to thank Robert Hirschfeld and Gilad Bracha, for serving on my committee and providing useful feedback.

Additionally, I want to thank my lab mates at the Software and Languages Research Group at UCSC. I will fondly remember our discussions about programming languages, technology, games, politics and metaphysical topics. In particular, Thomas Schmitz answered many of my questions about theorem proving, Dustin Rhodes often surprised me with his pragmatic and unconventional perspective, Sohum Banerjea never shied away from debates on any topic, and Tim Disney is one of the reasons why JavaScript became the basis for this dissertation.

Furthermore, I also want to thank important colleagues and mentors that supported me during internships, especially Marcelo Siero, Dan Ingalls and Robert Krahn.

During my years in Santa Cruz, I consider myself lucky to enjoy the friendship of Josh Pang, Saein Park and many others on this long journey.

Lastly, this thesis would not have been possible without the support and understanding of my family.

The text of this dissertation includes passages of the following published articles:

- Chapter 2 contains material adapted from C. Schuster and C. Flanagan, "Live Programming for Event-Based Languages", Reactive and Event-based Languages and Systems Workshop, REBLS 2015, Pittsburgh, PA, USA, as well as C. Schuster and C. Flanagan, "Live Programming by Example: Using Direct Manipulation for Live Program Synthesis", LIVE Workshop on Live Programming Systems, LIVE 2016, Rome, Italy.

- Chapters 3 and 4 contain material adapted from C. Schuster, S. Banerjea, and C. Flanagan, "esverify: Verifying Dynamically-Typed Higher-Order Functional Programs by SMT Solving", Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA.

- Chapters 5 and 6 contain material submitted for publication at the International Conference on the Art, Science, and Engineering of Programming, ‹Programming› 2019, Genova, Italy.

The dissertation author was the primary investigator and author of these papers. The coauthors listed in these publication directed and supervised the research which forms the basis for this dissertation.

Nobody actually creates perfect code the first time around, except me.
But there's only one of me.

— Linus Torvalds

■ Chapter 1

Introduction

A computer is a universal machine in the sense that anything that can be computed at all can be implemented and executed by any Turing-complete system. The sequence of steps taken by the computer are expressed by program code written in a programming language. Different programming languages may have different formal and mental models and abstractions but ultimately have the same power for expressing computation. Due to being universal, there are infinite possible programs a computer can execute. Therefore, selecting precisely the desired program behavior requires a very precise language. In that sense, programming languages fundamentally differ from natural languages which rely on context, assumptions and cultural knowledge. This difference causes programming to be extremely difficult. In particular, it is often relatively easy to informally describe ideas about expected program behavior but tedious, frustrating and error-prone to implement this behavior in a programming language. In the end, most regular computer users do not program their computers and, instead, treat software as incomprehensible opaque products that cannot be customized to expectations. Even for professional software developers, programming has generally not become easier or

Code Edits

Feedback

Intended Behavior

Actual Behavior
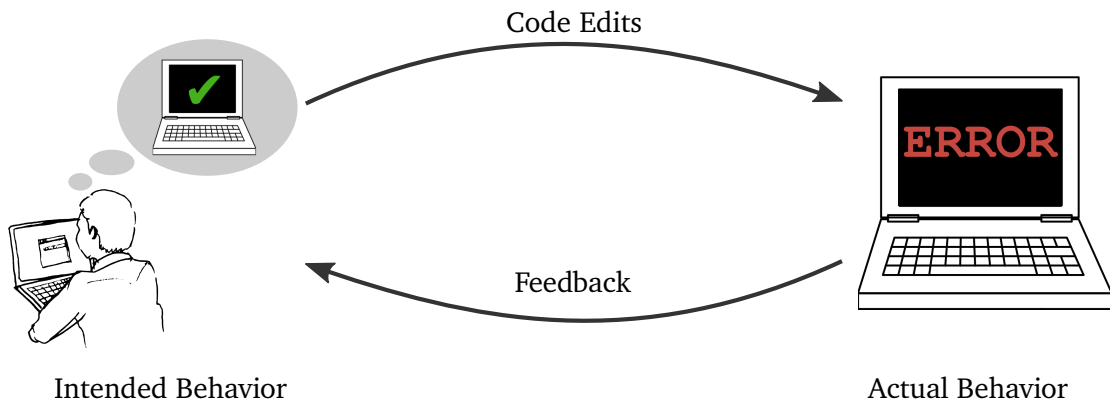
■ **Figure 1.1**: The "Gulfs of execution and evaluation" [85] explain the difficulty of using tools to achieve goals. In the context of programming environments, the user observes the program behavior in terms of output and feedback by the programming environment, compares the actual behavior with the intended goal (evaluation), modifies the program through code edits (execution) and repeats this process.

more intuitive over the last decades. To manage complexity, professional software development requires knowledge about a continuously evolving plethora of different tools, language features and abstractions. In contrast, the way programmers interact with their programming environment has changed relatively little over time and therefore offers a high potential for improvement.

As a general model for thinking about tool interaction, Norman proposes the "Gulfs of execution and evaluation" [85]. In that model, the tool user has intended goals that are executed with actions and evaluated based on feedback. A tool is easier to use if execution and feedback are more closely connected to the intended goals. Taking a water faucet as example, the goal might be a certain temperature and pressure but if two

2

provided handles control the flow of warm and cold water independently, the user needs to develop an intuition of the mixing process through repeated trial-and-error in order to achieve the intended result. An alternative tool design with a handle for temperature and a separate handle for pressure would make it easier for the user to achieve the desired goals.

Figure 1.1 shows this model applied to programming environments. Here, the programmer needs to understand the current behavior of a program and how it can be changed through code edits to match the expected behavior. The design of the programming environment should take this iterative process of execution and evaluation into account to simplify the programming process (see also Lieberman and Fry [71]). It is noteworthy that this inherent "tool problem" persists independent of progress in machine learning and artificial intelligence because the intended behavior still needs to be communicated and validated in a way that also allows the user to understand and iteratively change the program behavior.

Programming environments can support the iterative process of execution and evaluation by reducing the time gap between code edits and feedback. In particular, *live programming* aims to provide immediate and continuous feedback for code edits to running programs.

In addition to manual evaluation of the program behavior and whether it conforms with the goals and the mental model of the programmer, it is also possible to automatically check whether the program code is consistent with a second formalized description of the intended behavior. For example, by providing examples in the form

of executable tests, any discrepancy between expected test outcome and program code can be detected. Besides tests, the code can also be checked against annotations such as types, assertions and invariants. If these annotations are expressive enough to describe functional properties, this enables an automatic static *program verification*.

This dissertation includes contributions to both live programming and program verification and explores how future programming environments can be designed to leverage benefits of both live programming and program verification in an integrated way.

1.1 Live Programming

For any non-trivial application, it is unlikely that the intended behavior can be implemented correctly without any feedback or experimentation. Rather, programming is an iterative process involving trial-and-error. Programming environments can support this process by

- providing detailed feedback about the program execution, e.g. allowing the programmer to observe the state of variables and sequence of steps with a *debugger*, and by

- providing immediate or even continuous feedback for code edits, e.g. allowing the programmer to edit the code and observe effects without having to re-compile and restart the program.

These objectives are not only important for the concrete design of the program-

ming environment – the latter also affects the design and semantics of the programming language. The ability to change the code of a running application would radically change the common edit-compile-restart cycle and enable quick experimentation and exploration. As an analogy, it is significantly simpler to continuously adjust a water hose than to aim and shoot with a bow and arrow.

As Figure 1.1 illustrates, interactions with programming environments involve an iterative cycle of code edits by the programmer and feedback to evaluate how these edits affect the program behavior. Live programming is particularly promising for improving this process and supporting human understanding due to its emphasis on continuous feedback and learning by doing. The positive effects of practical experience and experimentation on the learning process have been shown empirically [51, 91] and form a core principle of computer science education [50, 54, 86]. The manual effort involved in getting feedback to a change in the code and the time delay it takes for these changes to become visible discourage quick experiments in this kind of trial-and-error setting. Essentially, the programmer should not be required to mentally simulate mundane parts of the program if the environment can do it as well. Therefore, live programming aims to provide immediate and continuous feedback without the need to switch back and forth between editing the code and running the program.

An unavoidable challenge of live programming is the inherent entanglement between execution state and program code. Code updates in the presence of these dependencies may require the programmer to specify data conversions which delays feedback and thereby hinders live programming. Possible solutions are often trade-offs that

restrict potential update points in the execution or compromise on consistency by leaving references to outdated code in the new execution state.

1.2   Program Verification

For large and complex projects, any specification of the intended behavior is prone to have errors, i.e. unintended differences with the expected behavior of the program. The (executable) program code itself is the most complete specification of the behavior, so bugs in the code often cause the program to stop working or misbehave. However, if the intended behavior is described with multiple different specifications, the programming environment can automatically check whether these specifications are consistent with each other and thereby reduce the risk of errors going unnoticed. Besides the actual program code, automatic tests, type annotations, and assertions are often used to specify expected behavior.

Since **tests** are executable programs, any discrepancy between the program code and the test assertions can be detected simply by running the tests. Therefore, failed tests can be examined with the same debugging tools that enable stepping through the code and inspecting the execution state. This integration enables programmers to obtain detailed and concrete feedback and greatly contributed to the adoption of automatic testing as a general practice in software development.

However, tests require concrete example inputs for their execution. Therefore, automatic testing can only check the program behavior for a finite set of possible inputs.

This is also true for random testing approaches such as QuickCheck [23]. Depending on the test coverage and the expectations about robustness, this limitation may be more or less severe. For example, programs that receive input from untrusted sources may need to guarantee certain security properties in the presence of adversarial inputs, while programs for visualizing data sets might be checked adequately with a few manual tests.

Figure 1.1a shows an example of a `max` function and an accompanying unit test. Here, the expected return value can be checked against a concrete result. Alternatively, it is also possible to merely check whether the result is at least as big as the input arguments.

In addition to tests, **types** are a way to constrain the possible program behavior such that variables, function arguments and function results can only be certain sets of values. Simple types are often too imprecise to specify the functional correctness of a program but type systems can still be used to ensure the absence of certain categories of errors. For example, memory safety and the absence of segmentation faults can be ensured by dynamic type checking (as in JavaScript and Python) or by a combination of dynamic and static type checking (e.g. Java and C#). With explicit type annotation in the code, the programming environment can also provide feedback about discrepancies between the stated type and the runtime values. Especially static type checking can be useful as it provides guarantees for all possible executions but, even if static type checking fails, gradual typing and dynamically-enforced *contracts* can be used to obtain helpful feedback for understanding and fixing potential bugs occurring at runtime.

The main drawback of type checking is the trade-off between expressiveness and overhead for the programmer. Primitive type systems often have a simple type syn-

```
1  function max(a, b) {
2    return a > b ? a : b;
3  }
4  function testMax() {
5    const result = max(23, 42);
6    assert(result == 42); // or following examples below: result >= 23 && result >= 42
7  }
```

(a) The unit test `testMax` checks the behavior of `max` for concrete example inputs.

```
1  /*@ max :: (a: number, b: number) => { number | v >= a && v >= b } */
2  function max(a: number, b: number): number {
3    return a > b ? a : b;
4  }
```

(b) Type annotations for arguments and result expressed as Refinement Types [117]

```
1  function max(a, b) {
2    requires(typeof a === 'number' && typeof b === 'number');
3    ensures(result => result >= a && result >= b);
4    return a > b ? a : b;
5  }
```

(c) Annotated pre- and postconditions for program verification with ESVERIFY

■ **Listing 1.1**: Three different approaches for describing and checking correctness properties: testing, type checking and program verification.

tax, support subtyping and automatic type inference and provide helpful error messages but are incapable of expressing functional correctness properties. Rich type systems with dependent function types, type refinements and effects are expressive enough to more precisely specify the intended program behavior but often require explicit annotations and a mental model of the intricate type checking algorithm in order to comprehend the error messages and the limitations of type inference and subtyping.

Figure 1.1b shows an example of a `max` function with type annotations. While

the annotation in lines 2 and 3 use basic types as common in TypeScript [12], the comment in the first line specifies a more precise return type that establishes a minimum bound for the result with a refined type [117].

As a third alternative, the intended program behavior can also be described with **assertions**, i.e. logical propositions about values and variables in the code. Checking of assertions is closely related to type checking. However, while type checking is based on a separate syntactic class of types and typing rules that compare, infer and manipulate these types, assertion checking is about determining the truth of a proposition in logic. Depending on the assertion language, different methods can be used to check assertions statically or dynamically. If assertions are simple boolean expressions, they can be checked at runtime by simply evaluating these expressions as part of the execution of the main program. However, assertions involving higher-order functions cannot be checked immediately for all possible values, so dynamic checking of function arguments and results has to be deferred until these values are available (similarly to *contracts*). Static checking of assertions, i.e. *program verification*, is comparable with rich and expressive type systems, as it enables the programmer to express functional correctness properties and check whether the program code conforms to these properties but it also has similar disadvantages in terms of overhead and complexity. In particular, the program verifier requires annotations such as pre-, postconditions and invariants in order to prove that an assertion is valid for all values, and the programmer has to know the basic verification process in order to understand and fix verification issues.

Figure 1.1c shows an example of a `max` function with annotated pre- and post-

9

condition. By assuming the preconditions, a program verifier such as ESVERIFY can prove that the postcondition holds for all possible values of a and b.

1.3   Research Goal and Method

Both live programming and program verification are well-known concepts that have been studied and applied for decades. The idea of live programming can be traced back to systems with self-contained development tools such as Smalltalk [40] but there is also renewed interest with software projects such as Swift Playground and Elm [25] as well as academic research conferences such as the LIVE workshop. Similarly, program verification and formal methods in general have been the topic of extensive prior research with impressive achievements in the last years such as a verified C compiler [67] and a verified microkernel [56]. Even for non-academic projects, verified programming languages such as Dafny [61] are increasingly adopted to verify correctness properties of commercial software.

However, the integration of program verification into development environments is a relatively new topic with many open questions. The complexity of the verification process and the resulting usability issues for the programmer are still major obstacles for program verification. While live programming is generally orthogonal to static checking, the live programming perspective seems to provide an ideal approach for exploring novel solutions with fast feedback cycles. The central question of this dissertation research is therefore:

"How can live programming environments support verification to provide a

better programming experience?"

There is a vast design space of possible solutions to this question. A comprehensive evaluation of a possible solution would require a stable and mature implementation of both a program verifier and a live programming environment. Unfortunately, these tools require immense engineering effort to create. For example, the projects mentioned above generally have teams of full-time developers or active open source communities. In order to explore and evaluate possible solutions as part of this dissertation research, I followed an approach that instead relies on prototype implementations and simplified formal models.

The **prototype implementations** created as part of this dissertation are not ready for productive use on real-world projects. However, they allow experimentation with smaller code examples, and, by using JavaScript as both source and object language, they enable live online demos that can be used in a browser without local installation. These demos illustrate with practical examples, the potential benefits and limitations of solutions to a wider audience.

Furthermore, **formal definitions** summarize the core ideas of the proposed solutions in a concise way. By omitting orthogonal concepts of actual programming languages, the formalism clarifies how the approach can be applied to a wide range of languages.

In addition to sketching potential solutions with prototype implementations and formal definitions, the proposed design also has to be evaluated in order to determine

its properties and to compare it with other solutions. Here, prototypes can be used for limited case studies such as implementations of well-known algorithms, as well as user studies with participants who solve smaller programming tasks with the tool and provide feedback about their experience. Additionally, a formal definition also enables formal proofs of properties such as soundness.

1.4   Outline and Contributions

Chapter 2 presents an approach for live programming in the context of graphical user interfaces (GUIs), and describes how this approach enables back-in-time debugging and programming-by-example. As explained in Section 1.1, live code updates may cause issues due to references and dependencies between execution state and code. While these dependencies cannot be avoided completely, a programming environment can provide a better support of live programming by restricting the programming model such that event-handling is separated from output rendering. This approach allows live code updates in-between events and back-in-time debugging while re-rendering the output at each step to provide immediate and up-to-date feedback. Additionally, this approach also enables a form of programming-by-example such that code edits are automatically inferred from output examples or direct manipulation of the user interface. In addition to a concrete prototype implementation, the chapter also presents a cursory formalization of these concepts.

Chapter 3 introduces ESVERIFY, a program verifier for JavaScript based on SMT

solving. ESVERIFY is able to verify both higher-order functions and dynamically-typed programming idioms in JavaScript and is designed with the goal of ensuring quick feedback and a comprehensible verification process. In particular, ESVERIFY explicitly avoids automatic inference procedures and heuristics that are common in other program verifiers. For example, it uses a custom trigger-based quantifier instantiation algorithm that is more robust, predictable and simpler than default instantiation heuristics in SMT solvers. As a result, ESVERIFY requires more explicit annotations to verify some programs but verification errors are easier to understand and inspect with a programming environment.

Chapter 4 includes a formal development of program verification that follows the ESVERIFY approach. The formalism defines program verification for λ^S, a pure functional core language. The verification rules involve verification conditions (VCs). Quantifiers in these VCs are instantiated according to a custom quantifier instantiation algorithm and validity of the resulting propositions is based on an axiomatization of the SMT solver. Based on this formalism, it can be shown that verification is decidable, i.e. verification terminates for all input programs, and sound, i.e. verified programs adhere to their specification during evaluation without getting stuck.

Chapter 5 describes how verification errors in ESVERIFY can be turned into executable tests that serve as concrete witnesses. Error messages based on the VC itself are often not detailed enough to explain verification errors to the programmer. However, by generating an executable counterexample based on the output of the SMT solver, the verification error can be inspected in terms of concrete variable values and its execution can

be debugged step-by-step. The test generation therefore represents a basis for further integration into a programming environment. Additionally, some verification errors are caused by limitations of the verifier and insufficient annotations rather than bugs in the code. Automatically generated tests can assist the programmer in distinguishing these errors and fixing annotations.

Chapter 6 presents IDVE, a prototype of a programming environment that integrates program verification with interactive tool support. In particular, IDVE shows counterexamples with concrete values directly in the code editor and it features a verification inspector. The verification inspector displays detailed information about the VC, such as assertions and assumptions, and it also enables step-by-step debugging of the automatically generated test. Furthermore, the verification inspector also enables the programmer to edit assertions and assumptions of a VC in order to support quick feedback and experimentation similar to live programming. Thereby, the verification inspector allows programmers to explore the verifier state without manually adding `assert` statements to the code, analogous to how interactive debuggers let programmers avoid `printf` debugging. Additionally, this chapter also describes an evaluation of IDVE with an online user study with 18 participants where participants solved small programming and verification tasks and answered a survey about their experience.

Finally, Chapter 7 surveys related work and Chapter 8 concludes this thesis.

Appended to this thesis is the Lean source code of the formal development and the proved theorems in Appendix A. Additionally, Appendix B lists the programming tasks

14

that were part of the user study[1] and Appendix C includes a record of all survey answers by participants in the user study.

To summarize, this dissertation research includes the following contributions:

- a live programming environment for event-based GUI applications based on a programming model that separates event handling from output rendering,

- an integration of live programming with programming by example based on direct manipulation of the string literals in the user interface,

- an approach for verifying dynamically-typed JavaScript programs with higher-order functions,

- a bounded quantifier instantiation algorithm that enables trigger-based instantiations without heuristics or matching loops,

- a prototype implementation called ESVERIFY,

- a formalization of the verification rules and a proof of soundness in the Lean theorem prover,

- an extension for ESVERIFY that automatically generates test cases for failed verification conditions with synthesis of function values and assertion-violating calls,

- an integrated development and verification environment (IDVE) with a novel interactive verification inspector and debugging interface, and a

---

[1]An archived version of the user study is available at https://esverify.org/userstudy-archived.

- user study to evaluate whether and how IDVE assists with simple programming and verification tasks.

If there is any delay in that feedback loop between thinking of something and building on it, then there is this whole world of ideas which will never be.

— Bret Victor

■ Chapter 2

Live Programming Environments for JavaScript

This chapter characterizes *live programming* as a feature of programming environments, distinguishes it from other related concepts, outlines challenges and describes a live programming approach for GUI applications. This approach restricts the programming model such that event handling is separated from output rendering, and thereby enables live code changes between events while continuously updating the rendered output. Besides a concrete implementation of a live programming environment for JavaScript and a language-independent formal definition, this chapter also presents *live programming by example* as an extension to live programming that infers code updates from output examples and by direct manipulation.

2.1 Introduction

The term *live programming* is used in different contexts with no clear consensus on its definition. Live programming can denote the act of programming as part of a live art or music performance in front of an audience. Another use of the term live programming

involves a programming environment that executes parts of a program while displaying intermediate results alongside the code similar to interactive workspaces/notebooks or 'playgrounds'. However, this research focusses on live programming as the ability to change the code of a running application without restarting it (also known as *hot-swapping*) and providing immediate feedback about these code changes.

Live programming is closely related to Dynamic Software Updating (DSU) [31, 47, 39, 98] but focusses on development tool support rather than patching of production systems. Ambiguities and limitations of live code updates are not catastrophic failures for live programming environments. Instead, live programming favors a 'best effort' approach that minimizes the overhead for the development process. In contrast, DSU requires programmers to precisely specify data migrations in order to resolve ambiguities and ensure persistency of production data.

Furthermore, live programming goes beyond self-contained development environments with live code updates, such as Smalltalk [40] and LISP [93]. In live programming environments, immediate and continuous feedback about code updates is not a side effect but a primary goal. In particular, this often involves re-execution of the changed code as well as dependent parts of the program to see the effects of the code update.

For a more comprehensive discussion and survey or existing work on live coding, live programming and exploratory programming, see Rein et al. [88].

Using an arbitrary programming language with function values and heap references as an example, Figure 2.1 shows the entanglement between source code and execution state. Live programming involves runtime updates to the code while retaining
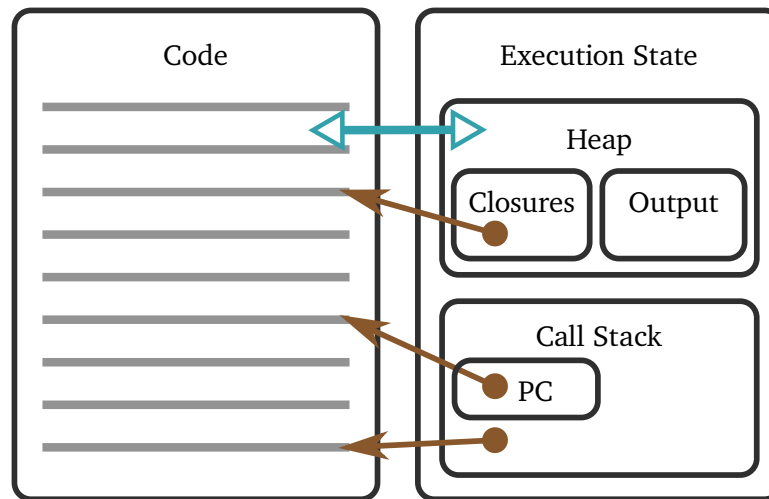
Figure 2.1: The execution state of a running program is inherently linked to the source code, so code updates have to resolve direct links (shown in brown) and potential mismatches between declarations and runtime data (shown in teal).

parts of the execution state, so each of these dependencies poses a question:

- At which point does the execution, i.e. the program counter (PC), resume if the currently executed statement or expression is changed?

- What happens to the current function call stack if the corresponding functions are removed from the code?

- How are closures maintained if the surrounding code is modified?

- Can the currently visible output be updated to reflect the new version of the rendering code?

- What happens if the format and shape of heap-allocated objects is changed in the source code?

A programming system with support for live programming has to answer these questions and address potential inconsistencies. This often involves trade-offs such as restricting live programming to certain fragments of the source code, delimiting potential update points in the execution, or leaving references to outdated code in the new execution state.

2.2 Separating Rendering from Event Handling

It has been shown that certain application architectures enable live programming for a large range of program edits without manual data migration. A live programming system with continuously updated feedback was first presented by Hancock [43] and further explored by systems like SuperGlue, which uses dynamic inheritance and explicit FRP signals [74].

Burckhardt et al. and McDirmid presented a solution to these challenges for reactive event based systems [14, 75]. In contrast to DSU systems, which guarantee safe updates with the cost of manual data conversions, a *best effort* approach better suits the goals of live programming. To prevent outdated code in closures and avoid complicated transformations of these closures, the global application state is restricted to not contain any function values. Furthermore, if the code for rendering output is separated from event handling code, it can be evaluated continuously to keep the displayed output up-to-date.

More recent work outlined the possible design space between live programming

```
1  <span id="val">Count: 0</span>
2  <button id="incButton">+</button>
3  <script>
4    var count = 0;
5    $("#incButton").on("click", function () {
6      count++;
7      $("#val").html("Count: " + count);
8    });
9  </script>
```

Listing 2.1: A simple counter implemented with jQuery DOM manipulation and imperative event handling.

systems that resume computation (with a possibly inconsistent state) and systems that record and replay execution [75], as well as introducing *managed time* as a concept for supporting both live programming and time travel [76].

This section exemplifies the obstacles for live programming, introduces a particular architecture pattern for stateful *reactive* GUI applications that supports continuous feedback in a live programming environment, and briefly discusses ways to enforce this programming model statically and dynamically.

2.2.1   Traditional/Imperative GUI Programming

To illustrate the approach, it is useful to first consider a 'traditional' programming style for GUI applications.

Listing 2.1 shows a jQuery implementation of a graphical 'click counter' as a minimal example of an interactive application. The user interface consists of a text span to display the current value of the counter, as well as a button. Here, the jQuery function

'$' is used to select elements in the browser document object model (DOM) based on their `id` attribute, attach event handlers and manipulate the DOM. The JavaScript code uses a global variable `count` and registers an event handler for the button such that every click increments the count and updates the display.

Independent of coding style, this imperative way of changing the user interface and registering event handlers poses a significant obstacle for live programming.

As an example, renaming "Count:" to "Clicks:", involves code changes to lines 1 and 7. In a live programming environment, these changes should ideally update the registered event handlers, which are closures stored in the DOM state, *and* update the visible output of the program *without* modifying the application state, i.e. the variable "`count`" should remain unchanged. However, this essentially involves removing the obsolete event handlers from the DOM, registering the new event handlers (re-evaluating line 5) *and* updating the output (re-evaluating line 7) *without* mutating the state (*not* re-evaluating line 6). Due to the structure of the code, the only choice is to update the event handlers without re-evaluating them, thereby displaying an inconsistent output until the next `click` event, or to restart the whole application, thereby losing the current value of `count` and losing the benefits of live programming.

2.2.2 Model-View-Update Pattern

In order to solve the problems outlined above, a program can be separated into three parts: a *model*, a *view* and an *update* component. This architecture is similar to the well-known and popular Model-View-Controller (MVC) pattern [58].

Since its origin in the context of Smalltalk, MVC has been adapted and applied in different ways for building user interfaces for both desktop and web applications.

Following the usual object-oriented design methodology, the model, the view and the controller in MVC are sets of objects that fulfil different roles of the application and adhere to the principle of separation of concerns. The *model* is responsible for domain logic and state, the *view* is the visible user interface and the *controller* reacts to user input by changing the model, the current view, and its own state. It is noteworthy that model, view and controller all have internal state which has to be kept consistent with imperative updates due to the lack of declarative dependency definitions.

The Model-View-Update pattern (MVU) is an adaptation of MVC that incorporates ideas from functional and reactive programming in order to provide a more declarative programming style [105]. In contrast to MVC, the *update* component only operates on the model and cannot directly access the view while the *view* only generates a user interface without maintaining internal state. Therefore, both the view and the update component are stateless and can be understood as pure functions. Essentially,

- the **model** encapsulated the entire application state (including the state of the UI),

- the **view** returns the output/user interface based on the current model, and

- the **update** component processes events, yielding a new model.

It is noteworthy that the separation of model, view and update also applies to reuse of code as part of a library or larger application. While component-based UI

```
1   /* Model (in this case a simple global variable) */
2   let count = 0;
3
4   /* Update (functions that mutate the model) */
5   function inc() {
6     count++;
7   }
8
9   /* View (a pure function that generates and returns the user interface) */
10  function render() {
11    return (
12      <div>
13        <span>Count: {count}</span>
14        <button onclick={inc}>Inc!</button>
15      </div>
16    );
17  }
```

■ **Listing 2.2**: Simple counter example in Listing 2.1 rewritten as MVU application with JSX/React-style view.

systems use a single component object or class as composable and reusable abstraction, MVU requires the model, view and update to be composed separately.

There are variations of MVU and other patterns for UI architectures that advocate unidirectional data flow [105] which have recently become popular for developing web applications such as Redux, Ur/Web [16], the Elm Architecture[1], and reactive programming in general [13, 52, 89, 92]. However, the approach in this chapter focusses on the MVU pattern outlined above due to its simplicity relative to more sophisticated solutions.

Listing 2.2 illustrates how the example in Listing 2.1 can be rewritten with the

---

[1]The Elm Architecture: https://guide.elm-lang.org/architecture/.
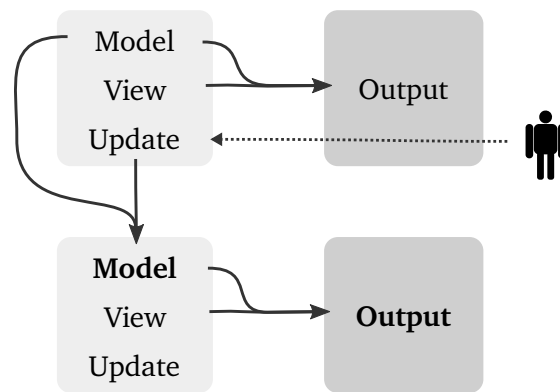
■ **Figure 2.2**: In the MVU pattern, interactions by the user are the basis for updates that result in a new model thereby a new/changes output.

MVU pattern. Here, the model simply consists of a global variable containing the current click count as integer. In more complex applications, the model would be defined in terms of user-defined classes and data types. Updates to the model are initiated by event handlers that are declaratively bound to elements of the UI. In this case, the `inc` function is bound to the button in line 14 and simply mutates the model by incrementing the count. Finally, the view is a top-level `render()` function that may invoke other view functions and returns a tree structure of the output. The details of how this tree structure is defined and constructed is application-specific and insignificant for live programming. In Listing 2.2, inline XML/HTML tags are used to create the user interface (also known as JSX syntax popularized by React).

2.2.3 User Interactions in MVU Applications

Conceptually, the MVU architecture style handles user interface interactions as input event (see Figure 2.2). The output is generated by the view based on the current state

of the model. Any interaction by the user, such as clicking with the mouse or pressing a button on the keyboard, represents an input event. This event, alongside the previous state of the application, initiates an event handling procedure that results in a new and potentially changed model. The view then renders the new model by generating a new user interface. It is important to note that this re-rendering is initiated and managed by the environment or framework and not invoked by user code. This avoids redundant rendering of unchanged parts of the output and enables differential DOM updates.

2.3   Live Programming Environment Integration

The MVU architecture pattern described in the previous section has advantages for live programming environments. In particular, it enables continuous feedback for live code updates, back-in-time debugging and runtime version control.

2.3.1   Code Updates

As illustrated by Figure 2.1 in Section 2.1, any approach for live programming needs to resolve the inherent entanglement between code and execution state, including the current program counter, the function call stack and closures.

An event-based model of program execution and an MVU architecture style (with additional restrictions) avoids some of these links and thereby facilitates live programming.

First, the execution is assumed to be strictly single-threaded with a single main event-loop, as common in JavaScript. In this model, the program code is not actively

26

executed by any thread while the system is waiting for the next event. Therefore, the execution state between events does not contain a current program counter or active call stack with regards to the application code. This greatly simplifies code updates at these update points.

Second, the MVU architecture style summarizes the complete application state in the model. Therefore, the view and update components can be swapped at runtime while retaining most of the execution state.

Finally, the application state is restricted to not contain or reference function values. In some cases, it might be feasible to swap an existing function value for a newer version of the same function. However, if the original function was removed from the code, the function value might become outdated. Additionally, code updates might move or rename function definitions and thereby cause ambiguities if the identity of functions is determined solely based on the previous and updated version of the code. As a possible solution, the programming environment could model code updates as semantic actions instead of plain text edits to resolve these ambiguities. Furthermore, if a function value references variables in the surrounding scopes then even changes to the surrounding code could affect values in the environment of the closure. Conceptually, these same issues also apply to objects as instances of class definitions. However, the identity of (named) classes is less prone to be confused and methods are not usually closing over an environment. In conclusion, avoiding function values in the application state ensures that code updates do not result in ambiguity or outdated references but alternative approaches might enable live updates even in the presence of closures.

(a) Changes to the view code will re-render the output immediately

(b) Changes to the update code only affect subsequent events
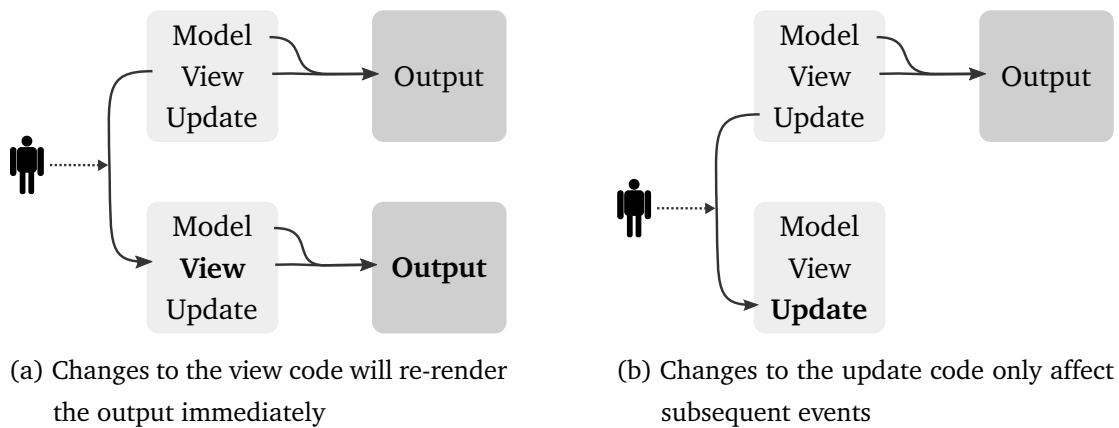
■ Figure 2.3: Live code updates in MVU applications.

Figure 2.3 depicts how the code of the view and the update component in an MVU application can be updated at runtime. It is important to note that changes to the view code result in an re-rendered output. Thereby, the programmer is able to get immediate feedback about code changes. Changes to the update component, on the other hand, only take effect at the next event to be processed. As a potential alternative, it would be possible to replay past events with the new update component.

2.3.2 Navigating Execution History and Code Versions

As described in the previous section, the MVU pattern allows the application state to be rendered on-demand with a stateless view function. In addition to the current application state, this also enables any previous state in the execution history to be displayed as a simple form of time travel ("debugging back in time" [68]).

On a high level, each step in the execution history is an event and each event handling procedure produced a corresponding model. Therefore, it is possible to use

(a) A trace of model states in MVU enables back-in-time debugging

(b) Live code updates in MVU also enable runtime version control
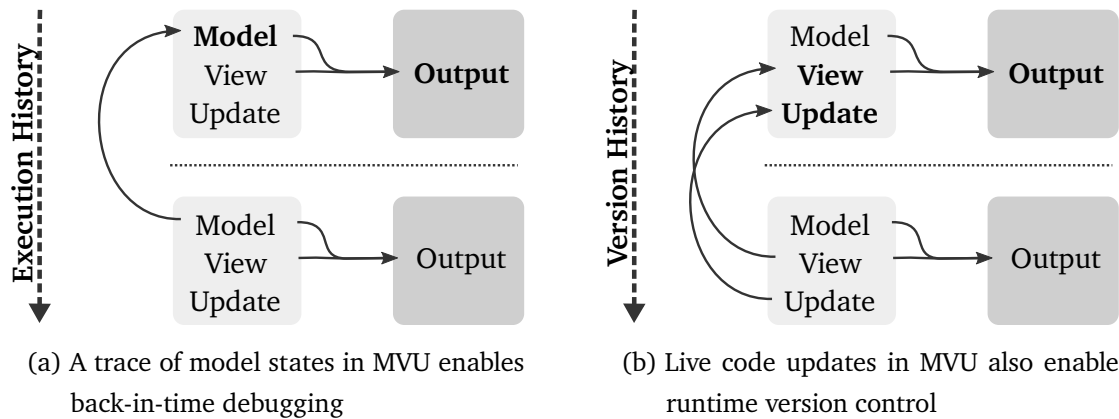
Figure 2.4: Event handling and live code updates with the MVU pattern also enable past states of the application and different versions of the code to be navigated at runtime.

the current view to visualize past states of the application as shown in Figure 2.4a. The implementation of this trace could either maintain snapshots for the model, an event log to be replayed, or a combination of both.

Similarly, the concept of live code updates can be extended to *runtime version control*. Instead of swapping the view and update code based on a source edit, the programming environment loads a different version of the code while retaining most of the execution state. Runtime version control is a common technique in self-supporting systems. For example, Monticello enables programmers to load different versions of a package into a live Squeak/Smalltalk environment [84]. In the context of MVU applications, navigating the version history has the additional benefit of continuous updates to the output as depicted in Figure 2.4b.

2.3.3 Enforcing MVU Pattern

As mentioned in Section 2.2.2, the view should not be able to alter the state of the model. In pure functional programming languages, this is trivially the case. In languages with imperative updates, such as JavaScript, the view may accidentally change the model when rendering the output. This may cause unintended effects with live code updates. For example, automatic re-rendering after a code change might corrupt the model state in an irreversible way. In that case, the programmer has to restart the application and loses the benefits of live programming. However, it is possible to dynamically and statically detect and enforce that the view does not alter the model.

For example, a dynamic contract could check that the model before and after each rendering is identical or, alternatively, an immutability proxy membrane could grant read access but prohibit mutation [113].

Instead of detecting mutation at runtime, it would also be possible to statically check the code with a type and effect system. This avoids the performance overhead of dynamic checking but might reject valid programs. While mostly preferable for languages with type annotations, static analysis methods can also check unannotated dynamically-typed languages such as JavaScript [83, 95].

Similarly, it is possible to either statically or dynamically enforce that the model does not contain or reference function values/closures as discussed in Section 2.3.1.

2.3.4 Formalism

The previous sections described a live programming approach for JavaScript. However, this approach can also be generalized and applied to other languages that have an event-based execution. This section gives a formal definition of a minimal language in order to clarify the minimal requirements for the program structure and execution and to model live code updates as system interactions alongside regular event processing.

Figure 2.5 formally defines the structure and transition relation for a generic live programming system based on the MVU pattern.

The state of the system is summarized by the application state m (*model*), rendering code v (*view*) and event handling code u (*update*), and any system transition $\langle m, v, u \rangle \xrightarrow[o]{i} \langle m, v, u \rangle$ is triggered by a system interaction i and produces output o. In particular, the system processes both regular input events q and runtime code updates while providing continuous feedback.

In this formalism, the output of the application is not generated imperatively (e.g. via `printf` statements); instead, it is summarized as a single value o (e.g. the DOM of a JavaScript application or the framebuffer of an OpenGL application). m, v, u and the output o are assumed to be values in a language with lambda abstractions and function application but the syntax definitions for values and expressions e depend on the concrete programming language implementation and are mostly left unspecified.

Regular input events such as mouse clicks are processed according to E-EVENT transition rule. The update function u is invoked with the current application state m

31

m (Model) v (View) u (Update) o (Output) $\langle m, v, u \rangle$ (System Configuration)

$$m, v, u, o, a \ ::= \ q \ | \ \lambda x.e \ | \ ... \qquad\qquad\qquad\qquad \text{(Values)}$$

$$e \ ::= \ a \ | \ x \ | \ e(e) \ | \ ... \qquad\qquad\qquad\qquad \text{(Expressions)}$$

$$q \ ::= \ [\text{keypress } a] \ | \ [\text{click } a \ a] \ | \ ... \qquad\qquad\qquad \text{(Events)}$$

$$i \ ::= \ [\text{event } q] \ | \ [\text{swap } v \ u] \qquad\qquad\qquad\qquad \text{(Interactions)}$$

$$e \downarrow a \qquad \text{(Evaluation)} \qquad \langle m, v, u \rangle \xrightarrow[o]{i} \langle m, v, u \rangle \qquad \text{(System Transition)}$$

$$\frac{u(m, q) \downarrow m' \qquad v(m') \downarrow o}{\langle m, v, u \rangle \xrightarrow[o]{[\text{event } q]} \langle m', v, u \rangle} \ \text{E-EVENT}$$

$$\frac{v'(m) \downarrow o}{\langle m, v, u \rangle \xrightarrow[o]{[\text{swap } v' \ u']} \langle m, v', u' \rangle} \ \text{E-SWAP}$$

Figure 2.5: Formal definition of a system for MVU applications. A system transition is triggered by an interaction i and produces output o. In particular, the system processes regular input events q and enables runtime updates of the view and update code while providing continuous feedback. The concrete syntax classes for values a, expressions e and the evaluation semantics $e \downarrow a$ are left unspecified.

and the event data q, yielding a new model m'. This model is then used by the view v to render the new output o.

This system also supports live programming, i.e. hot swapping of code at runtime with immediate feedback. The rule E-SWAP describes how new versions of the view v' and the update u' code are replacing the previous code v and u. The new view v' is used with the current application state m to provide feedback o about code changes.
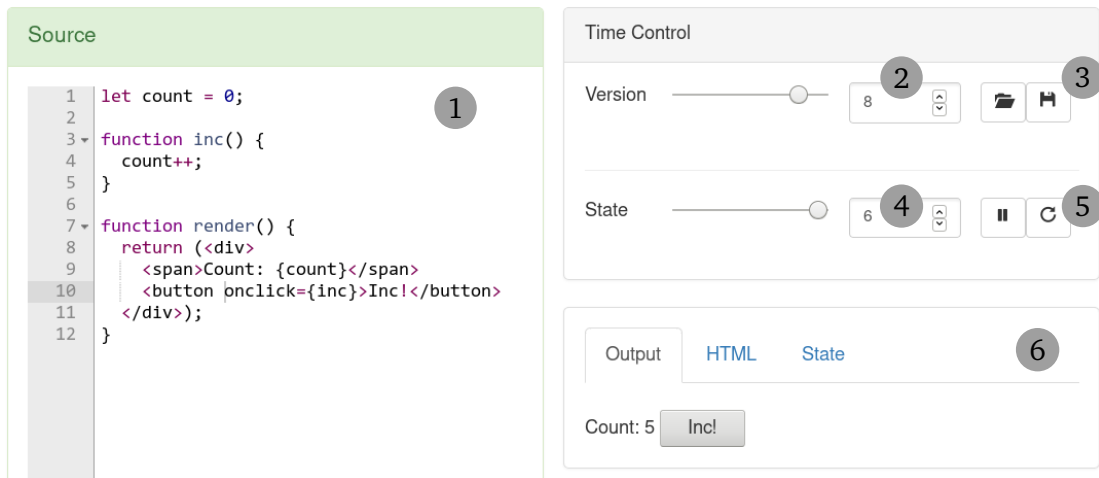
Figure 2.6: The live programming environment features an editor, a live view of the output as well as controls for traveling to previous code versions/execution states and for resetting the state to initial values.

Changes to the update code are not immediately visible but subsequent events will be processed with u' instead of u.

2.3.5 Implementation

The implementation of live programming approach as a web-based online programming environment for JavaScript involved both user interface design as well as techniques for checking conformance to the MVU programming model. Its source code[2] and a live demo[3] are both publicly available.

The environment features a code editor with syntax highlighting and a side panel for navigating the execution and version history and displaying the output. More advanced features such as project management for programs with multiple files and

---

[2]Source code at `http://github.com/levjj/rde/`
[3]Online live demo at `https://levjj.github.io/rde/`

nested folders are not currently available.

A screenshot of the programming environment is shown in Figure 2.6. The code editor ① includes the complete source code of the application. Every change made in this editor creates a new version that is appended to the version history. A slider in the "Time Control" panel ② allows the programmer to restore a previous version indexed by its revision number, and also to load and save the current file ③ . Similarly, the programmer can go go 'back-in-time' to a previous application state with slider controls ④ . Each state corresponds to an input event in the application trace. The application is assumed to be continuously running but execution can also be manually restarted ⑤ . This is particularly important if changes made to the code are incompatible with the current state or to ensure that changes to a global initializer takes effect. Finally, the third panel ⑥ enables the programmer to see the current output, examine its underlying HTML structure, and inspect the state of global variables. As discussed in Sections 2.3.1 and 2.3.2, editing the code or navigating the version and execution history causes the output to be refreshed in order to provide continuous feedback.

In addition to the user interface of the programming environment, the implementation also involved management of the MVU programming model that enable event processing and live code updates. In particular, the programming environment wraps event handlers, such as the `inc` function in Figure 2.6, in a function that automatically schedules a re-rendering of the output after each event.

Furthermore, the programming environment also ensures that there is a global `render` function returning the output as JSX tree structure. The model is assumed to

encompass all global variables, so in order to guard the model against mutation during rendering, global variable references in the program, such as `count`, are rewritten to property accesses of a state object, such as `state["count"]`. Thereby, the `render` function can be evaluated with a deep copy of the state that dynamically traps mutations with the `Object.freeze` mechanism in JavaScript, applied recursively to the entire model. These deep copies of the model are also appended to the trace of application states to enable back-in-time debugging. Finally, the programming environment also checks that the model does not contain or reference any functions by performing a recursive **typeof** test on all global variables after each event handling process.

2.4 Live Programming by Example

This section describes an extension to live programming that infers live code updates from examples provided by the user at runtime. It is based on the MVU programming model introduced in Section 2.2.2. Instead of presenting a concrete implementation, this section first describes the approach conceptually and presents a formalism. Section 2.5 then describes a live programming environment with direct manipulation of string constants as a concrete instance of *live programming by example*.

2.4.1 Live Code Updates based on Output Examples

Live programming systems as described in the previous section enable program updates at runtime for changes made to the source code. By separating rendering from event handling with the MVU pattern, the application output can be updated in reaction to

35

code changes to provide visual feedback to the developer. Expanding on this idea, the output of the program can also be used by the programmer to *make* changes to the program. This is particularly beneficial for graphical user interface programs as editing the visible output, e.g. by *direct manipulation* [101], is more convenient or intuitive than edits to the generating source code.

As illustrated in Figure 2.7, instead of editing the source code directly, the visible output can serve as a basis for expressing intended changes to the program. On the one hand, the programmer could provide an example of the intended output that is used by an inference algorithm to synthesize a view implementation. On the other hand, user interface interactions by the programmer such as direct manipulation of the output could be applied directly to the generating view code. Depending on the domain and programming language, there are various different programming-by-example techniques [70, 41] to 'repair' or 'synthesize' a new program whose rendered output partially or fully conforms to the provided output example.

2.4.2 Formal Definition

Live programming by example is applicable to a wide range of applications and programming languages. The potential benefits and limitations depend on the concrete format of the output, the available interactions by the user, and the inference algorithm. However, it is possible to model live programming by example as an abstract system transition that extends the formalism given in Figure 2.5

Figure 2.8 extends the formal definition given in Figure 2.5 with support for live
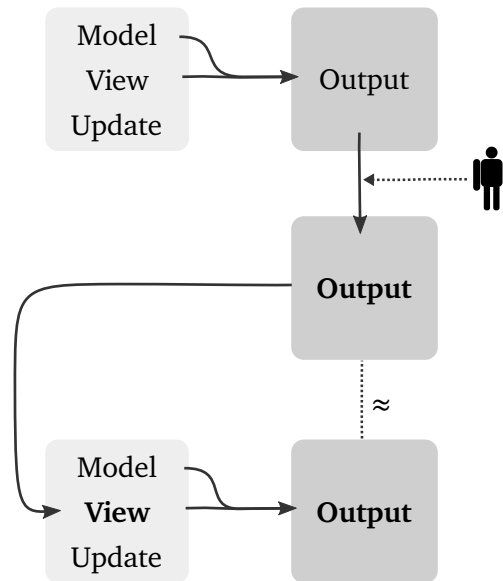
Figure 2.7: With live programming by example, the view code can be changed based on output example, e.g. by direct manipulation of the previous output. The new view code should be inferred to closely match the user-supplied output example.

programming by example.

Given output o, the developer can initiate a live code update by changing the output o into a desired output example o' that is then used for synthesis (E-EXAMPLE). The judgement $(o, m, v) \triangledown v'$ infers a modified view function v' from v such that, ideally, the output o'' generated by the synthesized render function v' exactly matches the user example ($o' = o''$). However, it may be advantageous to use a heuristic that prioritizes smaller changes to the view v and tolerates minor differences between new output and the example ($o' \approx o''$).

Apart from language details of the implementation and the concrete UI representation of the output, this definition highlights the design space for live programming

$$m \text{ (Model)} \quad v \text{ (View)} \quad u \text{ (Update)} \quad o \text{ (Output)} \quad \langle m, v, u \rangle \text{ (System Configuration)}$$

$$m, v, u, o, a \quad ::= \quad q \mid \lambda x.e \mid \ldots \hspace{4cm} \text{(Values)}$$

$$e \quad ::= \quad a \mid x \mid e(e) \mid \ldots \hspace{4cm} \text{(Expressions)}$$

$$q \quad ::= \quad [\text{keypress } a] \mid [\text{click } a\ a] \mid \ldots \hspace{3cm} \text{(Events)}$$

$$i \quad ::= \quad [\text{event } q] \mid [\text{swap } v\ u] \mid [\textbf{example } o] \hspace{2cm} \text{(Interactions)}$$

$$e \downarrow a \quad \text{(Evaluation)} \qquad \langle m, v, u \rangle \xRightarrow[o]{i} \langle m, v, u \rangle \qquad \text{(System Transition)}$$

$$(o, m, v) \triangledown v \hspace{4cm} \text{(View Inference)}$$

$$\frac{(o', m, v) \triangledown v' \qquad v'(m) \downarrow o'' \qquad o' \approx o''}{\langle m, v, u \rangle \xRightarrow[o'']{[\text{example } o']} \langle m, v', u \rangle} \text{ E-EXAMPLE}$$

Figure 2.8: Formal definition of a system that supports live programming by example as an extension to the syntax and semantics given in Figure 2.5.

by example. Both the program synthesis technique ($\triangledown$) as well as the user intent of supplied examples ($\approx$) enable a wide range of different approaches ranging from the simple manipulation of string literals to sophisticated direct manipulation interactions and end-user programming.

2.5 Live Programming by Direct Manipulation of the Output

As a concrete instance of live programming by example, this section presents an approach for editing string literals in JavaScript applications through UI manipulation. The implementation is an extension to the programming environment discussed in Section 2.3.5.

```
1  var str = "";
2
3  function keyup(evt) {
4      str = evt.target.value;
5  }
6
7  function render() {
8      return (<div>
9          <input value={str} onkeyup={keyup} />
10         <p>{str.replace(/keyboard/g,"leopard")}</p>
11     </div>);
12 }
```

■ **Listing 2.3**: JavaScript implementation of a simple interactive keyword replacer using the MVU pattern.

The approach uses a dynamic string taint analysis [118] to tracks the source location of string constants in the view code. Thereby, the environment enables the programmer to change string constants appearing in the HTML output and infer changes to the original source code. This programming environment serves as a precursor for future systems that allow synthesis of more complex live code modifications by direct manipulation. The source code[4] and a live demo[5] are integrated into the live programming environment described above.

2.5.1 Example Interaction

In order to illustrate how a programmer might use a live programming example to change the view code, this section uses a simple interactive application as an example.

---

[4]Source code at http://github.com/levjj/rde/
[5]Online live demo at http://levjj.github.io/rde/

The GUI of this application consists of an input field and a label. The label shows the entered text but replaces keywords in the input text according to a fixed rule. This example is inspired by `https://xkcd.com/1031/` and has been implemented by several browser extensions. Listing 2.3 shows an implementation. The 'model' is simply the string entered by the user: `str`. The `render` function returns the user interface and also binds an event handler that changes the model as the user types into the input field.

Since this implementation follows the MVU pattern (see Section 2.2.2), the string constant `"leopard"` in the source code can be changed to `"butterfly"` and the programming environment immediately re-renders the output accordingly.

However, instead of editing the source code directly, this code change can also be performed by direct manipulation of the output such that the generating string literal in the source code will be modified to match the intended output example.

Figure 2.9 shows how the programming environment can be used to achieve this change. First, the programming environment provides a way to halt normal execution ① and enable a special interaction mode for the application's user interface. This ensures that interactions with the UI are not handled by the application itself but by the programming environment. As an alternative to introducing a special interaction mode, it may be possible to reserve certain controls for this purpose, e.g. with a special *meta key* for direct manipulation or by using *halo controls* [73].

Different forms of direct manipulation may be available (e.g. resizing or re-ordering via drag and drop). In this example, the text of the static label (`<p>`) becomes editable by the user/programmer ② .

40
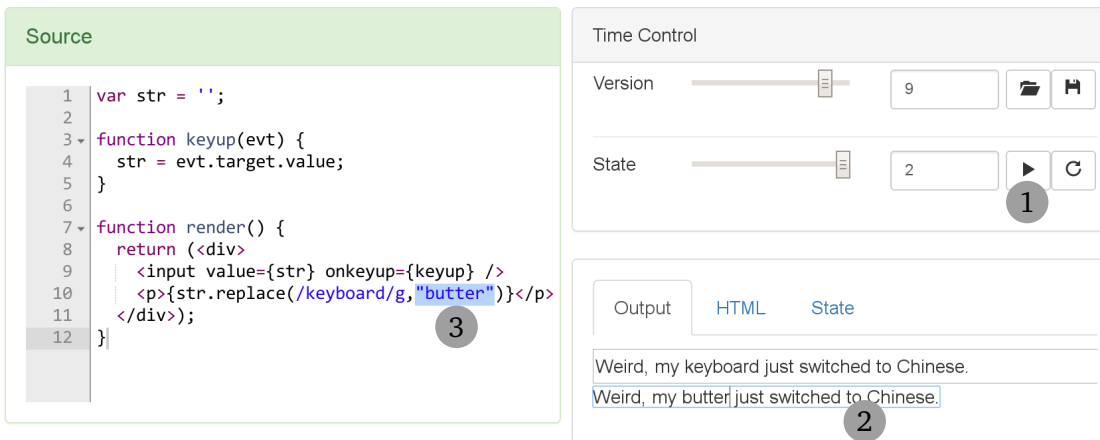
Figure 2.9: A live programming environment with support for live programming by example. Stopping the normal execution ①prevents event processing but enables direct manipulation of the UI including editing the text displayed in the output ②. Based on this UI manipulation, the corresponding string literal in the source code ③is changed automatically.

Parts of the label text `"leopard"` originate from a string literal in the program source code. Therefore, the string literal is highlighted in the editor ③and changes in the user interface will also be applied to the source code. Text parts that do not originate from source code literals cannot be modified. Any change to the source code causes the output to be re-rendered, so if the word "keyboard" would appear more than once in the input text, all of its occurrences would be replaced according to the new `render` function thereby ensuring consistency between code and output.

2.5.1.1 Dynamic String-Origin Tracking

In order to support the interaction outlined in the previous section, the environment needs a mapping of strings in the output to their generating string literals in the source code. This mapping can be obtained by instrumenting the execution such that string values are replaced by custom string objects with origin information. These string objects include a custom implementation of concatenation, substring extraction, replacement and other common string operators (e.g. `str.toLowerCase()`), such that origin information is retained but otherwise behave like regular string values. This form of dynamic analysis is closely related to taint analysis for information flow security.

As a first step of the source code instrumentation process, all string literals/constants in the code are replaced by expressions that create custom tagged string objects with both the original string value as well as a unique identifier to track the source code location.

```
1    var x = "ab";
2 -> var x = stringlit("ab", 23);
```

Additionally, built-in unary and binary operations are replaced by function calls implementing these operators.

```
1    var y = x + "c";
2 -> var y = addop(x, stringlit("c", 42));
```

While concatenation with the regular + operator behaves as expected, this instrumentation is necessary to retain the origin information. Due to concatenation (and

other operators) different parts of a string can originate from different string literals in the code. The origin information therefore includes all subparts of a string alongside the identifier of the generating string literal and offset.

```javascript
1  var x = "ab";    // [["ab",23,0]]
2  var y = x + "c"; // [["ab",23,0],["c",42,0]]
3  var z = y[1];    // [["b", 23,1]]
```

JavaScript code that is not part of the program, especially built-in/native code, is not subject to source code rewriting. To ensure correct behavior for applications passing tagged strings to uninstrumented functions, JavaScript's metaprogramming facilities such as `Proxy` and property definitions for `Symbol.iterator`, etc. are used to ensure that these tagged string objects are mostly indistinguishable from regular strings. In particular, tagged string values are wrapped in a proxy [113] that automatically converts tagged strings to primitive strings when no instrumentation is possible or necessary (e.g. for parsing strings as integers), and input events from the DOM passed to event handlers are wrapped in a proxy membrane that transparently converts primitive strings to tagged strings. While this approach preserves program behavior, it is possible for a string to lose its origin information due to built-in JavaScript functions.

2.5.1.2 Programming Environment Integration

The kinds of supported UI interactions depend on the domain and concrete representation of the output. In the context of the live programming environment for JavaScript, the output is a tree of HTML/DOM elements and attributes. With string origin tracking,

plain text content, attributes and element names can potentially contain origin information.

The programming environment shown in Figure 2.9 supports two ways of manipulating the output for the purpose of live program synthesis. The developer can either edit the raw HTML code or manipulate text in the graphical user interface. The HTML representation has the advantage that all parts of the output, including element names and attribute keys and values, can easily be modified on a textual level. Manipulation of the visible UI is limited to the plain text content of HTML elements but allows programmers to perform changes in an intuitive way on the level of the actual output.

Given a modified DOM tree, the program synthesis generally follows the following informal algorithm:

1. Determine the previous output based on the current rendering code and application state.

2. Compute the difference between the provided example and the previous output. (The modified DOM tree either has new characters inserted, existing characters removed or both[6].)

3. Check origin information of the modified characters. Modifications to string parts that do not have origin information cannot be handled and will be suppressed.

4. Determine source code location of the generating string literals using string origin information and a mapping from string literals to AST nodes.

---

[6]Changes to the tree structure of the DOM output are not currently supported and remain future work.

5. Use the source code location and computed offsets to insert or delete characters in the program source code.

6. Evaluate/recompile code and obtain a new rendering method.

7. Render the updated output using the modified rendering code and the current program state.

Listing 2.4 shows a simplified JavaScript implementation of the algorithm above when deleting characters. In this case, `"pard"` is deleted from `"leopard"` in the UI, so the `example` contains the modified UI that differs from the previous `output` as rendered with the current program view and application state. This difference (`diff`) is used to determine the originating string literal in the AST of the program and, thereby, the location in the original source code. According to the difference in the outputs, four characters were deleted, so this change is also applied to the program source code. Finally, the program gets re-compiled, yielding a new view function and the output is re-rendered.

2.6   Discussion and Future Work

This chapter outlined an approach for live programming of event-based GUI applications. In addition to live code updates, this technique also enables back-in-time debugging, runtime version control and live programming-by-example as a way to change the code of a running program including a prototype implementation for direct manipulation of string constants in the user interface.

```
1   // Global state of the programming environment
2   var source;  // "var str = ''; ... str.replace(/keyboard/g, "leopard") ... "
3   var program; // {render: function(s) { ... }, astnodes: ...}
4   var state;   // {str: "Weird, my keyboard just ..."}
5
6   // Given an example as a tree representation of the output,
7   // modify the original source code and update the environment
8   function updateByExample(example) {              // {.. "Weird, my leo just.." }
9     var output = program.render(state);            // {.. "Weird, my leopard just.." }
10    var diff = outputdiff(output, example);        // {delete: 4, offset: 23, idx: 42}
11    var origin = output.find(diff.idx).origin;     // {strlit: 3, offset: 3}
12    var astnode = program.ast.find(origin.strlit); // {srcloc: {start: 23, .. }, ..}
13    var offset = astnode.srcloc.start + origin.offset;
14    source = source.substr(0, offset)              // ... replace(/keyboard/g, "leo
15          + source.substr(offset + diff.delete);   // ") ... "
16    program = compile(source);
17    show(program.render(state));
18  }
```

■ **Listing 2.4**: Simplified algorithm for synthesizing code updates when deleting characters in the UI as shown in Figure 2.9. In this example, `"pard"` is deleted from `"leopard"`.

2.6.1 Live Programming for MVU applications

The approach relies on the MVU programming pattern with a stateless view and update component and a model that encompasses the entire application state.

Live code updates may involve added, modified or removed function definitions. Unfortunately, function values/closures in the application state cannot always be automatically transformed to match the new code, therefore closures are currently not allowed in the application state. Restricting the global state in this way limits expres-

siveness but avoid ambiguity and ensures that code updates do not result in outdated references (as discussed in Section 2.3.1). More research is necessary to evaluate the design decisions and study how programmers can benefit from live programming for other architecture patterns.

Moreover, live programming may lead to inconsistencies in the application state. If the event handling code is updated and execution resumed, input events in the execution history will have been processed by different versions of the same event handler. Therefore, the resulting application state may be inconsistent, i.e. it may differ from the application states produced by either version of the event handler for a restarted and replayed execution. As long as the application state is internally consistent, inconsistencies with past input events are mostly benign and insignificant for the programming experience. However, there could be situations in which resuming execution and replaying events is more desirable for development or debugging.

Finally, changes to the initialization code of the program are not re-evaluated and do not provide feedback to the programmer. If this change also alters the data type of the application state, the current runtime state may become incompatible with the code and require either manual data migration or a restart of the execution.

Future work might also involve common programming language techniques to improve the performance overhead, such as proxy membranes for enforcing immutability and first-order states, copy-on-write for optimizing back-in-time debugging, and incremental computation for the new output [1].

47

2.6.2 Live Programming by Example

The programming environment presented in this chapter supports a simple form of live programming by example but mainly serves as a precursor for future systems that support more sophisticated program synthesis guided by more flexible forms of direct manipulation.

With the presented approach for live programming by example, code updates to the rendering code can be synthesized and applied immediately. However, updates to the event handling code will only affect subsequent events and therefore live programming by example as presented in this chapter is not directly applicable to the event handling code. A possible solution is to replay past events instead of resuming execution with the existing state. It is not always clear how many events have to be replayed as replaying all events may not be practical or desirable for long-running applications and replaying just the last event may not suffice but future work may expand the live programming environment to enable inference for rendering code as well as event-handling code.

Changing string literals in the program by manipulating text in the output is a relatively simple implementation of live programming by example and thereby avoids ambiguities that are common in program synthesis applications[7]. However, more complex live programming by example solutions have to address potential ambiguities with heuristics or manual user intervention. For example, the programming environment might present multiple ranked synthesis candidates for the programmer to choose from.

---

[7]The string synthesis technique is still ambiguous as inserted characters between two different generating string literals can be inserted either at the end of the first string literal or the start of the next one. This ambiguity is currently resolved by always prioritizing the first string literal.

Finally, the approach still needs to be evaluated for larger applications and development tasks — potentially as part of a user study[8].

---

[8]See e.g. Campusano, Bergel, and Fabry [15] and Lieber, Brandt, and Miller [69]

In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders.

— Douglas Crockford

■ Chapter 3

Program Verification for JavaScript

This chapter describes the design and implementation of ESVERIFY, a program verifier for JavaScript. The goal of ESVERIFY is to ensure a comprehensible verification process with quick feedback as part of an integrated programming environment as well as support for dynamically-typed programming idioms in JavaScript that are not well supported by type systems. ESVERIFY is based on SMT solving but instead of using heuristics for quantifier instantiation, it uses a custom trigger-based quantifier instantiation algorithm. As a result, ESVERIFY requires more explicit annotations to verify some programs but its verification errors are easier to understand and inspect. Finally, this section also presents case studies to illustrate and evaluate how smaller JavaScript program can be verified with ESVERIFY.

3.1 Overview

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.

This chapter introduces 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 [28] 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]

---

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

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.

3.2 ESVERIFY

By supporting a subset of ECMAScript/JavaScript, a dynamically-typed scripting language, ESVERIFY is unlike existing verifiers for statically-typed programming languages. The goal of ESVERIFY is not 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.

3.2.1 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 [32], this approach enables a better integration with existing tooling support such as refactoring tools and syntax highlighters.

The assertion language is a subset of JavaScript. It does not support all of

```
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); // failing postcondition
6    if (a >= b) {
7      return a;
8    } else {
9      return a; // bug
10   }
11 }
```

■ **Listing 3.1**: A JavaScript function `max` annotated with pre- and postconditions.

JavaScript's semantics. In particular, it is restricted to pure expressions that do not contain function definitions.

3.2.2 `max`: A Simple Example

Listing 3.1 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.

3.2.3 Stateful Programs and 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 recur-

53

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

sion 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.

Listing 3.2 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[3]. 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

---

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

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 [35]. Recent research suggest that automatic inference can also be extended to program invariants [30] and specifications [45]. However, this topic is orthogonal to the program verification approach presented in this thesis.

3.2.4   Higher-order Functions

In order to support function values as arguments and results, ESVERIFY introduces a `spec` construct in pre-, postconditions and assertions. Listing 3.3 illustrates this syntax in lines 7 and 8 of the higher-order `twice` function. The argument `f` needs to be a function that satisfies the given constraints, and therefore the callsite `twice(inc, n)` in line 16 requires ESVERIFY to compare the pre- and postconditions of `inc` with pre- and postconditions in lines 7 and 8. 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.

3.2.5   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.

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

■ **Listing 3.3**: 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**.

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 3.4.

Despite these restrictions, it is possible to express complex recursive data structures. Section 3.4 presents examples of such data structures, including user-defined list classes containing sorted integers.

```
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); // does not hold (a[1] might be 4)
7    assert(a[2] > 1); // does not hold (a might have only 2 elements)
8  }
```

■ **Listing 3.4**: ᴇsᴠᴇʀɪғʏ includes basic support for immutable arrays. The elements of an array can be described with **every**.

3.2.6 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. As mentioned in Section 3.1, the latest edition of the ECMAScript standard [28] includes *Promises* [72] 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. ᴇsᴠᴇʀɪғʏ can accurately express these kinds of specifications in pre- and postconditions as shown in Listing 3.5, while standard type systems need to resort to code changes, such as sum types and injections.

57

```
1   class Promise {
2     constructor (value) { this.value = value; }
3   }
4   function resolve (fulfill) {
5     // "fulfill" is promise, a then-able or a value without a "then" property
6     requires(fulfill instanceof Promise ||
7              spec(fulfill.then, () => true, () => true) ||
8              !('then' in fulfill));
9     ensures(res => res instanceof Promise);
10
11    if (fulfill instanceof Promise) {
12      return fulfill;
13    } else if ('then' in fulfill) {
14      return new Promise(fulfill.then());
15    } else {
16      return new Promise(fulfill);
17    }
18  }
```

■ **Listing 3.5**: The standard `Promise.resolve()` function in JavaScript has complex polymorphic behavior. This simplified mock definition illustrates how ESVERIFY enables such dynamic programming idioms.

3.3 Implementation

The ESVERIFY prototype implementation[4] 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[5]. Alternative integrations such as extensions for Vim and Emacs also exist.

The basic verification process and overall design of ESVERIFY is depicted in

---

[4]Implementation Source Code: https://github.com/levjj/esverify/
[5]Online live demo of ESVERIFY: https://esverify.org/try
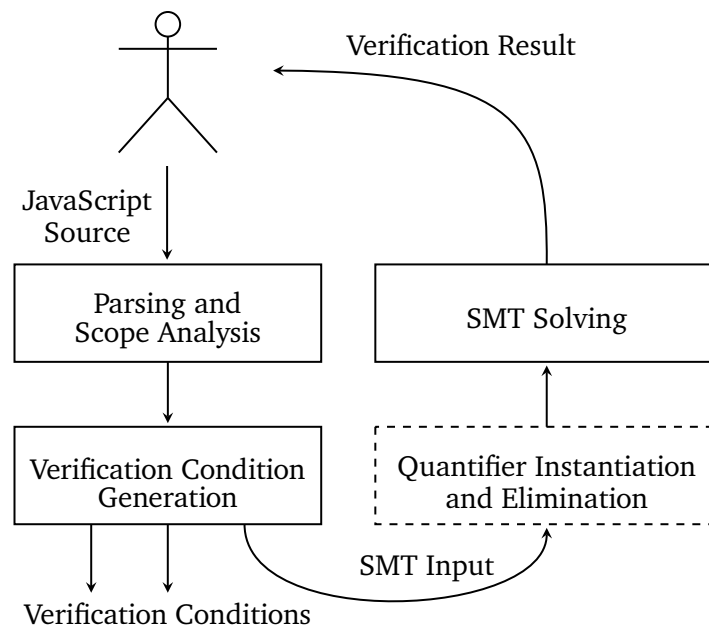
■ **Figure 3.1**: The basic verification workflow: ᴇꜱᴠᴇʀɪꜰʏ generates and statically checks verification conditions by SMT solving.

Figure 3.1.

The first step of the process involves **parsing** the source code and restricting the input language to a subset of JavaScript supported by ᴇꜱᴠᴇʀɪꜰʏ. Some of these restrictions may be lifted in future versions of ᴇꜱᴠᴇʀɪꜰʏ, 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, ᴇꜱᴠᴇʀɪꜰʏ 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

59

```
1  function max(a, b) {
2    requires(typeof(a) === 'number');
3    requires(typeof(b) === 'number');
4    ensures(result => result >= a);
5    if (a > b) {
6      return a;
7    } else {
8      return b;
9    }
10 }
```

$$\forall a \forall b.$$

$$\text{typeof}(a) = \text{`` } number\text{''}$$

$$\wedge \quad \text{typeof}(b) = \text{`` } number\text{''}$$

$$\wedge \quad a > b \implies result = a$$

$$\wedge \quad \neg(a > b) \implies result = b$$

$$\implies result \geq a$$

Figure 3.2: The code on the left is annotated with a postcondition in line 4. A simplified verification condition for this postcondition is shown on the right.

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 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 3.2 illustrates this process

$$(\forall a, b.\max(a, b) \geq a)$$

$$\Longrightarrow \max(3, 5) > 0$$

$$(\forall a, b.\max(a, b) \geq a)$$

$$\wedge \quad \max(3, 5) \geq 3$$

$$\Longrightarrow \max(3, 5) > 0$$

Figure 3.3: 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.

for a simple example. The verification condition shown on the right checks whether the preconditions and the translated function body imply the postcondition in line 4. Section 4.5 describes the verification rules in more detail.

The verification condition is then transformed with a **quantifier instantiation** procedure. As illustrated by Figure 3.3, 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[6]. 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 [77] or CVC4 [7, 8]. If the solver cannot find a solution for the negated verification condition, i.e. if the solver cannot refute the propo-

---

[6]The formal definition of the quantifier instantiation algorithm is given in Section 4.3.

sition, verification succeeded. Otherwise, the returned model includes an assignment of free variables that acts as a counterexample.

3.4   Evaluation

A comprehensive evaluation of ESVERIFY for real-world projects would require a stable and mature implementation and immense engineering effort. However, it is also possible to illustrate potential benefits and limitations with limited case studies such as verified implementations of well-known algorithms. Therefore, this section presents and discusses a series of non-trivial programs and verifies their functional correctness.

3.4.1   Reversing an Ascending List

This example involves a class definition for lists of integers and a `reverse` function. By adding annotations about pre- and postconditions, ESVERIFY can statically verify that reversing an ascending list yields a descending list.

First, the program defines the class `IntList` as a linked list with `head` and `tail`.

```
1  class IntList {
2    constructor (head, tail) {
3      this.head = head;
4      this.tail = tail;
5    }
6    invariant () {
7      return typeof(this.head) === 'number' &&
8             (this.tail === null || this.tail instanceof IntList);
9    }
10 }
```

The class invariant of `IntList` ensures that the current element (`head`) is a number and that the linked list (`tail`) is also an `IntList`. Instead of introducing a second class for empty lists, this program uses **null** to represent empty lists.

In order to verify the reverse function, is is necessary to define what it means for a list to be ascending. In ESVERIFY, these definitions, so-called *predicates*, are written as standard JavaScript functions despite being used solely for verification purposes.

```
11  function isAscending (list) {
12    requires(list === null || list instanceof IntList);
13    ensures(res => typeof(res) === 'boolean');
14    ensures(pure());
15
16    return list === null || list.tail === null ||
17           list.head <= list.tail.head && isAscending(list.tail);
18  }
```

The body of this function determines whether a given list contains ascending integers. The precondition in line 12 ensures that arguments are either integer lists or **null**, which represents the empty list, and the postconditions in lines 13 and 14 restrict the function to return boolean values and not invoke side effects. These annotations are primarily for checking the recursive call of `isAscending` in line 17. The remainder of the program will use the `isAscending` predicate function as a definition for ascending integer lists. In other verified languages, such as Dafny [62], `isAscending` would correspond to a "ghost function" but ESVERIFY does not currently differentiate between predicates and regular functions.

The definition of `isDescending` follows accordingly.

63

```
19  function isDescending (list) {
20    requires(list === null || list instanceof IntList);
21    ensures(res => typeof(res) === 'boolean');
22    ensures(pure());
23
24    return list === null || list.tail === null ||
25          list.head >= list.tail.head && isDescending(list.tail);
26  }
```

ESVERIFY does not currently support mutation of class instances. Therefore, the main algorithm for reversing the list is purely functional. Instead of modifying the list supplied as argument, it uses a recursive helper function reverseHelper to traverse the list and build up a reversed list. The implementation of the reverse function is shown below. Most importantly, it requires the argument to be an ascending integer list and it ensures that the result is a descending integer list.

```
27  function reverse (list) {
28    requires(list === null || list instanceof IntList);
29    requires(isAscending(list));
30    ensures(res => res === null || res instanceof IntList);
31    ensures(res => isDescending(res));
32    ensures(pure());
33
34    if (list === null) {
35      isDescending(null);     // trigger instantiation
36      return null;
37    } else {
38      isAscending(list);      // trigger instantiation
39      isAscending(list.tail); // ..
40      isDescending(null);     // ..
41      return reverseHelper(null, list.head, list.tail);
42    }
43  }
```

As shown in lines 34–36, an empty list as input argument results in an empty list as return value. While it seems obvious that an empty list is descending, ESVERIFY requires an explicit hint as shown in line 35 to unfold the definition of `isDescending` for the argument **null**. This function call does not contribute to the result, it is merely a *trigger* for the underlying SMT solver and the quantifier instantiation algorithm.

Similarly, the preconditions of `reverseHelper` require that `list.tail` is ascending which necessitates triggers that expand the definition of `isAscending` in lines 38–40.

Finally, the recursive helper function `reverseHelper` implements the actual reverse algorithm. A reversed list is accumulated in `revList` (initially just **null**) and the `pivot` is the current number, not smaller than the descending numbers in `revList` and not greater than the ascending numbers in the remaining list `list`.

Again, pre- and postconditions are used to specify these constraints and triggers are used to unfold the definitions of `isAscending` and `isDescending` for concrete arguments.

```
44  function reverseHelper (revList, pivot, list) {
45    requires(revList === null || revList instanceof IntList && revList.head <= pivot);
46    requires(isDescending(revList));
47    requires(typeof pivot === 'number');
48    requires(list === null || list instanceof IntList && pivot <= list.head);
49    requires(isAscending(list));
50    ensures(res => res === null || res instanceof IntList);
51    ensures(res => isDescending(res));
52    ensures(pure());
53
54    const newRevList = new IntList(pivot, revList);
55    isDescending(newRevList); // trigger instantiation
56    if (list === null) {
```

```
57      return newRevList;
58    } else {
59      isAscending(list);       // trigger instantiation
60      isAscending(list.tail); // ..
61      return reverseHelper(newRevList, list.head, list.tail);
62    }
```

The overall program can be verified with ESVERIFY and thereby the functional correctness of the `reverse` function statically checked. Out of 64 total lines in the source code, 40 are primarily for verification purposes. This includes the class invariant, annotated pre- and postconditions, the definitions of `isAscending` and `isDescending`, and function calls in the code that serve as triggers.

3.4.2   MergeSort Algorithm

As a slightly more complex example, this section includes an implementation of Merge-Sort and verifies that returned lists are sorted.

The algorithm is based on linked lists of integers, analogous to the class definition used for the reverse algorithm in Section 3.4.1.

```
1  class IntList {
2    constructor (head, tail) {
3      this.head = head;
4      this.tail = tail;
5    }
6    invariant () {
7      return typeof(this.head) === 'number' &&
8             (this.tail === null || this.tail instanceof IntList);
9    }
10 }
```

Additionally, the MergeSort algorithm defines a predicate `isSorted` as a function that is identical to the `isAscending` definition in Section 3.4.1.

```
11  function isSorted (list) {
12    requires(list === null || list instanceof IntList);
13    ensures(res => typeof(res) === 'boolean');
14    ensures(pure());
15
16    return list === null || list.tail === null ||
17           list.head <= list.tail.head && isSorted(list.tail);
18  }
```

The MergeSort algorithm divides the input list into two partitions that are sorted independently and then merged. In order to split and return an integer list with two partitions as a single value, the program introduces a second class definition that represents a pair of two integer lists.

```
19  class IntListPartition {
20    constructor (left, right) {
21      this.left = left;
22      this.right = right;
23    }
24    invariant () {
25      return (this.left === null || this.left instanceof IntList) &&
26             (this.right === null || this.right instanceof IntList);
27    }
28  }
```

The `partition` function takes the original list as first argument and then accumulates two lists recursively while alternating between appending to the first or second list.

67

```
29  function partition (lst, fst, snd, alternate) {
30    requires(lst === null || lst instanceof IntList);
31    requires(fst === null || fst instanceof IntList);
32    requires(snd === null || snd instanceof IntList);
33    requires(typeof(alternate) === 'boolean');
34    ensures(res => res instanceof IntListPartition);
35    ensures(pure());
36
37    if (lst === null) {
38      return new IntListPartition(fst, snd);
39    } else if (alternate) {
40      return partition(lst.tail, new IntList(lst.head, fst), snd, false);
41    } else {
42      return partition(lst.tail, fst, new IntList(lst.head, snd), true);
43    }
44  }
```

The merge routine is the most complex part of the algorithm. Given two lists, if neither of them is empty, the method will compare the head elements and then select the lesser element. All remaining elements will be recursively merged such that prepending the selected elements results in a sorted list. The postconditions explain this behavior and triggers are used to unfold the definition of isSorted.

```
45  function merge (left, right) {
46    requires(left === null || left instanceof IntList);
47    requires(isSorted(left));
48    requires(right === null || right instanceof IntList);
49    requires(isSorted(right));
50    ensures(res => res === null || res instanceof IntList);
51    ensures(res => isSorted(res));
52    ensures(res => (left === null && right === null) === (res === null));
53    ensures(res => !(left !== null && (right === null || right.head >= left.head))
54               || (res !== null && res.head === left.head));
55    ensures(res => !(right !== null && (left === null || right.head < left.head))
56               || (res !== null && res.head === right.head));
57    ensures(pure());

58
59    if (left === null) {
60      return right;
61    } else if (right === null) {
62      return left;
63    } else if (left.head <= right.head) {
64      isSorted(left);        // trigger instantiation
65      isSorted(left.tail);  // ..
66      const merged = merge(left.tail, right);
67      const res = new IntList(left.head, merged);
68      isSorted(res);         // trigger instantiation
69      return res;
70    } else {
71      isSorted(right);       // trigger instantiation
72      isSorted(right.tail); // ..
73      const merged = merge(left, right.tail);
74      const res = new IntList(right.head, merged);
75      isSorted(res);         // trigger instantiation
76      return res;
77    }
78  }
```

Finally, the main function simply returns the given list if it is empty or has just

a single element and otherwise uses `partition`, recursion and `merge` to implement the MergeSort algorithm.

```
79  function sort (list) {
80    requires(list === null || list instanceof IntList);
81    ensures(res => res === null || res instanceof IntList);
82    ensures(res => isSorted(res));
83    ensures(pure());
84
85    if (list === null || list.tail === null) {
86      isSorted(list);
87      assert(isSorted(list));
88      return list;
89    }
90    const part = partition(list, null, null, false);
91    return merge(sort(part.left), sort(part.right));
92  }
```

In summary, about 48 out of a total 96 lines are verification annotations, including invariants, pre- and postconditions and the predicate function `isSorted`.

3.4.3 Custom Generic List Class

While the previous two examples use a custom class specifically for lists of numbers, it is also possible to define a list class that can be parameterized by a generic invariant that holds for each element.

Here, the field `each` is a function value that acts as a predicate, i.e. it is a pure boolean-valued function that is used for verification returns **true** for all elements in the list.

```
1  class List {
2    constructor (head, tail, each) {
3      this.head = head; this.tail = tail; this.each = each;
4    }
5    invariant () {
6      return spec(this.each, (x) => true, (x, y) => pure() && typeof(y) === 'boolean')
7        && (true && this.each)(this.head) // same as 'this.each(this.head)'
8                                           // but avoids binding 'this'
9        && (this.tail === null || (this.tail instanceof List &&
10                                    this.each === this.tail.each));
11   }
12 }
```

Mapping a function `f` over a list results in a new list where each element corresponds to the result of `f`. In this example, the current list predicate `lst.each` has to satisfy the precondition of `f`. Additionally, there is a third argument `newEach` that represents the new predicate of the list after mapping. Therefore, the postcondition of `f` needs to be at least as strong as `newEach`. To simplify reasoning, the pseudo call `pure()` in the postcondition ensures the absence of side effects. It is important to note that function calls in an assertion context are uninterpreted, so the call `newEach(y)` in line 21 only refers to the function return value but does not actually invoke the function.

```
13 function map (f, lst, newEach) {
14   requires(spec(newEach, (x) => true, (x, y) => pure() && typeof(y) === 'boolean'));
15   requires(lst === null || spec(f, (x) => (true && lst.each)(x),
16                                    (x, y) => pure() && newEach(y)));
17   requires(lst === null || lst instanceof List);
18   ensures(res => res === null || (res instanceof List && res.each === newEach));
19   ensures(pure()); // necessary for recursive calls
20
21   return lst === null ? null
22                       : new List(f(lst.head), map(f, lst.tail, newEach), newEach);
23 }
```

It would be possible to rewrite the previous two examples using this parameterized list class. This demonstrates how generic data structures can be expressed with ESVERIFY analogous to parameterized type systems.

3.4.4 Theorems and Proofs written in JavaScript

Despite being a program verifier, ESVERIFY can also be used to write general theorems and proofs.

As previously shown in Listing 3.2, `while` loops with loop invariants are analogous to simple induction proofs over natural numbers.

More generally, `spec` can be used to reify propositions instead of describing computation results. In particular, the postcondition need not only describe the 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 value can then be supplied as argument to higher-order functions to build up longer proofs.

It is important to note that ESVERIFY does not currently check whether recursive functions and loops terminate. Therefore, ESVERIFY can only prove partial correctness and not total correctness. If an annotated function is evaluated to completion, its postcondition holds but theorems proved with ESVERIFY do not apply to non-terminating programs. Future work or external termination checkers could enable ESVERIFY to also prove total correctness.

For an example, the following Listing 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 [115].

```
1  function proof_f_mono (f, proof_f_inc, n, m) {
2    requires(spec(f,
3        (x) => Number.isInteger(x) && x >= 0,
4        (x, y) => Number.isInteger(y) && pure()));
5    requires(spec(proof_f_inc,
6        x => Number.isInteger(x) && x >= 0,
7        x => f(x) <= f(x + 1) && pure()));
8    requires(Number.isInteger(n) && n >= 0);
9    requires(Number.isInteger(m) && m >= 0);
10   requires(n < m);
11   ensures(f(n) <= f(m));
12   ensures(pure()); // no side effects
13
14   proof_f_inc(n); // instantiate proof for n
15   if (n + 1 < m) {
16     // invoke induction hypothesis (I.H.)
17     proof_f_mono(f, proof_f_inc, n + 1, m);
18   }
19 }
```

Here, f is an arbitrary function from non-negative integers to integers and proof_f_inc is a function argument that is not used for computation. Instead, a function value that satisfies the **spec** in lines 5–7 is a proof that f is monotonous for any non-negative x. Additionally, the arguments n and m are integers with n < m. The overall function proof_f_mono establishes f(n)<= f(m) with an induction proof (written as recursion) with proof_f_inc invoked at each step.

The function proof_f_mono can now be used to show f(n)<= f(m) for a concrete function f. The example below shows a definition of the Fibonacci function fib and a

73

proof that `fib` is increasing. It is now possible to show `fib(n)<= fib(m)` for arbitrary `n` and `m` with `n < m` by passing `fib` and `proof_fib_inc` into `proof_f_mono`.

```
20  function fib (n) {
21    requires(Number.isInteger(n) && n >= 0);
22    ensures(res => Number.isInteger(res));
23    ensures(pure());
24    if (n <= 1) {
25      return 1;
26    } else {
27      return fib(n - 1) + fib(n - 2);
28    }
29  }
30  function proof_fib_inc (n) {
31    requires(Number.isInteger(n) && n >= 0);
32    ensures(fib(n) <= fib(n + 1));
33    ensures(pure());
34    fib(n); // unfolds fib at n
35    fib(n + 1);
36    if (n > 0) {
37      fib(n - 1);
38      proof_fib_inc(n - 1); // I.H.
39    }
40    if (n > 1) {
41      fib(n - 2);
42      proof_fib_inc(n - 2); // I.H.
43    }
44  }
45  function proof_fib_mono (n, m) {
46    requires(Number.isInteger(n) && n >= 0);
47    requires(Number.isInteger(m) && m >= 0);
48    requires(n < m);
49    ensures(fib(n) <= fib(m));
50    ensures(pure());
51    proof_f_mono(fib, proof_fib_inc, n, m);
52  }
```

3.5 Future Work and Conclusions

This chapter described an approach for static verification of dynamically-typed JavaScript programs. The implementation, ESVERIFY, supports both dynamic programming idioms as well as higher-order functions. Internally, the verifier relies on a bounded quantifier instantiation algorithm and SMT solving, yielding concrete counterexamples for verification errors. While ESVERIFY enables verification of non-trivial programs such as Merge-Sort, 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 [100], and to extend it with reasoning about the heap, such as regions or dynamic frames [103]. Finally, while the approach presented in this chapter is purely static, future work might use runtime checks to enable sound execution of programs that are not fully verified. This idea is also used for hybrid and gradual type checking [102, 57, 2] and has recently been adapted to "soft verification" [82] and "gradual verification" [6].

Beware of bugs in the above code; I have only proved it correct, not tried it.

— Donald Knuth

■ Chapter 4

Formal Development of Program Verification with λ^S

In order to clarify the approach of ESVERIFY and reason about its properties, this chapter introduces a formal development of an idealized JavaScript-inspired, statically verified but dynamically typed language (λ^S).

The verification rules of λ^S are defined in terms of verification conditions whose validity is checked with a custom decision procedure. Therefore, this chapter first defines the class of logical propositions and axiomatizes their validity, then describes the decision procedure including quantifier instantiation, and finally gives the syntax and semantics of λ^S and shows that its verification rules are sound.

The definitions, axioms and theorems in this section are also formalized in the Lean theorem prover [78]. The source code of the formal development including definitions, axioms and theorems can be found in Appendix A and the full proof is available online at `https://github.com/levjj/esverify-theory/`.

Additionally, this chapter also includes a brief comparison with static refinement types [116]. Refined base type and dependent function types can be translated to

λ^S annotations and, while a full proof is beyond the scope of this thesis, the translated λ^S program is presumably verifiable if the original program is well-typed. This suggests that ESVERIFY is at least as expressive as a language with refinement types.

4.1 Overview

This chapter formally defines λ^S, a JavaScript-inspired dynamically typed language. Functions in λ^S are annotated with pre- and postconditions, rendered as logical propositions. These propositions can include unary and binary 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 perform the right instantiations, and therefore quantifiers imperil the decidability of the verification process [38, 90]. In order to ensure a decidable and predictable verification process, λ^S employs a bounded quantifier instantiation algorithm such that function calls in the source program act as hints ("triggers") that instantiate universal quantifiers. The algorithm only performs a bounded number of trigger-based instantiations and thereby avoids both brittle instantiation heuristics and infinite matching loops. Using this decision procedure for

$$\phi \in \text{Propositions} \quad ::= \quad \tau \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid pre_1(\otimes, \tau) \mid$$

$$pre_2(\oplus, \tau, \tau) \mid pre(\tau, \tau) \mid post(\tau, \tau) \mid \forall x.\phi$$

$$\tau \in \text{Terms} \quad ::= \quad v \mid x \mid \otimes \tau \mid \tau \oplus \tau \mid \tau(\tau)$$

$$\otimes \in \text{UnaryOperators} \quad ::= \quad \neg \mid isInt \mid isBool \mid isFunc$$

$$\oplus \in \text{BinaryOperators} \quad ::= \quad + \mid - \mid \times \mid / \mid \wedge \mid \vee \mid = \mid <$$

$$v \in \text{Values} \quad ::= \quad \text{true} \mid \text{false} \mid n \mid \langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle$$

$$\sigma \in \text{Environments} \quad ::= \quad \varnothing \mid \sigma[x \mapsto v]$$

$$n \in \mathbb{N} \qquad\qquad f, x, y, z \in \text{Variables}$$

■ Figure 4.1: Syntax of logical propositions used in the verifier.

verification conditions, it can be shown that verification of λ^S is sound, i.e. verifiable λ^S programs do not get stuck.

4.2   Logical Foundation

Figure 4.1 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 4.4.

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

⊢ ϕ. These axioms are mostly standard but are listed here explicitly because they form the foundation for both the mechanized Lean proof as well as the subsequent definitions and theorems in this chapter.

The term "true" is trivially valid.

Axiom 1. ⊢ true.

$$\boxed{\vdash \phi}$$

Validity of conjunctions and disjunctions follows standard inference rules.

Axiom 2. Iff both ⊢ ϕ_1 and ⊢ ϕ_2 then ⊢ $\phi_1 \land \phi_2$.

Axiom 3. If ⊢ ϕ_1 then ⊢ $\phi_1 \lor \phi_2$.

Axiom 4. If ⊢ ϕ_2 then ⊢ $\phi_1 \lor \phi_2$.

Axiom 5. If ⊢ $\phi_1 \lor \phi_2$ then ⊢ ϕ_1 or ⊢ ϕ_2.

The following axioms for negation assume that valid propositions do not include contradictions and that the law of the excluded middle holds.

Axiom 6. ⊢ $\phi \land \neg\phi$ is not true.

Axiom 7. ⊢ $\phi \lor \neg\phi$ is true.

For convenience, a notation for implication can be defined in terms of disjunction and negation.

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

There is no evaluation relation for terms. Instead, equality of terms is defined with a selection of axioms. For example, a standalone term is considered valid if it equals "true".

Axiom 8. Iff $\vdash \tau$ then $\vdash \tau = true$.

Equalities involving unary and binary operators are axiomatized in terms of a partial function δ, e.g. $\delta(+, 2, 3) = 5$.

Axiom 9. Iff $\delta(\otimes, v_x) = v$ then $\vdash v = \otimes v_x$.

Axiom 10. Iff $\delta(\oplus, v_x, v_y) = v$ then $\vdash v = v_x \oplus v_y$.

The propositions $pre_1(\otimes, v_x)$ and $pre_2(\oplus, v_x, v_y)$ can be used to reason about the domain of δ, i.e. the values for which the operators $\otimes$ and $\oplus$ are defined.

Axiom 11. If $\vdash pre_1(\otimes, v_x)$ then $(\otimes, v_x) \in dom(\delta)$.

Axiom 12. 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.

Definition 1 (Substitution). $\phi[x \mapsto v]$ denotes the proposition in which free occurrences of x in ϕ 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 τ.

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

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

Substitution in terms and propositions can also be expressed in terms of an environment σ mapping variables to values.

Definition 2 (Lookup). $\sigma(x)$ looks up the value associated with x in the environment σ.

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

An environment σ is a model for a proposition if the substituted proposition is valid. This definition of models is unconventional but it facilitates the subsequent formal development.

Notation 2 (Model). $\sigma \vDash \phi \quad \overset{\text{def}}{=\!=} \quad \vdash \sigma(\phi)$ $\boxed{\sigma \vDash \phi}$

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

Definition 4 (Satisfiability). The notation $Sat(\phi)$ indicates that the SMT solver found a model that satisfies ϕ. $\boxed{Sat(\phi)}$

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

81

Proof. SMT solving is not decidable for arbitrary propositions but the QF-UFLIA fragment of quantifier-free formulas with equality, uninterpreted function symbols and linear integer arithmetic is known to be decidable [79, 17]. Therefore, SMT solving terminates for all inputs and its result can be assumed to be consistent with the axioms above. $\square$

4.3 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 [38, 90] 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.

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

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 4.2.

To make the definition more concise, it is helpful to first define contexts $P^+[\circ]$ and $P^-[\circ]$ for fragments of 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

$$P^+[\circ] \qquad\qquad ::= \circ \mid \neg P^-[\circ] \mid P^+[\circ] \wedge P \mid P \wedge P^+[\circ] \mid P^+[\circ] \vee P \mid P \vee P^+[\circ]$$

$$P^-[\circ] \qquad\qquad ::= \neg P^+[\circ] \mid P^-[\circ] \wedge P \mid P \wedge P^-[\circ] \mid P^-[\circ] \vee P \mid P \vee P^-[\circ]$$

$$calls^-(P) \qquad\qquad \overset{\text{def}}{=} \{\, call(\tau) \mid \exists P^-.\ P = P^-[call(\tau)] \,\}$$

$$lift^+(P) \qquad\qquad \overset{\text{def}}{=} \text{match } P \text{ with}$$

$$\qquad\qquad P^+[\forall x.\{call(x)\} \Rightarrow P'] \to lift^+\big(P^+[call(y) \Rightarrow P'[x \mapsto y]]\big) \ (y \text{ fresh})$$

$$\qquad\qquad P^-[\exists x.P'] \qquad\qquad \to lift^+\big(P^-[P'[x \mapsto y]]\big) \qquad\qquad (y \text{ fresh})$$

$$\qquad\qquad \text{otherwise} \qquad\qquad \to P$$

$$instantiateOnce^-(P) \overset{\text{def}}{=} P\left[\ P^-\Big[(\forall x.\{call(x)\} \Rightarrow P')\Big] \mapsto P^-\left[\begin{array}{l}(\forall x.\{call(x)\} \Rightarrow P') \wedge \\[2mm] \bigwedge\limits_{call(\tau) \in calls^-(P)} P'[x \mapsto \tau]\end{array}\right]\ \right]$$

$$erase^-(P) \qquad\qquad \overset{\text{def}}{=} P\left[\begin{array}{l} P^-\Big[(\forall x.\{call(x)\} \Rightarrow P'')\Big] \mapsto P^-\Big[\text{true}\Big] \\[2mm] \qquad\quad P^+\Big[call(\tau)\Big] \qquad \mapsto P^+\Big[\text{true}\Big] \\[2mm] \qquad\quad P^-\Big[call(\tau)\Big] \qquad \mapsto P^-\Big[\text{true}\Big] \end{array}\right]$$

$$instantiate^-(P,n) \quad \overset{\text{def}}{=} \text{if } n = 0$$

$$\qquad\qquad \text{then } erase^-(lift^+(P))$$

$$\qquad\qquad \text{else } instantiate^-(instantiateOnce^-\big(lift^+(P)\big), n-1)$$

$$\langle P \rangle \qquad\qquad \overset{\text{def}}{=} \text{let } n = \text{maximum level of quantifier nesting of } P \qquad\qquad \boxed{\langle P \rangle}$$

$$\qquad\qquad \text{in } \neg Sat\,(\neg instantiate^-(P,n))$$

■ **Figure 4.2**: The decision procedure lifts, instantiates and finally eliminates quantifiers. The number of iterations is bounded by the maximum level of quantifier nesting.

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, 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 both non-recursive and trivially terminate. Since the maximum level of nesting is finite, $\langle P \rangle$ performs only a finite number of instantiation rounds. With existential quantifiers only in negative positions, the erased and lifted result is quantifier-free, so according to Theorem 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 5 (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 $\neg \forall x.\ \neg prop(P)$.*

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

Proof. With Axiom 15, it can be shown that $lift^+$ preserves equisatisfiability. Additionally, conjuncts inserted by *instantiateOnce⁻* could also be obtained via classical (not trigger-based) instantiation using Axiom 14. Furthermore, since *erase⁻* only removes quantifiers in negative positions and triggers that are inconsequential for validy, 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, Theorem 1 can be used to show that a valid VC according to the decision procedure is also valid without trigger-based instantiation[1]. □

4.4 Syntax and Semantics of λ^S

Figure 4.3 defines the syntax of λ^S. Values v are either integer or boolean constants, or closures consisting of a function definition and an environment σ (as previously defined in Figure 4.1). Expressions e are assumed to be in A-normal form [33] such that each step of the computation looks up values in the environment and augments the environment with the result. Function calls are processed with a call stack configuration κ that resumes once the callee finished its computation with a value. This formalism avoids substitution in expressions and assertions in order to simplify subsequent proofs.

Function definitions in λ^S are written as let $f(x)$ req R ens $S = e_1$ in e_2 with an annotated precondition R and postcondition S. These annotations can include terms such as constants, variables and uninterpreted function application $\tau(\tau)$, as well as logical connectives and function specifications "spec $\tau(x)$ req R ens S".

---

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

$$\otimes \in \text{UnaryOperators} ::= \neg \mid isInt \mid isBool \mid isFunc$$

$$\oplus \in \text{BinaryOperators} ::= + \mid - \mid \times \mid / \mid \wedge \mid \vee \mid = \mid <$$

$$v \in \text{Values} ::= \text{true} \mid \text{false} \mid n \mid \langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle$$

$$e \in \text{Expressions} ::= \text{let } x = \text{true in } e \mid \text{let } x = \text{false in } e \mid \text{let } x = n \text{ in } e \mid$$
$$\text{let } f(x) \text{ req } R \text{ ens } S = e \text{ in } e \mid \text{let } y = \otimes x \text{ in } e \mid$$
$$\text{let } z = x \oplus y \text{ in } e \mid \text{let } y = f(x) \text{ in } e \mid \text{if } (x) \, e \text{ else } e \mid \text{return } x$$

$$R, S \in \text{Specs} ::= \tau \mid \neg R \mid R \wedge R \mid R \vee R \mid \text{spec } \tau(x) \text{ req } R \text{ ens } S$$

$$\kappa \in \text{Stacks} ::= (\sigma, e) \mid \kappa \cdot (\sigma, \text{let } y = f(x) \text{ in } e)$$

■ **Figure 4.3**: Syntax of λ^S programs. Function definitions have pre- and postconditions written as simple logical propositions with the *spec* syntax for higher-order functions. The syntax of operators and values follows the definition in Figure 4.1.

As an example, the following JavaScript function uses ESVERIFY annotations to specify its pre- and postcondition:

```
1  function inc (x) {
2    requires(Number.isInteger(x));
3    ensures(result => Number.isInteger(y) && result > x);
4    return x + 1;
5  }
6  ...
```

The same annotated function could be expressed in λ^S as follows:

let $inc(x)$ req $(isInt(x))$ ens $(isInt(inc(x)) \wedge inc(x) > x) =$

    let $y = x + 1$

    in return y

in ...

Since λ^S is a pure functional language, the postcondition can refer to the function result with an uninterpreted invocation such as $inc(x)$ instead of introducing a variable binding such as `result` above.

The operational semantics of λ^S is specified by a small-step evaluation relation over stack configurations κ, as shown in Figure 4.4. 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 6 (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 $x \in \sigma$.

4.5   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 4.6. 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 as shown in Figure 4.5. Since λ^S is purely functional, P still holds after evaluating e, so Q corresponds to the *marginal postcondition* and $P \wedge Q[\bullet]$ corresponds to the *strongest postcondition*.

For simplicity, the formalism follows the convention that the free variables of P, denoted $FV(P)$, must be exactly the set of variables in scope at e.

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

$$(\sigma, \text{let } x = v \text{ in } e) \hookrightarrow (\sigma[x \mapsto v], e) \qquad\qquad\qquad \text{[E-VAL]}$$

$$\text{where } v \in \{\text{true}, \text{false}, n\}$$

$$(\sigma, \text{let } f(x) \text{ req } R \text{ ens } S = e_1 \text{ in } e_2) \hookrightarrow (\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S \{e_1\}, \sigma\rangle], e_2) \text{ [E-CLOSURE]}$$

$$(\sigma, \text{let } y = \otimes x \text{ in } e) \hookrightarrow (\sigma[y \mapsto v], e) \qquad\qquad\qquad \text{[E-UNOP]}$$

$$\text{where } v = \delta(\otimes, \sigma(x))$$

$$(\sigma, \text{let } z = x \oplus y \text{ in } e) \hookrightarrow (\sigma[z \mapsto v], e) \qquad\qquad\qquad \text{[E-BINOP]}$$

$$\text{where } v = \delta(\oplus, \sigma(x), \sigma(y))$$

$$(\sigma, \text{let } z = f(y) \text{ in } e) \hookrightarrow (\sigma_f[g \mapsto \sigma(f), x \mapsto \sigma(y)], e_f) \cdot (\sigma, \text{let } z = f(y) \text{ in } e)$$

$$\text{where } \sigma(f) = \langle g(x) \text{ req } R \text{ ens } S \{e_f\}, \sigma_f\rangle \text{ [E-CALL]}$$

$$(\sigma, \text{return } z) \cdot (\sigma_2, \text{let } y = f(x) \text{ in } e_2) \hookrightarrow (\sigma_2[y \mapsto \sigma(z)], e_2) \qquad\qquad\qquad \text{[E-RETURN]}$$

$$(\sigma, \text{if } (x) \; e_1 \text{ else } e_2) \hookrightarrow (\sigma, e_1) \qquad\qquad\qquad \text{[E-IF-TRUE]}$$

$$\text{if } \sigma(x) = \text{true}$$

$$(\sigma, \text{if } (x) \; e_1 \text{ else } e_2) \hookrightarrow (\sigma, e_2) \qquad\qquad\qquad \text{[E-IF-FALSE]}$$

$$\text{if } \sigma(x) = \text{false}$$

$$\kappa \cdot (\sigma, \text{let } y = f(x) \text{ in } e) \hookrightarrow \kappa' \cdot (\sigma, \text{let } y = f(x) \text{ in } e) \qquad\qquad\qquad \text{[E-CONTEXT]}$$

$$\text{if } \kappa \hookrightarrow \kappa'$$

■ **Figure 4.4:** Operational semantics of λ^S

$$Q[\bullet] \ \in \ \text{PropositionContexts} \ ::= \ P \mid \eta[\bullet] \mid \neg Q[\bullet] \mid Q[\bullet] \wedge Q[\bullet] \mid Q[\bullet] \vee Q[\bullet] \mid$$

$$pre_1(\otimes, \eta[\bullet]) \mid pre_2(\oplus, \eta[\bullet], \eta[\bullet]) \mid$$

$$pre(\eta[\bullet], \eta[\bullet]) \mid post(\eta[\bullet], \eta[\bullet]) \mid call(\eta[\bullet]) \mid$$

$$\forall x. \{call(x)\} \Rightarrow Q[\bullet] \mid \exists x. \ Q[\bullet]$$

$$\eta[\bullet] \ \in \ \text{TermContexts} \qquad ::= \ \bullet \mid \tau \mid \otimes \eta[\bullet] \mid \eta[\bullet] \oplus \eta[\bullet] \mid \eta[\bullet](\eta[\bullet])$$

■ **Figure 4.5**: Proposition and term contexts contain a hole • for the evaluation result.

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.

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 postcondi-

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

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

$$\frac{\begin{array}{c} x \in FV(P) \quad y \in FV(P) \quad z \notin FV(P) \\[4pt] \langle P \Rightarrow pre(\oplus, x, y) \rangle \quad P \wedge z = x \oplus y \vdash e : Q \end{array}}{P \vdash \text{let } z = x \oplus y \text{ in } e : \exists z.\ z = x \oplus y \wedge Q} \text{ VC-BINOP}$$

$$\frac{\begin{array}{c} f \notin FV(P) \quad x \notin FV(P) \quad f \neq x \quad x \in FV(R) \quad FV(R) \subseteq FV(P) \cup \{f, x\} \\[4pt] FV(S) \subseteq FV(P) \cup \{f, x\} \quad P \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \vdash e_1 : Q_1 \\[4pt] \langle P \wedge \text{spec } f(x) \text{ req } R \text{ ens } S \wedge R \wedge Q_1[f(x)] \Rightarrow S \rangle \\[4pt] P \wedge \text{spec } f(x) \text{ req } R \text{ ens } (Q_1[f(x)] \wedge S) \vdash e_2 : Q_2 \end{array}}{P \vdash \text{let } f(x) \text{ req } R \text{ ens } S = e_1 \text{ in } e_2 : \exists f.\ \text{spec } f(x) \text{ req } R \text{ ens } (Q_1[f(x)] \wedge S) \wedge Q_2} \text{ VC-FN}$$

$$\frac{\begin{array}{c} f \in FV(P) \quad x \in FV(P) \quad \langle P \wedge call(x) \Rightarrow isFunc(f) \wedge pre(f, x) \rangle \\[4pt] y \notin FV(P) \quad P \wedge call(x) \wedge post(x) \wedge y = f(x) \vdash e : Q \end{array}}{P \vdash \text{let } y = f(x) \text{ in } e : \exists y.\ call(x) \wedge post(f, x) \wedge y = f(x) \wedge Q} \text{ VC-APP}$$

$$\frac{\begin{array}{c} x \in FV(P) \quad \langle P \Rightarrow isBool(f) \rangle \\[4pt] P \wedge x \vdash e_1 : Q_1 \quad P \wedge \neg x \vdash e_2 : Q_2 \end{array}}{P \vdash \text{if } (x)\ e_1 \text{ else } e_2 : (x \Rightarrow Q_1) \wedge (\neg x \Rightarrow Q_2)} \text{ VC-ITE}$$

$$\frac{x \in FV(P)}{P \vdash \text{return } x : x = \bullet} \text{ VC-RETURN}$$

■ **Figure 4.6**: The judgement $P \vdash e : Q$ verifies the expression e while assuming P, yielding a marginal postcondition Q with a hole $\bullet$ for the evaluation result.

tion 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 3 (Function Specifications)**.**

$$\text{spec } \tau(x) \text{ req } R \text{ ens } S \overset{\text{def}}{=} isFunc(\tau) \wedge \forall x.\{call(x)\} \Rightarrow ((R \Rightarrow pre(\tau, x)) \wedge (post(\tau, x) \Rightarrow S))$$

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.

4.6 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 Figure 4.6 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. $\square$

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

As mentioned in Section 4.2, 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 16. 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 17. Iff $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle, x \mapsto v_x] \vDash R$

then $\vdash pre(\langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle, v_x)$.

Axiom 18. 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 $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle, x \mapsto v_x] \vDash Q_2[f(x)] \wedge S$

       then $\vdash \text{post}(\langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle, v_x)$.

Axiom 19. 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 $\vdash \text{post}(\langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle, v_x)$

       then $\sigma[f \mapsto \langle f(x) \text{ req } R \text{ ens } S \{e\}, \sigma \rangle, x \mapsto v_x] \vDash Q_2[f(x)] \wedge S$

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 $(\varnothing, e) \hookrightarrow^* \kappa$ then $terminated(\kappa)$ or $\kappa \hookrightarrow \kappa'$ for some κ'.

Proof. Due to the complex quantifier instantiation of the decision procedure, verification safety is first proven for a verification judgement that is similar to the one in Figure 4.6 but uses the undecidable validity $\vdash prop(P)$ without quantifier instantiation rather than the decision procedure $\langle P \rangle$. 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 such that a given runtime stack is considered verifiable if, at each stack frame, the expression is verifiable with the translation of σ as precondition. With Theorem 3, it follows that soundness of this verification judgement also implies soundness with trigger-based quantifier instantiation. A complete proof is available at: `https://github.com/levjj/esverify-theory/`. $\qquad\square$

4.7 Extensions

The 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.

4.7.1 Imperative Programs

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.

To simplify reasoning, it is useful to distinguish heap-manipulating functions from pure functions. In ESVERIFY, the pseudo call `pure()` appearing in a postcondition indicates that the function is prohibited from manipulating the heap. A formal development might introduce separate syntactical classes for these types of functions.

In order to express invariants for heap manipulating code such as functions and loops, ESVERIFY additionally introduces a syntax `old(x)` that denotes the previous value of a variable. This enables annotations such as `x === old(x)` to confine the effects of heap-manipulating code to certain mutable variables. This issue can also be addressed with segmentation logic, effect systems, regions and dynamic frames [103].

$$c \in \text{ClassNames} \qquad fd \in \text{FieldNames} \qquad f, x, y, z, this \in \text{Variables}$$

$$v \in \text{Values} \quad ::= \quad \dots \mid C(\overline{v})$$

$$e \in \text{Expressions} \quad ::= \quad \dots \mid \text{let } y = \text{new } C(\overline{x}) \text{ in } e \mid \text{let } y = x.fd \text{ in } e$$

$$\tau \in \text{Terms} \quad ::= \quad \dots \mid \tau.fd \mid C(\overline{\tau})$$

$$P \in \text{VCs} \quad ::= \quad \dots \mid fd \text{ in } \tau \mid \tau \text{ instanceof } C \mid access(\tau) \mid \forall x.\{access(x)\} \Rightarrow P$$

$$D \in \text{ClassDefs} \quad ::= \quad \text{class } C(\overline{fd}) \text{ inv } S$$

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

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

■ **Figure 4.7**: Extending the verification rules of λ^S with simple immutable classes with class invariants.

4.7.2 Recursive Data types and Classes

The core language λ^S can be extended to support "classes" as shown in Figure 4.7. 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 3.4.3.

The class invariant has to be instantiated for concrete instances of the class, so

96

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, VCs need to be augmented with a preamble such that for each class $C(\overline{fd})$ inv S the following quantifier is assumed:

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

Instantiating this quantifier with an *access(x)* trigger yields a class invariant S with *this* replaced by the accessed object, as well as a description of the fields $\overline{fd}$.

4.8 Comparison with Refinement Types

The verification rules shown in Figure 4.6 resemble static typing rules. This section provides a brief comparison of the program verification approach with static type checking by defining an automatic translation of types to λ^S annotations and examining concrete examples. The results suggest that ESVERIFY is at least as expressive as systems such as LiquidHaskell [116] but a comprehensive formalization of dependent type systems and a formal proof to determine their expressiveness is beyond the scope of this thesis.

The language λ^T is similar to λ^S but with type annotations instead of pre- and postconditions. Figure 4.8 sketches 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.

The typing rule for function definitions (T-FN) is shown in Figure 4.8. This annotated return type T might refer to the function argument in order to support dependent

$$R, S \in \text{Specs} \qquad ::= \quad \tau \mid \neg R \mid R \wedge R \mid R \vee R$$

$$t \in \text{TypedExpressions} \quad ::= \quad \ldots \mid \text{let } f(x : T) : T = t \text{ in } t$$

$$T \in \text{Types} \qquad ::= \quad \{ x : B \mid R \} \mid x{:}T \to T$$

$$B \in \text{BaseTypes} \qquad ::= \quad \text{Bool} \mid \text{Int}$$

$$\Gamma \in \text{TypeEnvironments} \quad ::= \quad \varnothing \mid \Gamma, x : T$$

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

$$\frac{B_1 = B_2 \quad \langle \llbracket \Gamma \rrbracket \wedge R \Rightarrow S \rangle \quad x \notin dom(\Gamma)}{\Gamma \vdash \{ x : B_1 \mid R \} <: \{ x : B_2 \mid S \}} \;\; \text{ST-REF} \qquad \boxed{\Gamma \vdash T <: T}$$

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

$$\llbracket \varnothing \rrbracket \quad \overset{\text{def}}{=\!=} \quad \text{true} \qquad\qquad\qquad\qquad \boxed{\llbracket \Gamma \rrbracket}$$

$$\llbracket \Gamma, x : T \rrbracket \quad \overset{\text{def}}{=\!=} \quad \llbracket \Gamma \rrbracket \wedge \llbracket x : T \rrbracket$$

$$\llbracket \tau : \{x : \text{Bool} \mid R\} \rrbracket \quad \overset{\text{def}}{=\!=} \quad isBool(\tau) \wedge R[x \mapsto \tau] \qquad \boxed{\llbracket \tau : T \rrbracket}$$

$$\llbracket \tau : \{x : \text{Int} \mid R\} \rrbracket \quad \overset{\text{def}}{=\!=} \quad isInt(\tau) \wedge R[x \mapsto \tau]$$

$$\llbracket \tau : (x : T_x \to T) \rrbracket \quad \overset{\text{def}}{=\!=} \quad \text{spec } \tau(x) \text{ req } \llbracket x : T_x \rrbracket \text{ ens } \llbracket \tau(x) : T \rrbracket$$

■ **Figure 4.8**: Selected typing and subtyping rules of a statically typed language λ^T. Functions are annotated with refined base types or dependent function types where refinements R are analogous to specifications in λ^S.

types but other free variables in refinements can break hygiene, so T-FN restricts free variables in user-provided types T_x and T accordingly and also prevents name clashes with previously defined symbols f and x. For checking the function body t_1, the type environment Γ is augmented with types for x and the function f itself to enable recursion. Instead of checking the postcondition with a verification conditions as in λ^S, the computed type T_1 for the function body is compared with the annotated return type T by checking subtyping. Type checking then proceeds with the expression t_2.

The subtyping relation is also shown in Figure 4.8. 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 higher-order function in λ^T is well-typed as the return function type is a subtype of the argument function type:

$$\text{let } f \left(g : \quad \left(x : \{x : \text{Int} \mid x > 3\} \rightarrow \{y : \text{Int} \mid y > 8\}\right)\right) :$$

$$\left(x : \{x : \text{Int} \mid x > 4\} \rightarrow \{y : \text{Int} \mid y > 7\}\right) \ = \ g \quad \text{in } ...$$

This expression translates to an ESVERIFY program with **spec** in pre- and postcondition:

```
1  function f (g) {
2    requires(    spec(g, x => x > 3, (x,y) => y > 8));
3    ensures(g => spec(g, x => x > 4, (x,y) => y > 7));
4    return g;
5  }
```

This program is verifiable with the quantifier instantiation algorithm described in Section 4.3. 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 thesis 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.

Program testing can be a very effective way to show the presence of bugs,
but is hopelessly inadequate for showing their absence.

— Edsger W. Dijkstra

■ Chapter 5

Automatic Test Generation with Counterexamples

The ESVERIFY program verifier and its formalization λ^S were presented and discussed in Chapters 3 and 4 respectively. However, in order to use verification as part of an iterative development process, the verifier also needs to provide useful and comprehensible feedback to the programmer.

Simple error messages are often not sufficient to explain verification errors to the programmer but counterexamples in the form of executable tests can serve as an explicit witness and enable step-by-step debugging of the relevant parts of the code with concrete variable values. This chapter describes how the ESVERIFY program verifier can be extended with a test generation algorithm and it discusses how these generated tests assist the programmer in determining whether a verification issue corresponds to a bug in the code or an insufficient annotation or invariant.

The implementation of the test generator is already merged into ESVERIFY itself and serves as a basis for an integrated development and verification environment that will be discussed in Chapter 6.
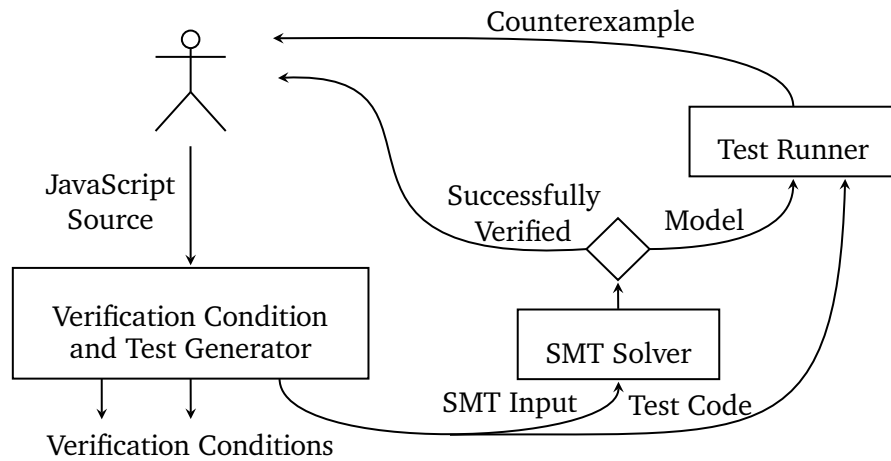
■ **Figure 5.1**: The basic verification workflow: ESVERIFY generates verification conditions to be checked by SMT solving. In order to explain verification issues to the programmer, ESVERIFY also generate tests for failed verification conditions that serve as counterexamples.

5.1 Overview

Figure 5.1 illustrates the basic verification process. The generation of verification conditions and the SMT solving procedure are described in more detail in Section 3.3 but this chapter extends this process with automatic counterexample generation and testing.

In summary, the **verification** step 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 set of free variables with unknown values, and

- a synthesized test with *holes*.

Each returned verification conditions consists of such a context and an assertion to check, such as a function postcondition.

The next step of the verification process involves checking the verification condition with an **SMT solver**. 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.

Such a model can then be combined with the synthesized partial test. Inserting concrete values into the holes of the test yields an executable counterexample that can be evaluated by a **test runner**. The test result provides useful feedback such as a dynamic assertion violation and enables inspection with interactive step-by-step debuggers and other tools.

5.2   Verification Errors and Assertion Violations

The purpose of the automatic test generation is to provide better feedback about failed verification conditions. To that end, the generated test should reflect both the specifics of the verification process as well as the actual behavior of the source code. In the case of loops and recursion, these two goals come into conflict because the static verifier overapproximates program behavior and thereby detects potential assertion violations that are not encountered by the actual evaluation of the program.

As an example, Listing 5.1 shows a program with an assertion in line 11 that

```
1  let safe = true;
2  let i = 0;
3  while (i < 3) {
4      invariant(Number.isInteger(i) && i <= 3);
5      if (i === 42) {
6          safe = false;
7      }
8      i++;
9  }
10 assert(i === 3); // verifiable
11 assert(safe);    // cannot be verified
```

Listing 5.1: Verifier detects assertion violation due to missing loop invariant.

cannot be verified. For any statements below the `while` loop, the loop invariants can be assumed to hold and the loop condition will be false but, apart from these assumptions, all mutable variables in the code could have changed in an arbitrary way. Therefore, the assertion in line 11 cannot be verified despite `safe` remaining unchanged by the loop when executing the program. Adding `invariant(safe);` to the loop would let its verification succeed.

There are two possible options for generating a counterexample test in these situations. One option is to reuse large fragments of the original program for the test. If the test leads to an error or assertion violation, it can serve as a witness of an 'actual' error that would also occur during normal execution and that can be fixed using the standard debugging process. However, for the example shown in Listing 5.1, running such a test would not result in an assertion violation because the original program execution satisfies the assertion in line 11. This indicates that the static analysis did not accurately

```
1  let safe = true;
2  let i = 0;
3  // while loop omitted; variables assigned according to SMT model:
4  safe = 0;
5  i = 3;
6  // statements after the while loop:
7  assert(i === 3);
8  assert(safe);
```

■ **Listing 5.2**: Replacing **while** loop in Listing 5.1 with counterexample values for test generation.

model the actual program behavior. Instead, the verification error is caused by an insufficiently strong loop invariant or precondition/postcondition. Unfortunately, there is no feedback about which annotation might be missing or what the internal verification state is regarding mutable variables after the loop.

As a second option, the generated test for the failed assertion in Listing 5.1 can omit the original **while** loop and instead insert assignments to mutable variables according to the values in the SMT model. Listing 5.2 shows an example of such a test. Clearly, the generated test reliably causes an assertion violation. Also, by using values from the SMT model for mutable variables after the loop, the generated test might help the programmer better understand the verification process and its shortcomings, and how the loop invariants can be improved to satisfy the assertions below the loop. In this case, the loop invariants constrain the mutable variable i but the possibles values of safe are left unrestricted, so the SMT model may assign it **false** or even 0 after the loop. This is theoretically consistent with the specified invariants but different from the actual

program behavior of the loop.

As outlined in this section, both the test case with the original source code as well as the model-based generated test case yield useful feedback to the programmer, serving the two competing goals of inspecting both the actual program behavior and static verification process.

To provide the benefits of both approaches, the ESVERIFY test generator retains the original loops but also enable programmers to query the variables in the verifier state with an integrated debugger.

5.3   Dynamic Checking of Assertions

The generated test for a verification condition consists of a transformed fragment of the relevant source code and a dynamically-checked assertion.

Assertions such as pre-, postconditions and invariants are specified as JavaScript boolean expressions. Therefore, dynamically checking these assertions as part of a test can be performed by evaluating these boolean expressions and throwing an exception if the evaluation result is different from `true`. If the source programs use exceptions handling, a potential assertion violation should be reported even if the exception is caught. However, ESVERIFY does not currently support exception handling and rejects programs with `try`/`catch` blocks.

5.3.1 Higher-order Functions

In order to support verification of higher-order functions, ESVERIFY introduces a special `spec` syntax to describe the minimum pre- and postcondition of a function value.

As an example, Figure 5.3 shows a function `twice` that expects a function argument `f`. Here, `f` should accept any integer value `x` as argument and any result `y` returned by `f` needs to be an integer greater than the argument `x`. The return value of `twice` is itself a function that applies `f` twice. However, due to an error in line 10, the postcondition of `twice` cannot be verified and, additionally, a bug in line 14 violates the precondition of `f`. Section 5.3 describes how generated tests serve as counterexamples for these two errors.

In contrast to simple boolean expressions, function specifications expressed with the `spec` syntax, as used by the `twice` function in Listing 5.3, cannot be checked dynamically for all values at the point of the assertion. Instead, the function argument is wrapped in a *contract* that enforces the specified pre- and postcondition for each subsequent call in the scope of this `spec`.

5.3.2 Contract Checking

Listing 5.4 illustrates how the `twice` function can be transformed to enable dynamic checking of function specifications. The code shown here is slightly simplified. In particular, it does not collapse wrappers to avoid repeated and redundant wrapping. It is important to note that assuming and asserting a function specification result in a

107

```
1   function inc (x) {
2     requires(Number.isInteger(x));
3     ensures(y => Number.isInteger(y) && y > x);
4     return x + 1;
5   }
6   function twice (f) {
7     requires(spec(f, (x) => Number.isInteger(x),
8                      (x, y) => Number.isInteger(y) && y > x));
9     ensures(g => spec(g, (x) => Number.isInteger(x),
10                        (x, y) => Number.isInteger(y) && y > 0)); // should be y > x
11    return function (x) {
12      requires(Number.isInteger(x));
13      ensures(y => Number.isInteger(y) && y > x);
14      return f(f(null)); // should be f(f(x))
15    };
16  }
17  const incTwice = twice(inc);
18  const y = incTwice(3);
19  assert(y > 3);
```

■ **Listing 5.3**: ESVERIFY example with a higher-order `twice` function. The pre- and postcondition of its function argument `f` and of the returned function `g` are both described with the **spec** syntax. Bugs in lines 10 and 14 cause verification errors.

different transformation. In this example, the `twice` function and its returned inner function are assumed to adhere to their function specifications but the function argument `f` is not trusted to behave correctly when invoked with correct arguments. Therefore, the postcondition of `f` is dynamically checked but its precondition assumed, and conversely, the preconditions of `g` and the **spec** in the postcondition of `twice` are enforced but their postconditions are not. Incidentally, this mechanism is similar to blame assignment [2, 53].

```
1  function twice (f) {
2    f = function (x) { // spec(f, ...) in line 7
3      const y = f(x);
4      assert(Number.isInteger(y) && y > x);
5      return y;
6    };
7    const g = function (x) { // inner function in lines 11-15
8      assert(Number.isInteger(x)); // need to check precondition
9      const y = f(f(null));       // but can assume postcondition
10     return y;
11   };
12   g = function (x) { // spec(g, ...) in line 9
13     assert(Number.isInteger(x));
14     const y = g(x);
15     return y;
16   };
17   return g; // return the wrapped inner function
18 }
```

■ **Listing 5.4**: Transformed code for the `twice` function in Listing 5.3. The assignments in lines 2 and 12 install wrappers according to the **spec** in lines 7 and 9 of Listing 5.3.

The example in Listing 5.3 only includes first and second-order functions but it is also possible for a function specification with **spec** to occur within another **spec**. In that case, the transformation of function specifications is applied recursively, i.e. the inner wrapping code gets executed as part of the dynamic checks of the outer wrapper.

This technique of dynamically checking assertions is only used in generated counterexample test but it could also be adapted to support sound execution of partially

verified programs, similar to "soft verification" [82] and "gradual verification" [6].

5.4   Synthesis of Counterexample Values

As shown in Figure 5.1, if the SMT solver refutes a verification condition, it returns a model that assigns values to free variables in the verification condition. In order to generate executable tests, these values have to translated from an SMT internal format to valid JavaScript expressions that can be inserted into the testing code.

For opaque JavaScript values such as `undefined`, **null**, **true** and **false** this translation is trivial. Numeric values in JavaScript conflate integers and floating point numbers and therefore are represented as either integers or real numbers in the SMT format in order to support both integer semantics for array indexing as well as floating point semantics for arithmetic operators[1]. Modern SMT solvers such as z3 [77] and CVC4 [7, 8] contain theories for strings with support for indexing and substrings[2]. Therefore, these JavaScript strings can be directly represented as SMT strings.

ESVERIFY provides limited support for object-oriented programming by means of immutable "classes" without inheritance and only trivial constructors. As an example, Listing 5.5a shows a class definition with a method `m` containing an incorrect assertion. Adding a sufficiently strong class invariant to `A` would let verification succeed. In the generated test shown in Listing 5.5b, **this** is renamed and initialized with a constructor invocation according to the SMT model.

---

[1]The SMTLIB standard also includes floating point values which, in contrast to real numbers, have limited precision. Unfortunately, current SMT solver implement these as bitvectors which negatively affects solving performance.

[2]Theories for strings have been added relatively recently and are still prone to errors such as SMT solver timeouts when converting strings with `str.to.int`.

```
1  class A {
2    constructor (x) {
3      this.x = x;
4    }
5    m () {
6      assert(this.x > 0);
7    }
8  }
```

```
1  const this_0 = new A(false);
2  assert(this_0.x > 0);
```

(a) A simple class definition. The assertion in line 6 does not hold for all instances of A.

(b) Generated test for the failed assertion in line 6 of Listing 5.5a.

■ **Listing 5.5**: Generated tests for methods involve to synthesize **this** object.

For plain JavaScript objects/records and arrays, the test generation follows a similar strategy. Due to the modeling of data structures as immutable values instead of heap references, the generated counterexamples deviate from standard JavaScript semantics with regards to aliasing. Similarly, counterexamples with cyclic references in data structures are not currently supported.

In order to generate counterexamples for higher-order functions, the test generator has to synthesize function values. Program synthesis is an active research topic with various different approaches and techniques [36, 3, 119] but ESVERIFY only supports a limited synthesis for the purpose of test generation. In particular, the synthesis of function values is limited to pure functions that map primitive argument values to return values. Thereby, the function can be expressed as a series of conditionals.

Listing 5.6 shows a generated test for the violated precondition of f(**null**) in line 9 of Listing 5.3. Here, a synthesized function value is assigned to f and then wrapped

111

```
1  let f = function (x) {          // synthesized function as counterexample
2    if (x === 3) {
3      return 9174;
4    }
5    return false;
6  };
7  f = function (x) {              // spec(f, ...) in line 7
8    assert(Number.isInteger(x)); // need to check precondition
9    return f(x);                 // but can assume postcondition
10 };
11 let x = 3;                      // x is a variable in scope but unused
12 f(null);
```

■ **Listing 5.6**: Generated test for the precondition of `f(null)` in line 9 of Listing 5.3.

according to the specification in line 7 of Listing 5.3, resulting in an assertion violation

when invoked with `null`. The synthesized function is based on a partial mapping and

might include constants picked nondeterministically by the SMT solver. Therefore, the

synthesized function may not adhere to the function specification when invoked with

other arguments not included in the SMT mapping. This is why the synthesized function

`f` returns a seemingly random number such as 9174 for the argument 3 but does not return

"correct" values for other arguments.

5.5   Generating Counterexample Function Calls

When asserting function specifications, the specification is transformed to a

wrapper in the generated test. However, without subsequent calls, the test would simply

end without triggering an assertion violation. Therefore, the test generator also needs

```
1   let f = function (x) {    // synthesized function as part of counterexample
2     if (x === -2) {
3       return -1;
4     }
5     if (x === -1) {
6       return 0;
7     }
8     return false;
9   };
10  const g = function (n) { // original function body of twice (line 9)
11    return f(f(n));          // use f(f(n)) here instead of f(f(null))
12  }
13  g = function (x) {          // spec(g, ... ) in line 12
14    const y = g(x);
15    assert(Number.isInteger(y) && y > 0);
16    return y;
17  };
18  g(-2);                      // synthesized function call
```

◾ **Listing 5.7**: Generated test for the postcondition in line 9 of Listing 5.3. In addition
to synthesizing f and wrapping the returned function g, it also generated
a call.

to synthesize a violation-provoking call after installing the wrapper for the asserted spec-
ification.

As an example, the postcondition of the twice function in Listing 5.3 does not
hold because the inner postcondition in line 12 is not satisfied by the returned function.
Listing 5.7 shows a simplified generated test with a synthesized function value for f, the
original function body of twice (assuming f(f(null)) is replaced by f(f(n))), and a
wrapper for the function specification in the postcondition of twice. Additionally, the
generated test also includes a synthesized function call g(-2) that causes an assertion

113

violation in line 15. The argument values for this violation-provoking function call are determined based on the SMT model.

5.6 Conclusion and Future Work

Program verifiers enable expressing and checking various correctness properties but understanding and debugging resulting verification errors can be difficult. Therefore, this chapter outlined an approach for automatically generating executable counterexamples.

Program verification with ESVERIFY is based on SMT solving. This has the advantage that the SMT solver also provides a concrete model for invalid verification conditions, i.e. it finds an assignment of free variables to values that serves as a counterexample. The raw output of the SMT solver includes low level verification details, so it needs to be translated to the level of the original source program to be useable by the programmer.

Testing can be a very effective way to present and explain these counterexamples as it directly concerns the execution of the code and it also enables the use of existing development tools such as step-by-step debuggers. Therefore, the ESVERIFY program verifier was extended to generate tests as executable counterexamples for invalid verification conditions. In generated tests, annotations such as invariants, pre- and postconditions are converted to dynamically-checked assertions and contracts. Additionally, functions, arrays and complex objects in the SMT model are translated to JavaScript expressions.

This chapter presents a solution for synthesizing functions as part of counterexamples such that synthesized functions always return the same result for the same argu-

ment values. Support for imperative programs is limited as this simple synthesis method does not generate functions that manipulate global state or object graphs with cyclic references. Related work on program synthesis already investigated these issues, so future extensions to the test generator could incorporate a more sophisticated program synthesis technique.

As explained in Section 5.2, not all failed verification conditions are caused by actual bugs in the code. It is possible for a correct program to be rejected by the program verifier due to a lacking annotation such as a loop invariant or postcondition. This poses a question to test generation as it could either result in a non-failing test that runs without assertion violations or a failing test that uses values from the verification logic instead of fragments from the actual source code. Both of these approaches offer different benefits, so ESVERIFY implements a compromise solution. Future research might help to guide the design of automatic test generators in the presence of this trade-off.

Finally, the design of the test generator has to take into account whether and how generated tests are presented to the user. One option is to manage generated tests like regular unit tests. Thereby, these tests could be debugged with existing development tools and even added to an existing test suite but it would also expose automatically generated code to the programmer that may not be very readable. Instead of displaying generated test code directly, the test generator described in this chapter is used internally as part of an integrated development and verification environment that is described in the following chapter.

115

So much of the way we work with systems today is derived from pencil and paper.
Even whilst we are at the computer we are still thinking in pencil and paper.
There is an incredible opportunity now to rethink how we think, about systems.

— Bret Victor

■ Chapter 6

Integrated Development/Verification Environments

This chapter describes a programming environment for verification that enables interactive inspection and experimentation with live feedback. Based on the program verifier described in Chapter 3 and the automatic test generation in Chapter 5, the programming environment integrates the code editing, verification and debugging. In particular, it explains verification errors with concrete values that serve as counterexample, it lets programmers step through the relevant parts of the code leading up to the verification error with these values, and it enables live edits to assertions and assumptions of verification conditions. Thereby, the programmer can determine the cause of verification issues and narrow down their scope without having to manually add `assert` statements to the code. This kind of environment integration is analogous to step-by-step debuggers that let programmers explore the execution state of a program without having to resort to "printf debugging".

Additionally, this chapter also describes an online user study where participants

solved small programming and verification tasks and answered survey questions to evaluate how the development and verification features of the environment were used and perceived.

6.1   Overview

Program verifiers such as ESVERIFY enable the programmer to express correctness properties in an expressive assertion language, often based on first-order logic. However, the verification process for these assertions is relatively complicated and verification issues can become difficult to understand. In these cases, simple feedback mechanisms in the form of error messages may not be sufficient to help the programmer resolve these verification issues. In order to improve the programmer experience for verified programming, this chapter presents IDVE, an integrated development and verification environment that lets users interactively inspect and debug verification issues. The goal of IDVE is to support programmers with an interactive interface for understanding and interactively manipulating verification conditions. An essential component of this integration is the automatic generation of counterexamples for verification errors as described in Chapter 5.

As a brief overview, it is useful to illustrate the scope and goals of the proposed programming environment with an example, shown in Figure 6.1. Here, a JavaScript function `abs` is annotated to indicate that its result is always non-negative. However, due to a bug in the `abs` function, the result might be negative, violating the postcondition in
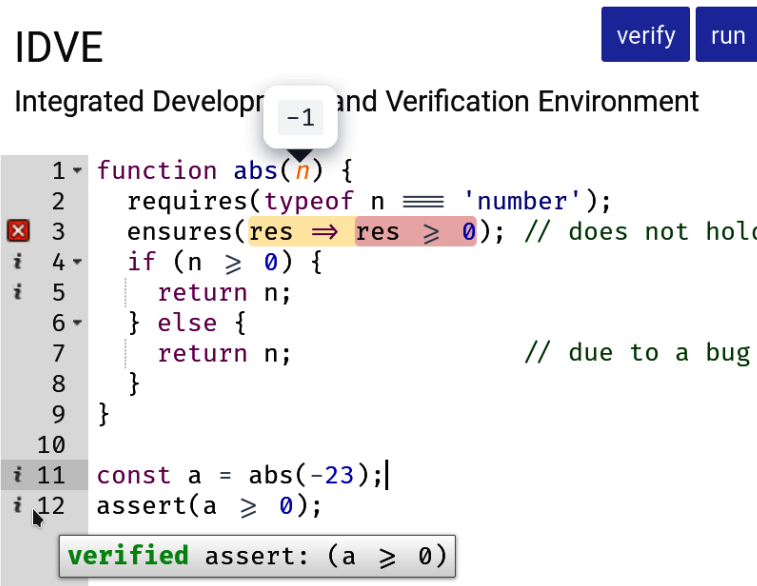
Figure 6.1: IDVE displays verification conditions for this annotated JavaScript program as line markers. The assertion in line 12 can be statically verified but a bug in line 7 causes a verification error for the postcondition in line 3, so IDVE shows -1 as counterexample for n.

line 3.

The prototype implementation of the integrated development and verification environment, abbreviated as IDVE, helps the programmer identify verification conditions and inspect potential verification errors. Figure 6.1 does not show the full programming environment, but it illustrates how symbols next to the line numbers are used to indicate verification conditions. Hovering over these marks with the mouse cursor display additional details – similar to type errors. For failed verification conditions, IDVE also displays counterexample values as editor popups. For example, it displays -1 as a value for the function argument n that causes a violation of the postcondition.

Additionally, IDVE also enables programmers to inspect specific verification con-

ditions by opening an interactive inspector panel (not shown in Figure 6.1) that lets users inspect, add and remove assumptions and assertions – similar to "watch expressions" in an interactive debugger. Thereby, the verification inspector allows programmers to explore the verifier state without manually adding `assert` statements to the code, analogous to how interactive debuggers let programmers avoid `printf` debugging. The environment also includes an integrated debugger for the automatically generated test cases that lists variables in scope, shows the current call stack and allows step-by-step debugging.

Finally, the environment and its usability for developing verified programs was evaluated with a user study with 18 participants that have at least basic knowledge of JavaScript. The test subjects were given a brief introduction to the features of IDVE, had to solve a series of simple programming tasks with the environment[1], and answered a brief survey about their experience[2]. Results indicate that more than half of the participants were able to use the features of IDVE effectively to solve the programming and verification tasks. All participants reported that they found the tools either helpful or potentially helpful. However, an improved user interface design might enable more programmers to successfully use these features.

---

[1] The tutorial steps as well as the experiments are listed in Appendix B and an archived version of the user study is available online at `https://esverify.org/userstudy-archived`.

[2] Survey results are included in Appendix C.

6.2 Environment Integration

In order to be useful in practice, program verification has to be integrated into the development process. Ideally, feedback provided by the verifier should be instantaneous, continuous, informative, comprehensible and actionable. Instantaneous feedback requires the verification procedure to be fast enough to avert noticeable delays. This also enables continuous feedback by implicitly invoking the verifier after each code change. Most program verifiers, including ESVERIFY, can check small to medium source files in less than a second and thereby enable sufficiently fast feedback. Providing comprehensible and actionable feedback, in contrast, is still a major challenge for program verification because the complexity of the verification procedure can result in errors that are hard to understand. IDVE is an integrated development and verification environment that integrates an interactive verification inspector and a counterexample debugger to address this issue. The implementation of IDVE is open source[3] and a live demo is available online[4].
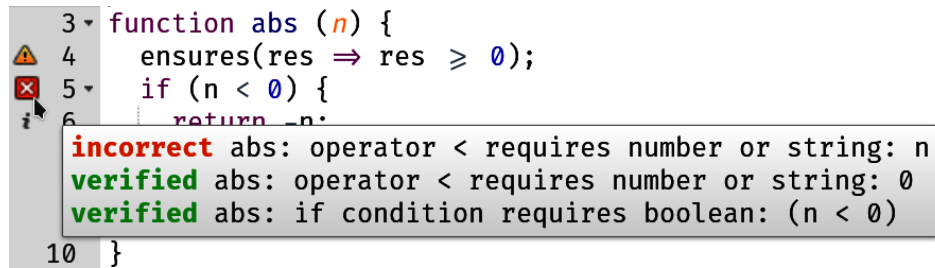
6.2.1 Basic Line Markers

Each verification condition identified by the verifier corresponds to a location in the source code, such as a postcondition of a function definition, a function call that has to satisfy a precondition, or the arguments of a binary operator which have to adhere to a certain type.

---

[3]Implementation source code: `https://github.com/levjj/esverify-web`
[4]Live demo of IDVE: `https://esverify.org/idve`

```
 3 ▾  function abs (n) {
 4       ensures(res ⇒ res ⩾ 0);
 5 ▾     if (n < 0) {
 6          return -n;
```

incorrect abs: operator < requires number or string: n
verified abs: operator < requires number or string: 0
verified abs: if condition requires boolean: (n < 0)

```
10    }
```

■ **Figure 6.2**: Verification conditions displayed as line markers with short error messages displayed as tooltips. Due to a missing precondition, the value of n may be incompatible with the + operator.

As a basic form of environment integration, the source location of a verification condition can be displayed as an annotation or line marker in the code editor. Figure 6.2 shows an example of an editor with line markers to indicate verification conditions, using different icons for verified, unverified and incorrect results. Here, a result is considered *incorrect* if the generated test causes an assertion violation while an *unverified* result might have failed verification due to a missing loop invariant rather than an actual bug in the code. In addition to the icon, these line markers also include a short message that can be displayed by hovering the mouse cursor over the icon.

In addition to verification errors, this type of feedback is also common for type errors. Indeed, the line markers in IDVE are also used for other errors such as parsing and scoping issues.

While this integration is relatively simple, non-intrusive and self-explanatory, the provided feedback is limited and may not be sufficiently detailed to help the programmer understand and fix potential verification issues.

121
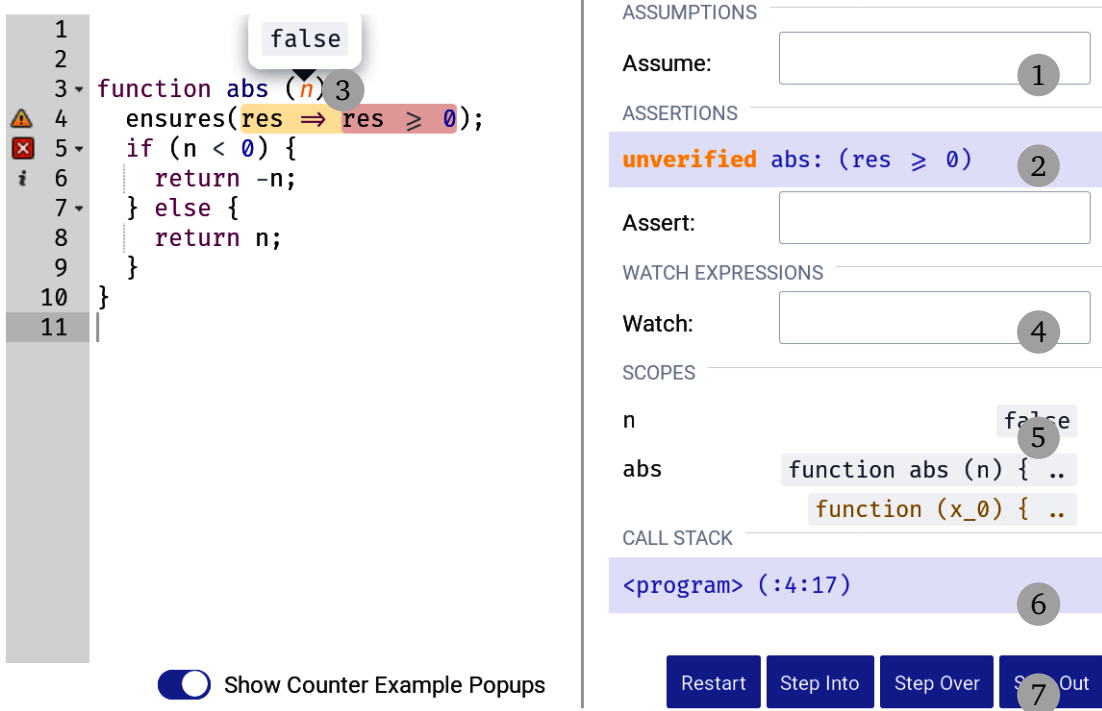
Figure 6.3: Selecting the unverified verification condition in line 4 opens a verification inspector on the right, showing assumptions, assertions and a debugger for the counterexample.

6.2.2 Verification Condition Inspector

By selecting one of the line markers discussed in the previous section, an additional panel can be opened with an interactive "inspector" for verification conditions.

Figure 6.3 shows an example with an active verification inspector. Here, the verification condition in line 4 cannot be verified and is selected in the editor on the left. The panel on the right then allows interactive inspection of the verification condition.

In particular, the inspector lists assumptions (1) such as preconditions and invariants. While there are no relevant assumptions in this example, the user can enter

an assumption in the form of boolean expressions and add it to the verification context for the selected verification condition. For example, by entering the JavaScript expression `typeof n === 'number'`, the verification condition will be re-examined with the new assumption, causing the verification of `res >= 0` to succeed.

Similarly, the inspector displays the asserted proposition ② but also allows additional assertions to be entered by the user and tested for the same assumptions and context. This feature of the verification inspector can be useful for interactive exploration and experimentation with the verification context without having to change the original source code (analogous to how interactive debuggers supersede "printf debugging"). This feature is novel in verified programming environments as existing environments such as Dafny IDE only display information about verification conditions without providing ways to interactively alter assumptions and assertions with a verification inspector.

6.2.3 Counterexample Popups

For each failed verification condition, a counterexample is synthesized based on the SMT solver output. This counterexample includes concrete JavaScript values for free variables such as function arguments and mutable variables in the surrounding scope. IDVE displays these counterexample values for the currently selected verification condition as popups ③ in the editor. These popups are directly connected to the relevant variable or parameter definition but they are also obscure the source code below and they only display short summaries that may be inadequate for complex values such as nested objects and arrays. Visualizations of nested data structures is also an important challenge

123

for regular debuggers and development tools [42].

6.2.4 Debugger Integration

Even with information about values of free variables such as those displayed with editor popups, it may not be obvious why a postcondition may not hold — especially for longer functions and methods.

In these cases, it might be helpful to inspect the current values of variables at different points in the function body. In general, this can be achieved by using an interactive step-by-step debugger. In the context of a failed verification condition, a counterexample test can be automatically generated for the purpose of debugging (see Chapter 5). Running this test will either result in an assertion violation, which serves as a concrete witness for a bug in the code or annotations, or a successful test execution without error, which indicates that a false positive was caused by the conservative static analysis of loops and recursion. In both cases, stepping through the code can help with understanding the verification issue and locating the root cause of the bug.

There are two possible approaches for debugging the counterexample test. On the one hand, the generated test could be "exported" and debugged with a traditional debugger in a separate tab or window. This would ensure a traditional debugging experience and could even enable the generated test to be added to an existing unit test suite. However, this approach requires the user to switch contexts between the original code and the test. Moreover, this approach exposes the automatically generated test code which may not be human readable and whose mapping to the original source code may

not be obvious.

On the other hand, it is possible to hide the automatically generated test code and debug the counterexample directly on the level of the original source program. Essentially, the debugger internally steps through the generated test while highlighting expressions and statements in the original source code that correspond to the current code fragment in the test. This approach avoids context switches but it might cause an unexpected order of execution steps in the visible source code if the control flow in the generated test differs from the control flow in the original program due to inserted dynamic checks or contract wrappers.

Figure 6.3 shows an integration using the latter approach. Here, the debugger is halted at the assertion `res >= 0`, and IDVE shows watched expressions 4, variables in scope 5, and the call stack 6. The user can add additional watch expressions and display their evaluation results. For example, entering `res` in 4 will show **false** as the returned value for this counterexample. Additionally, the execution can be stepped using standard debugger controls 7.

In contrast to debugging regular executions, the integrated debugger of the verification environment includes both a dynamic context from the actual test execution as well as a static context representing the verifier state at different points in the code. For example, for `abs`, the scope panel 5 contains both the function value used by the test execution in black and a synthesized function value in brown below. Only a single value is displayed if the value from the dynamic and static contexts agree. While this information is not relevant in this example, comparing the dynamic and static value of a

mutable variable after a loop may help to understand missing or incorrect loop invariants (see Section 5.2).

Internally, the debugger is implemented as an interpreter that operates directly on the JavaScript AST of the test code. This incurs a high performance penalty over standard debuggers and techniques such as source-to-source compilation [9] but it provides greater flexibility for incorporating additional features. For example, the IDVE debugger maintains both a dynamic execution and a static verifier context and it provides a better debugging experience for stepping through function wrappers.

6.3 Evaluation and User Study

IDVE was evaluated with a user study with 18 participants. This section describes the relevant research questions, the design of the user study, its results, and potential threats to validity.

6.3.1 Research Questions

In contrast to the program verifier itself, the design of the integrated development and verification environment is difficult to evaluate due to the subjective experience of programmers and the large solution space. This user study aims to provide insight into answering the following three research questions in order to inform future designs of such environments.

RQ1: Can IDVE assist in the development process? Do programmers use features such as line markers, interactive manipulation of assumptions and assertions, counterexample editor popups and integrated debugging if these features are available for solving a given task?

RQ2: Is the proposed user interface helpful and intuitive? Careful consideration is required for the design of user interfaces of development environments in order to balance the amount of information and interactive controls. Therefore, this user study should determine whether the proposed design is generally perceived as intuitive or as overwhelming and cumbersome.

RQ3: How does programming proficiency and prior experience with program verification affect utility? The proposed environment should ideally be accessible and usable by both novices and experienced programmers. However, programming expertise and experience with program verification might be a prerequisite for effectively using the proposed features of the environment.

6.3.2 Methodology

IDVE is still an early prototype and not yet ready for productive software development. Therefore, the user study focuses on how proposed features can be used for smaller programming and verification tasks. Test subjects had no prior experience with IDVE itself but indicated to have at least basic knowledge of JavaScript and might have used other program verifiers before.

127

The user study was conducted entirely online. Subjects were recruited with an invitation sent to a public mailing list and remained anonymous. 18 adults entered the study and were presented with a brief introduction of IDVE, followed by a series of programming and verification tasks, and finally surveyed about their experience using the tool.

Appendix B lists both the tutorial steps as well as instructions, provided code and hints for the experiments. Additionally, an archived version of these tasks is available online at `https://esverify.org/userstudy-archived`.

For the first step, a guided tutorial introduced

- the source code editor itself,

- a simple verification example similar to the one in Figure 6.1,

- an interaction with the verification inspector as described in Section 6.2.2, and

- an interaction involving stepping through a generated test with the integrated debugger discussed in Section 6.2.4.

After the tutorial, participants solved three short tasks:

1. The first experiment involves an incorrect factorial function that causes an infinite recursion for negative arguments. This task can either be solved by changing the precondition or by changing the implementation of the function. Participants could use the verification inspector and the integrated counterexample debugger but editor popups were disabled.

128

2. For the second experiment, a correct implementation of six-sided dice rolling function was given. However, the function was missing postconditions necessary for verification of the subsequent code. Editor popups and the verification inspector were both disabled, so only basic line markers were available for this experiment.

3. The third and final experiment involved a function for converting the number of minutes since midnight into a 24-hour digital clock format. The provided code included bugs in both the annotations as well as the implementation. Both the verification inspector and editor popups were available but the integrated debugger was disabled.

All experiments could be skipped at any time and did not measure time or success. Instead, test subjects proceeded to the next experiment at their own discretion and filled out a survey form about their experience at the end.

6.3.3 Results

A full record of survey answers for all 18 participants including written comments can be found in Appendix C. These answers provided empirical evidence towards answering the research questions above.

RQ1: Can IDVE assist in the development process? According to the survey results shown in Table 6.1, the usage and the perceived benefits vary for the three main features of IDVE. While half of the participants made use of counterexample popups, the verification inspector was only used by 39 percent of participants, and the integrated debugger

Response (%)	Verification Environment Feature		
	Verification Inspector	Counterexample Popups	Integrated Debugger
Used this feature in experiments	39	50	28
Unsuccessfully tried using it	33	33	22
Did not use it	27	17	50
The feature is helpful	33	55	44
It could be helpful with different UI	50	39	44
It is not useful for development	6	6	6
It impairs the development process	11	0	6

■ **Table 6.1**: Participants indicated which features were used in the experiments and whether these features are seen as helpful.

by 28 percent. In total, 15 participants reported using at least one of the tools. When features were used, they are generally seen as helpful or at least potentially helpful. This indicates that IDVE can effectively assist programmers.

RQ2: Is the proposed user interface helpful and intuitive? As shown in Table 6.1, the verification inspector, the counterexample editor popups and the integrated debugger were considered helpful by 33/55/44 percent of the subjects. Another 50/39/44 percent reported that these features could be helpful with an improved user interface. Additionally, a third of the subjects tried to use the verification inspector and the counterexample popups but reported being unsuccessful. Overall, the results suggest that the

user interface is an important factor for using verification in practice.

Incidentally, half of the participants did not try to use the integrated debugger despite considering it helpful or potentially helpful. This might be a result of the experimental setup with programming tasks that were too trivial to require debugging but it might also indicate that the debugger integration might benefit the most from user interface improvements.

RQ3: How does programming proficiency and prior experience with program verification affect utility? As part of the user study, participants ranked their JavaScript proficiency on a scale from 1 (novice) to 5 (expert) and indicated whether they had prior experience with program verification. Table 6.2 shows how this related to usage and perceived benefits of IDVE. Here, it is most noteworthy that all participants, including those without JavaScript expertise or prior verification experience, found at least one of the the features helpful. Also, no significant differences were reported on the actual usage of these features in the experiments. While these results suggest that IDVE is accessible for both beginner and experienced programmers, the number of test subjects may be too low to fully support these conclusions. In fact, three of the participants remarked in written feedback that a more comprehensive tutorial about program verification would have been helpful.

Experience with	JavaScript					Verification	
	1	2	3	4	5	no	yes
# Participants	1	2	3	6	6	8	10
Successfully used features	1	2	3	5	4	6	9
Sees features as potentially helpful	1	2	3	6	6	8	10

■ **Table 6.2**: Usage and perception of verification environment features in relation to self-proclaimed proficiency.

6.3.4 Threats to Validity

The user study had a limited scope with only three short programming tasks and 18 participants. Therefore, it is possible that a broader user study with larger programming projects and more participants would yield different results. However, while scalability could be a concern for the runtime performance of the verifier, it can be expected that the responses of this user study regarding usability are at least indicative of a general trend that would also be observed by a larger user study.

Additionally, the results of this user study might be specific to JavaScript. It is possible that this approach for an integrated development and verification environment would be inadequate or unpractical for a different programming language or a different domain. For example, the integrated debugger for automatically generated tests may not be applicable to programs involving concurrency or input/output to external services or components.

6.4 Future Work and Conclusions

Program verifiers enable expressing and checking various correctness properties but understanding and debugging resulting verification errors can be difficult. To assist programmers, this chapter proposed an integrated development and verification environment and discussed its features.

The environment enables inspection of verification conditions including the option to interactively add or remove assumptions and assertions. Additionally, the environment provides executable counterexamples and step-by-step debugging for failed verification conditions based on automatically-generated tests.

To evaluate this approach, a user study with 18 participants was conducted. The proposed development and verification environment is generally seen as helpful, especially its feature for displaying counterexample values as editor popups. However, the user study also found that interface design is an important factor that could be improved to ensure that the proposed environment integration is useful in practice.

There are still open questions that could be addressed by future work such as whether this approach scales to larger applications with multiple developers and how it can be applied to other domains and programming paradigms.

In a way, ideas only count for a little in computing,
because you kind of have to implement the stuff.
This is the part of the story that really makes me clutch at my throat, because every
time you implement something, five years go away – and you do learn something.

— Alan Kay

■ Chapter 7

Related Work

This chapter outlines and briefly discusses relevant publications and important related research projects in the areas of live programming, program verification, automatic test generation and integrated verification tools.

7.1   Live Programming

The term *live programming* is used in different contexts and can denote the act of programming as part of a live art or music performance but it can also refer to programming environments that continuously evaluate expressions and display the results alongside the code. In the context of this thesis, live programming is the ability to change the code of a running *stateful* application without restarting it, also known as *hot-swapping* or *dynamic software update* [47], while providing immediate and continuous feedback about these code changes. Such live programming systems were described and motivated by Hancock [43] and further explored by systems like Subtext [29], which uses a

tree representation of constantly executing expressions, SuperGlue, which uses dynamic inheritance and functional reactive programming [74], and Elm, which demonstrates live programming and time traveling with first-order functional reactive programming [25].

The solution described in Chapter 2 is based on earlier work that requires UI rendering and stateful computation to be separated and that prohibits function values (closures) in the application state to allow the state and the code to be updated independently. A paper by Burckhardt et al. describes this idea for TouchDevelop [14]. More recent work outlined the possible design space between live programming systems that resume computation (with a possibly inconsistent state) and systems that record and replay execution [75], as well as introducing *managed time* as concept for supporting both live programming and time travel [76] and approaches that combine the language design with the design of the programming environment [66].

The main contribution of the live programming research for this thesis is the design of a simple programming model based on the Model-View-Update pattern and an environment integration for JavaScript that offers similar live programming capabilities as Elm and TouchDevelop [14] without requiring special syntax, type annotations or library integration. Additionally, the proposed approach can also be easily extended to support back-in-time debugging and runtime version control, i.e. the ability to navigate to past execution states and past code versions while providing continuous feedback.

Programming-by-example allows users to create and modify programs by providing examples instead of editing the source code. Prior work on programming-by-example and programming-by-demonstration ranges from domain-specific macro sys-

135

tems, visual programming systems and automatically inferred string processing rules [70, 41] to depth-limited generate-and-test approaches for general-purpose programming languages. Most noteworthy, CodeHint [36] synthesizes short Java code snippets at runtime based on user-provided queries. In order to synthesize larger code snippets, SMT solver-aided approaches may be a promising alternative [111, 3]. In the end, the use of SMT solvers for program synthesis is closely related to their use for program verification. Indeed, program synthesis can be seen as generalized program verification where the output is not an error message or counterexample but an inferred code candidate that repair the input program to satisfy the specification [104, 80].

Chapter 2 also presented an approach for live programming by *direct manipulation*. Direct manipulation of the graphical user interface is a well-known form of user interaction [101] with popular applications in Smalltalk [40], Morphic [73] and others. However, direct manipulation usually only affects the current state of visible objects. Recent work on *prodirect manipulation* [22] shows how direct manipulation of SVG vector graphics can be used to automatically modify the SVG rendering code. The same idea has also been applied to the manipulation of string constants in PHP web applications [118]. These two projects are most closely related but, in contrast to the approach in Chapter 2, they do not support live code updates of stateful applications without restarting the execution.

7.2 Program Verification

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 [32, 63], JaVerT [34], Dafny [61, 62, 60] and LiquidHaskell [116, 114, 115].

ESC/Java [32] 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 [34] 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 programs include Loop-Sensitive Analysis [87], the TAJS Type analyzer for JavaScript [4], as well as type systems such as TypeScript, Flow and Dependent JavaScript [21]. ESVERIFY follows as different approach as it relies on manually annotated assertions that are generally more expressive than types.

Dafny [61] seeks to provide a full verification language, with support for both functional and imperative programming. Dafny offers programmers advanced constructs 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 3.4.4. 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 4.8 loosely resembles its formalization by Vazou et. al. [116]. More recently, LiquidHaskell introduced *refinement reflection* [115], 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 4.3 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 4.3, has been studied by extensive prior work [38, 90, 27, 64]. 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.

7.3 Automatic Test Generation

While the goal of this thesis is focused more on programming environments, Chapter 5 described the automatic test generation of verification counterexamples as a basis for further environment integration. Counterexample generation is an essential concept for both SMT solving and theorem proving [24]. Automatic test generation, i.e. generation of executable counterexamples, extends this idea and is a common technique for both program analysis and verification.

For program analysis, techniques such as symbolic execution, "whitebox testing" and parameterized testing [37, 110, 109, 18, 99] use path conditions to reason about control flow and generate random inputs in order to explore more program paths and thereby achieve higher test coverage. The approach in this thesis, however, is based on formal verification rather than symbolic execution. Here, the main difference is the reasoning about loops and recursion. While verification requires manual annotations to avoid false positives, symbolic execution approximates the program behavior and may produce false negatives.

On a side note, Chapter 3 targets JavaScript, a dynamically-typed language, but automatic generation of error witnesses has also been explored for type errors in OCaml [97].

Furthermore, Heidegger and Thiemann previously presented a system for annotating JavaScript code with contracts that are used for guided random testing [44]. However, instead of using a static analysis such as program verification, the contracts

have to be labelled explicitly by the programmer to guide the random testing.

Similarly, Klein, Flatt, and Findler proposed a system in which test inputs are generated for stateful programs by randomly creating primitive values and objects, calling random functions and methods available in the current context and synthesizing function bodies [55]. This approach is similar to the test generation presented in Chapter 5. However, it provides better support for object-oriented programs (see also Thummalapenta et al. [108]) and it might not be able to explore deeper issues due to the black box generation of test inputs.

Automatic test generation has also been previously explored in the context of program verification in systems such as KeY [11], for techniques such as abstract interpretation [120], and for runtime verification [106].

For partially verified code, Christakis, Müller, and Wüstholz proposed to use residual assumptions for complementary checking and dynamic testing in order to find more errors [19, 18].

Alternatively, conditional model checking is based on a condition that models control flow for properties that cannot be verified. This condition can either be translated back into a program that represents the unverified parts, or, alternatively, the unverified assertions can be used as slicing criterion for the original program. The residual program is then used for dynamic testing [26]. While conceptually similar, the automatic test generation presented in this paper targets higher-order functional programs and requires explicit invariants for loops.

More recently, Nguyen and Van Horn presented an approach for generating

counterexamples for high-order functional programs. The paper uses a restrictive core language similar to simply-typed lambda calculus and finds counterexamples by SMT solving with an approximation relation for stateful programs [81].

Finally, for partially verified programs, verified assertions and invariants that depend on unverified verification conditions are vulnerable, so robustness testing might be applicable in these cases [49].

7.4   Integrated Verification Tools

There has also been prior work on debugging tools in the context of program verification.

Even without static analysis, annotations such as function contracts may benefit from tool support such as dynamic activation/deactivation [48] and dedicating debugging features [5].

Tymchuk, Ghafari, and Nierstrasz recently investigated the integration of static analysis in a development environment by conducting interviews and concluded that this integration is essential for adoption [112].

For program analysis with symbolic execution, Hentschel, Bubel, and Hähnle recently presented a Symbolic Execution Debugger that is integrated into Eclipse and offers a similar user experience to traditional debuggers [46].

In the context of program verification, work on dedicated debugging tools overlaps with research on interactive theorem provers. For example, the Proof Script Debugger for the KeY System [10] offers step-by-step debugging for proof scripts and inspection

of the verifier state as assistance for theorem proving.

The most closely related research in the context of integrated development and verification environments is the work on tools for Boogie, Dafny and Viper [107]. In particular, Le Goues, Leino, and Moskal presented a verification debugger for Boogie, a low-level verification language [59] used internally by verifiers such as Dafny. At the same time, Dafny programs can also be debugged with an integrated development environment called Dafny IDE [65, 20] that addresses feedback about verification issues similar to the environment presented in Chapter 6. The Dafny IDE also allows inspection of counterexamples, including the state of variables at different stages in the verification of a function body. However, in contrast to the environment presented in Chapter 6, it does not enable the user to interactively modify assumptions and assertions of a verification condition.

Computers are useless. They can only give you answers.

— Pablo Picasso

■ Chapter 8

Conclusions

Computer programming is the act of specifying the intended behavior of software. The translation of the mental model into executable program code is challenging. Moreover, discrepancies between intentions and actual program behavior can be difficult to understand and to fix. This is an inherent problem of programming that persists despite future advances in artificial intelligence.

However, it is possible to assist programmers with programming environments and programming language features. For example, live programming enables a faster feedback cycle by allowing live code edits to running programs and program verification lets programmers specify the expected behavior with logical propositions that will be checked against the executable code to automatically detect errors. Neither of these two techniques is new but both are currently gaining renewed interest by both academic researchers and professional programmers. While these concepts are to some extent orthogonal, a live programming perspective could inform the design of a programming environment for verification with an emphasis on interactive experimentation and con-

tinuous feedback. This poses the following question:

> "How can live programming environments support verification to provide a better programming experience?"

8.1 Discussion of Research Method

In order to answer the question above, the research for this dissertation involved design proposals, implementations, formal developments, and empirical evaluations.

More concretely, the research was primarily guided by prototype implementations that demonstrate how certain features would affect programming for smaller programming problems and example applications. Building simple programming environments from the ground up has both advantages and disadvantages over adopting and extending existing solutions. On the one hand, it enables quick experimentation in a large design space with minimal constraints about the language or the user interface. Also, small online live demos might reach a wider non-academic audience and thereby illustrate the idea and feasibility without the need to install local software. On the other hand, the custom programming environments implemented for this thesis research have no existing users and are generally not mature enough for productive use on real-world projects. This makes usability aspects difficult to evaluate, especially with regards to scalability for larger programs, coordination within programming teams and long term benefits and costs for the software development process.

In addition to prototype implementations, the proposed designs were also formally defined. These formal definitions describe the core idea in a minimal and concise

144

way, and they enable proofs of properties independent of concrete specifics of existing systems or programming languages. However, these definitions may not perfectly model the actual system and are also relatively dense and hard to understand especially for non-academic readers.

The author of this dissertation also used this method for other topics in programming language research that is not included in this thesis, such as

- work on a light-weight static *effect system* for JavaScript based on SMT solving [95],

- a project involving declarative definitions of time-varying values and their dependencies with reactive variables [96], and

- a refactoring method for replacing code with macro invocations by running macro expansion in reverse [94].

8.2   Summary of Results

The live programming system presented in Chapter 2 is based on ideas from functional and reactive programming, and ensures continuous feedback for live code updates similar to prior work by Burckhardt et al. and others [14]. However, the proposed system is based on event-based UI programming in JavaScript and the Model-View-Update pattern to separate rendering from event-handling. Thereby, the programming environment can support back-in-time debugging, runtime version control and live programming-by-example. The latter enables live code edits based on direct manipulation of the output by the user.

The work on program verification is partly inspired by existing program verifiers such as Dafny [61]. However, heuristics and inference techniques in program verifiers can result in a brittle verification process without means to inspect and understand potential verification issues. Furthermore, Dafny requires type annotations in addition to preconditions and thereby introduces complex interactions between type checking and verification. Based on a novel approach for verification, the program verifier described in Chapter 3 enables verification of JavaScript programs, including higher-order functions, without traditional type annotations or typing rules. The underlying quantifier instantiation algorithm ensures that the verification process remains predictable and avoids timeouts. Finally, a formal development given in Chapter 4 proves that evaluation of verified programs adheres to the annotations.

This program verifier forms the basis for an integrated development and verification environment described in Chapter 6. By integrating directly into the verification process, the environment can support features that may not be easy to implement for existing program verifiers or environments. While the original goal was to create a live programming environment for verification, live code updates are equivalent to regular code updates for the purpose of static verification, so live programming does not directly interact with program verification. Nevertheless, the design of the environment is informed by live programming in the sense that experimentation and immediate feedback are primary objectives for its integrated debugging features. In particular, the environment displays details about verification conditions (similar to the Dafny Verification Debugger [65]) and it also enables live updates to assumptions and assertions of verification

conditions. Thereby, programmers do not have to add `assert` statements to the code, re-verify the program and switch contexts away from the verification inspector to obtain feedback. Any failed verification condition can also be examined by the means of an executable counterexample that enables step-by-step debugging through an automatically generated test case (see Chapter 5). Finally, a user study with 18 participants with and without prior program verification experience found that a majority of participants were able to effectively solve small programming and verification tasks with the environment. The environment was generally seen as helpful or potentially helpful with the user interface design as an essential factor for utility.

8.3 Future Work

The theoretic work on formalizing the program verifier in the Lean theorem prover enables both a soundness theorem for verified programs as well as a comparison with refinement type systems. In particular, it is possible to define a translation from types to annotations and both cursory paper proofs and experimental evidence suggest that any well-typed program translates into a verifiable program (see Section 4.8). However, a full formal proof of this conjecture in an automated theorem prover remains open for future work. The main difficulty encountered while working on this proof is the quantifier instantiation algorithm and the quantifier nesting bound. It is possible that an alternative algorithm without reference to the quantifier nesting bound would simplify a potential proof in a way that its result could be extended to the algorithm presented in this thesis.

A second area for future research is an integrated approach for using examples alongside verification annotations as a basis for synthesis and program repair. It is noteworthy that both the live programming-by-example mechanism in Section 2.4 and the test generation in Section 5.4 synthesize program code for different purposes. Ideally, program verification could be extended such that the output is not just an error message or a counterexample but a code candidate that repairs the program according to examples and specifications. Essentially, formal specifications can be used to restrict the search space for program synthesis analogous to examples in programming-by-example systems. Particularly GUI applications might benefit from such an approach as concrete examples are better suited for specifying the intended look and behavior of the GUI while verification annotations are more appropriate for documenting and specifying application logic. Therefore, an integrated approach for program synthesis and repair based on example and verification would combine both advantages.

Finally, the program verifier and the programming environments developed as part of this dissertation research are lacking some functionality that would be essential for productive use, such as termination checking, module imports and exports, and a preamble with verification definitions for all global objects, methods and functions in JavaScript. Continued work on these implementations would make it possible to evaluate the approach for developing larger applications. Similarly, a larger user study with teams of programmers might yield more insights about the potential benefits and limitations about live programming, program verification and their integration in a programming environment.

∎ Appendix A

Formal Definitions and Theorems in Lean

This appendix contains formal definitions and theorems for verification of λ^S. The definitions and theorems are given in Lean, an open source theorem prover based on dependent type theory [78].

This appendix does not list all proof steps and auxillary lemmas due to the limited space. However, the full proof is available online at

`https://github.com/levjj/esverify-theory/`.

A.1 `syntax.lean`

This file includes inductive definitions of syntactical objects such as values, expressions, terms and propositions.

```
1  -- syntax for values. expressions, terms, propositions, etc.
2
3  -- x ∈ VariableNames
4  @[reducible]
5  def var := ℕ
6
```

```
7   -- ⊗ ∈ UnaryOperators
8   inductive unop
9   | not    : unop
10  | isInt  : unop
11  | isBool : unop
12  | isFunc : unop
13
14  -- ⊕ ∈ BinaryOperators
15  inductive binop
16  | plus  : binop
17  | minus : binop
18  | times : binop
19  | div   : binop
20  | and   : binop
21  | or    : binop
22  | eq    : binop
23  | lt    : binop
24
25  mutual inductive value, exp, term, spec, env
26
27  -- v ∈ Values := true | false | n | <func f(x) R S {e}, σ>
28  with value: Type
29  | true  : value
30  | false : value
31  | num   : ℤ → value
32  | func  : var → var → spec → spec → exp → env → value
33
34  -- e ∈ Expressions := ...
35  with exp: Type
36  | true   : var → exp → exp                          -- let x = true in e
37  | false  : var → exp → exp                          -- let x = false in e
38  | num    : var → ℤ → exp → exp                      -- let x = n in e
39  | func   : var → var → spec → spec → exp → exp → exp -- let f(x) R S = e in e
40  | unop   : var → unop → var → exp → exp             -- let y = op x in e
41  | binop  : var → binop → var → var → exp → exp      -- let z = x op y in e
42  | app    : var → var → var → exp → exp              -- let z = x(y) in e
43  | ite    : var → exp → exp → exp                    -- if x then e else e
44  | return : var → exp                                -- return x
```

150

```
45  -- A ∈ LogicalTerms := v | x | ⊗A | A ⊕ A | A(A)
46  with term: Type
47  | value : value → term
48  | var   : var → term
49  | unop  : unop → term → term
50  | binop : binop → term → term → term
51  | app   : term → term → term
52
53  -- R,S ∈ Specs := A | ¬ R | R ∧ S | R ∨ S | spec A(x) req R ens S
54  with spec: Type
55  | term : term → spec
56  | not  : spec → spec
57  | and  : spec → spec → spec
58  | or   : spec → spec → spec
59  | func : term → var → spec → spec → spec
60
61  -- σ ∈ Environments := ∘ | σ[x ↦ v]
62  with env: Type
63  | empty : env
64  | cons  : env → var → value → env
65
66  -- s ∈ Stacks := (σ, e) | s · (σ, let y = f(x) in e)
67  inductive stack
68  | top  : env → exp → stack
69  | cons : stack → env → var → var → var → exp → stack
70
71  -- P,Q ∈ Propositions := A | ¬ P | P ∧ Q | P ∨ Q | pre(A, A) | pre(⊗, A) |
72  --                       pre(⊕, A, A) | post(A, A) | call(A) |
73  --                       ∀x. {call(x)} ⟹ P | ∃x. P
74  inductive prop
75  | term     : term → prop
76  | not      : prop → prop
77  | and      : prop → prop → prop
78  | or       : prop → prop → prop
79  | pre      : term → term → prop
80  | pre₁     : unop → term → prop
81  | pre₂     : binop → term → term → prop
82  | post     : term → term → prop
```

151

```
 83  | call    : term → prop
 84  | forallc : var → prop → prop
 85  | exis    : var → prop → prop
 86
 87  -- A[∘] ∈ TermContexts := ∘ | v | x | ⊗ A[∘] | A[∘] ⊕ A[∘] | A[∘] (A[∘])
 88  inductive termctx
 89  | hole   : termctx
 90  | value  : value → termctx
 91  | var    : var → termctx
 92  | unop   : unop → termctx → termctx
 93  | binop  : binop → termctx → termctx → termctx
 94  | app    : termctx → termctx → termctx
 95
 96  -- P[∘], Q[∘] ∈ PropositionsContexts := A[∘] | ¬ P[∘] | P[∘] ∧ Q[∘] |
 97  --        P[∘] ∨ Q[∘] | pre(A[∘], A[∘]) | pre(⊗, A[∘]) | pre(⊕, A[∘], A[∘]) |
 98  --        post(A[∘], A[∘]) | call(A[∘]) | ∀x. {call(x)} ⟹ P[∘] | ∃x. P[∘]
 99  inductive propctx
100  | term    : termctx → propctx
101  | not     : propctx → propctx
102  | and     : propctx → propctx → propctx
103  | or      : propctx → propctx → propctx
104  | pre     : termctx → termctx → propctx
105  | pre₁    : unop → termctx → propctx
106  | pre₂    : binop → termctx → termctx → propctx
107  | post    : termctx → termctx → propctx
108  | call    : termctx → propctx
109  | forallc : var → propctx → propctx
110  | exis    : var → propctx → propctx
111
112  -- call(x) ∈ CallTriggers
113  structure calltrigger := (x: term)
114
115  -- P,Q ∈ VerificationCondition := ...
116  inductive vc: Type
117  | term    : term → vc
118  | not     : vc → vc
119  | and     : vc → vc → vc
120  | or      : vc → vc → vc
```

```
121  | pre      : term → term → vc
122  | pre₁     : unop → term → vc
123  | pre₂     : binop → term → term → vc
124  | post     : term → term → vc
125  | univ     : var → vc → vc
```

A.2 `definitions1.lean`

This files includes the definition of variable substitution in terms and propositions as well as lifting of quantifiers as part of the quantifier instantiation algorithm.

```
1   -- first part of definitions
2
3   import .syntax .sizeof .eqdec
4
5   --   #######################################
6   --   ### MINOR DEFINITIONS AND NOTATIONS ###
7   --   #######################################
8
9   -- P → Q
10
11  @[reducible]
12  def spec.implies(P Q: spec): spec := spec.or (spec.not P) Q
13
14  @[reducible]
15  def prop.implies(P Q: prop): prop := prop.or (prop.not P) Q
16
17  @[reducible]
18  def propctx.implies(P Q: propctx): propctx := propctx.or (propctx.not P) Q
19
20  @[reducible]
21  def vc.implies(P Q: vc): vc := vc.or (vc.not P) Q
22
23
24
```

153

```
25  -- P ↔ Q
26
27  @[reducible]
28  def spec.iff(P Q: spec): spec := spec.and (spec.implies P Q) (spec.implies Q P)
29
30  @[reducible]
31  def prop.iff(P Q: prop): prop := prop.and (prop.implies P Q) (prop.implies Q P)
32
33  @[reducible]
34  def propctx.iff(P Q: propctx): propctx := propctx.and (propctx.implies P Q)
35                                                        (propctx.implies Q P)
36
37  @[reducible]
38  def vc.iff(P Q: vc): vc := vc.and (vc.implies P Q) (vc.implies Q P)
39
40  -- P ⋀ Q
41  class has_and (α : Type) := (and : α → α → α)
42  instance : has_and spec := ⟨spec.and⟩
43  instance : has_and prop := ⟨prop.and⟩
44  instance : has_and propctx := ⟨propctx.and⟩
45  instance : has_and vc := ⟨vc.and⟩
46  infixr `⋀`:35 := has_and.and
47
48  -- P ⋁ Q
49
50  class has_or (α : Type) := (or : α → α → α)
51  instance : has_or spec := ⟨spec.or⟩
52  instance : has_or prop := ⟨prop.or⟩
53  instance : has_or propctx := ⟨propctx.or⟩
54  instance : has_or vc := ⟨vc.or⟩
55  infixr `⋁`:30 := has_or.or
56
57  -- use ∘ as hole
58  notation ` ∘ ` := termctx.hole
59
60  -- simple coercions
61  instance value_to_term : has_coe value term := ⟨term.value⟩
62  instance var_to_term : has_coe var term := ⟨term.var⟩
```

154

```
63  instance term_to_prop : has_coe term prop := ⟨prop.term⟩
64  instance termctx_to_propctx : has_coe termctx propctx := ⟨propctx.term⟩
65  instance term_to_vc : has_coe term vc := ⟨vc.term⟩
66
67  -- use (t ≡ t)/(t ≣ t) to construct equality comparison
68  infix ≡ := term.binop binop.eq
69  infix ` ≣ `:50 := termctx.binop binop.eq
70
71  -- syntax for let expressions
72  notation `lett` x `=`:1 `true` `in` e := exp.true x e
73  notation `letf` x `=`:1 `false` `in` e := exp.false x e
74  notation `letn` x `=`:1 n`in` e := exp.num x n e
75  notation `letf` f `[`:1 x `]` `req` R `ens` S `{`:1 e `}`:1 `in` e'
76          := exp.func f x R S e e'
77  notation `letop` y `=`:1 op `[`:1 x `]`:1 `in` e := exp.unop y op x e
78  notation `letop2` z `=`:1 op `[`:1 x `,` y `]`:1 `in` e := exp.binop z op x y e
79  notation `letapp` y `=`:1 f `[`:1 x `]`:1 `in` e := exp.app y f x e
80
81  -- σ[x ↦ v]
82  notation e `[` x `↦` v `]` := env.cons e x v
83
84  -- (σ, e) : stack
85  instance : has_coe (env × exp) stack := ⟨λe, stack.top e.1 e.2⟩
86
87  -- κ · [σ, let y = f [ x ] in e]
88  notation st `·` `[` env `,` `letapp` y `=`:1 f `[` x `]` `in` e `]`
89          := stack.cons st env y f x e
90
91  -- env lookup as function application
92  def env.apply: env → var → option value
93  | env.empty _ := none
94  | (σ[x ↦ v]) y :=
95    have σ.sizeof < (σ[x ↦ v]).sizeof, from sizeof_env_rest,
96    if x = y ∧ option.is_none (σ.apply y) then v else σ.apply y
97  using_well_founded {
98    rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ e, e.1.sizeof)⟩],
99    dec_tac := tactic.assumption
100 }
```

```
101  instance : has_coe_to_fun env := ⟨λ _, var → option value, env.apply⟩

102

103  def env.rest: env → env
104  | env.empty := env.empty
105  | (σ[x ↦ v])  := σ

106

107  -- x ∈ σ

108

109  inductive env.contains: env → var → Prop
110  | same {e: env} {x: var} {v: value} : env.contains (e[x ↦ v]) x
111  | rest {e: env} {x y: var} {v: value} : env.contains e x → env.contains (e[y ↦ v]) x

112

113  instance : has_mem var env := ⟨λx σ, σ.contains x⟩

114

115  -- dom( σ )

116

117  def env.dom (σ: env): set var := λx, x ∈ σ

118

119  -- spec to prop coercion

120

121  @[reducible]
122  def prop.func (f: term) (x: var) (P: prop) (Q: prop): prop :=
123    term.unop unop.isFunc f ∧
124    prop.forallc x (prop.implies P (prop.pre f x) ∧
125                    prop.implies (prop.post f x) Q)

126

127  def spec.to_prop : spec → prop
128  | (spec.term t)      := prop.term t
129  | (spec.not S)       :=
130      have S.sizeof < S.not.sizeof, from sizeof_spec_not,
131      prop.not S.to_prop
132  | (spec.and R S)     :=
133      have R.sizeof < (R ∧ S).sizeof, from sizeof_spec_and₁,
134      have S.sizeof < (R ∧ S).sizeof, from sizeof_spec_and₂,
135      R.to_prop ∧ S.to_prop
136  | (spec.or R S)      :=
137      have R.sizeof < (R ∨ S).sizeof, from sizeof_spec_or₁,
138      have S.sizeof < (R ∨ S).sizeof, from sizeof_spec_or₂,
```

156

```
139        R.to_prop ⋁ S.to_prop
140  | (spec.func f x R S) :=
141      have R.sizeof < (spec.func f x R S).sizeof, from sizeof_spec_func_R,
142      have S.sizeof < (spec.func f x R S).sizeof, from sizeof_spec_func_S,
143      prop.func f x R.to_prop S.to_prop
144
145  using_well_founded {
146    rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ e, e.sizeof)⟩],
147    dec_tac := tactic.assumption
148  }
149
150  instance spec_to_prop : has_coe spec prop := ⟨spec.to_prop⟩
151
152  -- term to termctx coercion
153
154  def term.to_termctx : term → termctx
155  | (term.value v)       := termctx.value v
156  | (term.var x)         := termctx.var x
157  | (term.unop op t)     := termctx.unop op t.to_termctx
158  | (term.binop op t₁ t₂) := termctx.binop op t₁.to_termctx t₂.to_termctx
159  | (term.app t₁ t₂)      := termctx.app t₁.to_termctx t₂.to_termctx
160
161  instance term_to_termctx : has_coe term termctx := ⟨term.to_termctx⟩
162
163  -- term to termctx coercion
164
165  def prop.to_propctx : prop → propctx
166  | (prop.term t)        := propctx.term t
167  | (prop.not P)         := propctx.not P.to_propctx
168  | (prop.and P₁ P₂)      := P₁.to_propctx ⋀ P₂.to_propctx
169  | (prop.or P₁ P₂)       := P₁.to_propctx ⋁ P₂.to_propctx
170  | (prop.pre t₁ t₂)      := propctx.pre t₁ t₂
171  | (prop.pre₁ op t)      := propctx.pre₁ op t
172  | (prop.pre₂ op t₁ t₂)  := propctx.pre₂ op t₁ t₂
173  | (prop.post t₁ t₂)     := propctx.post t₁ t₂
174  | (prop.call t)        := propctx.call t
175  | (prop.forallc x P)   := propctx.forallc x P.to_propctx
176  | (prop.exis x P)      := propctx.exis x P.to_propctx
```

```
177
178  instance prop_to_propctx : has_coe prop propctx := ⟨prop.to_propctx⟩
179
180  -- termctx substituttion as function application
181
182  def termctx.apply: termctx → term → term
183  | ∘                      t := t
184  | (termctx.value v)      _ := term.value v
185  | (termctx.var x)        _ := term.var x
186  | (termctx.unop op t₁)   t := term.unop op (t₁.apply t)
187  | (termctx.binop op t₁ t₂) t := term.binop op (t₁.apply t) (t₂.apply t)
188  | (termctx.app t₁ t₂)    t := term.app (t₁.apply t) (t₂.apply t)
189
190  instance : has_coe_to_fun termctx := ⟨λ _, term → term, termctx.apply⟩
191
192  -- propctx substituttion as function application
193
194  def propctx.apply: propctx → term → prop
195  | (propctx.term t₁) t       := t₁ t
196  | (propctx.not P) t         := prop.not (P.apply t)
197  | (propctx.and P₁ P₂) t     := P₁.apply t ∧ P₂.apply t
198  | (propctx.or P₁ P₂) t      := P₁.apply t ∨ P₂.apply t
199  | (propctx.pre t₁ t₂) t     := prop.pre (t₁ t) (t₂ t)
200  | (propctx.pre₁ op t₁) t    := prop.pre₁ op (t₁ t)
201  | (propctx.pre₂ op t₁ t₂) t := prop.pre₂ op (t₁ t) (t₂ t)
202  | (propctx.post t₁ t₂) t    := prop.post (t₁ t) (t₂ t)
203  | (propctx.call t₁) t       := prop.call (t₁ t)
204  | (propctx.forallc x P) t   := prop.forallc x (P.apply t)
205  | (propctx.exis x P) t      := prop.exis x (P.apply t)
206
207  instance : has_coe_to_fun propctx := ⟨λ _, term → prop, propctx.apply⟩
208
209  --  ##############################
210  --  ### VARIABLE SUBSTITUTION ###
211  --  ##############################
212
213  -- free variables in terms, propositions and vcs
214
```

158

```
215  inductive free_in_term (x: var) : term → Prop
216  | var                                : free_in_term x
217  | unop {t: term} {op: unop}          : free_in_term t
218                                       → free_in_term (term.unop op t)
219  | binop₁ {t₁ t₂: term} {op: binop} : free_in_term t₁
220                                       → free_in_term (term.binop op t₁ t₂)
221  | binop₂ {t₁ t₂: term} {op: binop} : free_in_term t₂
222                                       → free_in_term (term.binop op t₁ t₂)
223  | app₁ {t₁ t₂: term}               : free_in_term t₁
224                                       → free_in_term (term.app t₁ t₂)
225  | app₂ {t₁ t₂: term}               : free_in_term t₂
226                                       → free_in_term (term.app t₁ t₂)
227
228  inductive free_in_prop (x: var) : prop → Prop
229  | term {t: term}                     : free_in_term x t
230                                       → free_in_prop t
231  | not {p: prop}                      : free_in_prop p
232                                       → free_in_prop (prop.not p)
233  | and₁ {p₁ p₂: prop}               : free_in_prop p₁
234                                       → free_in_prop (prop.and p₁ p₂)
235  | and₂ {p₁ p₂: prop}               : free_in_prop p₂
236                                       → free_in_prop (prop.and p₁ p₂)
237  | or₁ {p₁ p₂: prop}                : free_in_prop p₁
238                                       → free_in_prop (prop.or p₁ p₂)
239  | or₂ {p₁ p₂: prop}                : free_in_prop p₂
240                                       → free_in_prop (prop.or p₁ p₂)
241  | pre₁ {t₁ t₂: term}               : free_in_term x t₁
242                                       → free_in_prop (prop.pre t₁ t₂)
243  | pre₂ {t₁ t₂: term}               : free_in_term x t₂
244                                       → free_in_prop (prop.pre t₁ t₂)
245  | preop {t: term} {op: unop}         : free_in_term x t
246                                       → free_in_prop (prop.pre₁ op t)
247  | preop₁ {t₁ t₂: term} {op: binop} : free_in_term x t₁
248                                       → free_in_prop (prop.pre₂ op t₁ t₂)
249  | preop₂ {t₁ t₂: term} {op: binop} : free_in_term x t₂
250                                       → free_in_prop (prop.pre₂ op t₁ t₂)
251  | post₁ {t₁ t₂: term}              : free_in_term x t₁
252                                       → free_in_prop (prop.post t₁ t₂)
```

159

```
253 | post₂ {t₁ t₂: term}                  : free_in_term x t₂
254                                        → free_in_prop (prop.post t₁ t₂)
255 | call {t: term}                       : free_in_term x t
256                                        → free_in_prop (prop.call t)
257 | forallc {y: var} {p: prop}          : (x ≠ y) → free_in_prop p
258                                        → free_in_prop (prop.forallc y p)
259 | exis {y: var} {p: prop}             : (x ≠ y) → free_in_prop p
260                                        → free_in_prop (prop.exis y p)
261
262 inductive free_in_vc (x: var) : vc → Prop
263 | term {t: term}                          : free_in_term x t
264                                            → free_in_vc t
265 | not {P: vc}                             : free_in_vc P
266                                            → free_in_vc (vc.not P)
267 | and₁ {P₁ P₂: vc}                        : free_in_vc P₁
268                                            → free_in_vc (vc.and P₁ P₂)
269 | and₂ {P₁ P₂: vc}                        : free_in_vc P₂
270                                            → free_in_vc (vc.and P₁ P₂)
271 | or₁ {P₁ P₂: vc}                         : free_in_vc P₁
272                                            → free_in_vc (vc.or P₁ P₂)
273 | or₂ {P₁ P₂: vc}                         : free_in_vc P₂
274                                            → free_in_vc (vc.or P₁ P₂)
275 | pre₁ {t₁ t₂: term}                      : free_in_term x t₁
276                                            → free_in_vc (vc.pre t₁ t₂)
277 | pre₂ {t₁ t₂: term}                      : free_in_term x t₂
278                                            → free_in_vc (vc.pre t₁ t₂)
279 | preop {t: term} {op: unop}             : free_in_term x t
280                                            → free_in_vc (vc.pre₁ op t)
281 | preop₁ {t₁ t₂: term} {op: binop}       : free_in_term x t₁
282                                            → free_in_vc (vc.pre₂ op t₁ t₂)
283 | preop₂ {t₁ t₂: term} {op: binop}       : free_in_term x t₂
284                                            → free_in_vc (vc.pre₂ op t₁ t₂)
285 | post₁ {t₁ t₂: term}                     : free_in_term x t₁
286                                            → free_in_vc (vc.post t₁ t₂)
287 | post₂ {t₁ t₂: term}                     : free_in_term x t₂
288                                            → free_in_vc (vc.post t₁ t₂)
289 | univ {y: var} {P: vc}                  : (x ≠ y) → free_in_vc P
290                                            → free_in_vc (vc.univ y P)
```

160

```
291
292  -- notation x ∈ FV t/P
293
294  inductive freevars
295  | term: term → freevars
296  | prop: prop → freevars
297  | vc:   vc   → freevars
298
299  class has_fv (α: Type) := (fv : α → freevars)
300  instance : has_fv term := ⟨freevars.term⟩
301  instance : has_fv prop := ⟨freevars.prop⟩
302  instance : has_fv vc   := ⟨freevars.vc⟩
303
304  def freevars.to_set: freevars → set var
305  | (freevars.term t) := λx, free_in_term x t
306  | (freevars.prop P) := λx, free_in_prop x P
307  | (freevars.vc P)   := λx, free_in_vc x P
308
309  @[reducible]
310  def FV {α: Type} [h: has_fv α] (a: α): set var := (has_fv.fv a).to_set
311
312  @[reducible]
313  def closed {α: Type} [h: has_fv α] (a: α): Prop := ∀x, x ∉ FV a
314
315  -- fresh variables (not used in the provided term/prop)
316
317  def term.fresh_var : term → var
318  | (term.value v)       := 0
319  | (term.var x)         := x + 1
320  | (term.unop op t)     := t.fresh_var
321  | (term.binop op t₁ t₂) := max t₁.fresh_var t₂.fresh_var
322  | (term.app t₁ t₂)      := max t₁.fresh_var t₂.fresh_var
323
324  def prop.fresh_var : prop → var
325  | (prop.term t)        := t.fresh_var
326  | (prop.not P)         := P.fresh_var
327  | (prop.and P₁ P₂)      := max P₁.fresh_var P₂.fresh_var
328  | (prop.or P₁ P₂)       := max P₁.fresh_var P₂.fresh_var
```

```
329  | (prop.pre t₁ t₂)      := max t₁.fresh_var t₂.fresh_var
330  | (prop.pre₁ op t)      := t.fresh_var
331  | (prop.pre₂ op t₁ t₂)  := max t₁.fresh_var t₂.fresh_var
332  | (prop.post t₁ t₂)     := max t₁.fresh_var t₂.fresh_var
333  | (prop.call t)         := t.fresh_var
334  | (prop.forallc x P)    := max (x + 1) P.fresh_var
335  | (prop.exis x P)       := max (x + 1) P.fresh_var
336
337  -- substituation in terms, propositions and vcs
338
339  def term.subst (x: var) (v: value): term → term
340  | (term.value v')       := v'
341  | (term.var y)          := if x = y then v else y
342  | (term.unop op t)      := term.unop op t.subst
343  | (term.binop op t₁ t₂) := term.binop op t₁.subst t₂.subst
344  | (term.app t₁ t₂)      := term.app t₁.subst t₂.subst
345
346  def term.subst_env: env → term → term
347  | env.empty t := t
348  | (σ[x ↦ v]) t :=
349      have σ.sizeof < (σ[x ↦ v]).sizeof, from sizeof_env_rest,
350      term.subst x v (term.subst_env σ t)
351
352  using_well_founded {
353    rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ e, e.1.sizeof)⟩],
354    dec_tac := tactic.assumption
355  }
356
357  def prop.subst (x: var) (v: value): prop → prop
358  | (prop.term t)        := term.subst x v t
359  | (prop.not P)         := P.subst.not
360  | (prop.and P Q)       := P.subst ∧ Q.subst
361  | (prop.or P Q)        := P.subst ∨ Q.subst
362  | (prop.pre t₁ t₂)     := prop.pre (term.subst x v t₁) (term.subst x v t₂)
363  | (prop.pre₁ op t)     := prop.pre₁ op (term.subst x v t)
364  | (prop.pre₂ op t₁ t₂) := prop.pre₂ op (term.subst x v t₁) (term.subst x v t₂)
365  | (prop.call t)        := prop.call (term.subst x v t)
366  | (prop.post t₁ t₂)    := prop.post (term.subst x v t₁) (term.subst x v t₂)
```

162

```
367 | (prop.forallc y P)   := prop.forallc y (if x = y then P else P.subst)
368 | (prop.exis y P)       := prop.exis y (if x = y then P else P.subst)
369
370 def prop.subst_env: env → prop → prop
371 | env.empty P := P
372 | (σ[x ↦ v]) P :=
373     have σ.sizeof < (σ[x ↦ v]).sizeof, from sizeof_env_rest,
374     prop.subst x v (prop.subst_env σ P)
375
376 using_well_founded {
377   rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ e, e.1.sizeof)⟩],
378   dec_tac := tactic.assumption
379 }
380
381 def vc.subst (x: var) (v: value): vc → vc
382 | (vc.term t)        := term.subst x v t
383 | (vc.not P)         := vc.not P.subst
384 | (vc.and P Q)       := P.subst ⋀ Q.subst
385 | (vc.or P Q)        := P.subst ⋁ Q.subst
386 | (vc.pre t₁ t₂)     := vc.pre (term.subst x v t₁) (term.subst x v t₂)
387 | (vc.pre₁ op t)     := vc.pre₁ op (term.subst x v t)
388 | (vc.pre₂ op t₁ t₂) := vc.pre₂ op (term.subst x v t₁) (term.subst x v t₂)
389 | (vc.post t₁ t₂)    := vc.post (term.subst x v t₁) (term.subst x v t₂)
390 | (vc.univ y P)      := vc.univ y (if x = y then P else P.subst)
391
392 def vc.subst_env: env → vc → vc
393 | env.empty P := P
394 | (σ[x ↦ v]) P :=
395     have σ.sizeof < (σ[x ↦ v]).sizeof, from sizeof_env_rest,
396     vc.subst x v (vc.subst_env σ P)
397
398 using_well_founded {
399   rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ e, e.1.sizeof)⟩],
400   dec_tac := tactic.assumption
401 }
402
403
404
```

163

```
405  -- σ \ { x }
406  def env.without: env → var → env
407  | env.empty y := env.empty
408  | (σ[x ↦ v]) y := have σ.sizeof < (σ[x ↦ v]).sizeof, from sizeof_env_rest,
409                  if x = y then env.without σ y else ((env.without σ y)[x ↦ v])
410
411  using_well_founded {
412    rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ e, e.1.sizeof)⟩],
413    dec_tac := tactic.assumption
414  }
415
416  -- closed under substitution
417
418  @[reducible]
419  def closed_subst {α: Type} [h: has_fv α] (σ: env) (a: α): Prop := FV a ⊆ σ.dom
420
421  -- subst term
422
423  def term.substt (x: var) (t: term): term → term
424  | (term.value v')      := v'
425  | (term.var y)         := if x = y then t else y
426  | (term.unop op t₁)    := term.unop op t₁.substt
427  | (term.binop op t₁ t₂) := term.binop op t₁.substt t₂.substt
428  | (term.app t₁ t₂)     := term.app t₁.substt t₂.substt
429
430  def prop.substt (x: var) (t: term): prop → prop
431  | (prop.term t₁)       := term.substt x t t₁
432  | (prop.not P)         := P.substt.not
433  | (prop.and P Q)       := P.substt ∧ Q.substt
434  | (prop.or P Q)        := P.substt ∨ Q.substt
435  | (prop.pre t₁ t₂)     := prop.pre (term.substt x t t₁) (term.substt x t t₂)
436  | (prop.pre₁ op t₁)    := prop.pre₁ op (term.substt x t t₁)
437  | (prop.pre₂ op t₁ t₂) := prop.pre₂ op (term.substt x t t₁) (term.substt x t t₂)
438  | (prop.call t₁)       := prop.call (term.substt x t t₁)
439  | (prop.post t₁ t₂)    := prop.post (term.substt x t t₁) (term.substt x t t₂)
440  | (prop.forallc y P)   := prop.forallc y (if x = y then P else P.substt)
441  | (prop.exis y P)      := prop.exis y (if x = y then P else P.substt)
442
```

164

```
443  def vc.substt (x: var) (t: term): vc → vc
444  | (vc.term t₁)        := term.substt x t t₁
445  | (vc.not P)          := P.substt.not
446  | (vc.and P Q)        := P.substt ∧ Q.substt
447  | (vc.or P Q)         := P.substt ∨ Q.substt
448  | (vc.pre t₁ t₂)      := vc.pre (term.substt x t t₁) (term.substt x t t₂)
449  | (vc.pre₁ op t₁)     := vc.pre₁ op (term.substt x t t₁)
450  | (vc.pre₂ op t₁ t₂)  := vc.pre₂ op (term.substt x t t₁) (term.substt x t t₂)
451  | (vc.post t₁ t₂)     := vc.post (term.substt x t t₁) (term.substt x t t₂)
452  | (vc.univ y P)       := vc.univ y (if x = y then P else P.substt)
453
454  --   ################################
455  --   ### QUANTIFIER INSTANTIATION ###
456  --   ################################
457
458  -- simple conversion of propositions to verification conditions
459  -- (no quantifier instantiation)
460
461  def prop.to_vc: prop → vc
462  | (prop.term t)        := vc.term t
463  | (prop.not P)         := vc.not P.to_vc
464  | (prop.and P₁ P₂)     := P₁.to_vc ∧ P₂.to_vc
465  | (prop.or P₁ P₂)      := P₁.to_vc ∨ P₂.to_vc
466  | (prop.pre t₁ t₂)     := vc.pre t₁ t₂
467  | (prop.pre₁ op t)     := vc.pre₁ op t
468  | (prop.pre₂ op t₁ t₂) := vc.pre₂ op t₁ t₂
469  | (prop.post t₁ t₂)    := vc.post t₁ t₂
470  | (prop.call _)        := vc.term value.true
471  | (prop.forallc x P)   := vc.univ x P.to_vc
472  | (prop.exis x P)      := have P.sizeof < (prop.exis x P).sizeof,
473                             from sizeof_prop_exis,
474                             vc.not (vc.univ x (vc.not P.to_vc))
475
476  -- lift_p(P) / lift_n(P) lifts quantifiers in either positive or negative position
477  -- to become a top-level quantifier by using a fresh (unbound) variable
478
479  mutual def prop.lift_p, prop.lift_n
480
```

165

```
481  with prop.lift_p: prop → var → option prop
482  | (prop.term t) y        := none
483  | (prop.not P) y         := have P.sizeof < P.not.sizeof,
484                              from sizeof_prop_not,
485                              prop.not <$> P.lift_n y
486  | (prop.and P₁ P₂) y      := have P₁.sizeof < (P₁ ∧ P₂).sizeof,
487                              from sizeof_prop_and₁,
488                              have P₂.sizeof < (P₁ ∧ P₂).sizeof,
489                              from sizeof_prop_and₂,
490                              match P₁.lift_p y with
491                              | some P₁' := some (P₁' ∧ P₂)
492                              | none := (λP₂', P₁ ∧ P₂') <$> P₂.lift_p y
493                              end
494  | (prop.or P₁ P₂) y       := have P₁.sizeof < (P₁ ∨ P₂).sizeof,
495                              from sizeof_prop_or₁,
496                              have P₂.sizeof < (P₁ ∨ P₂).sizeof,
497                              from sizeof_prop_or₂,
498                              match P₁.lift_p y with
499                              | some P₁' := some (P₁' ∨ P₂)
500                              | none := (λP₂', P₁ ∨ P₂') <$> P₂.lift_p y
501                              end
502  | (prop.pre t₁ t₂) y     := none
503  | (prop.pre₁ op t) y     := none
504  | (prop.pre₂ op t₁ t₂) y := none
505  | (prop.post t₁ t₂) y    := none
506  | (prop.call t) y        := none
507  | (prop.forallc x P) y   := some (prop.implies (prop.call y) (P.substt x y))
508  | (prop.exis x P) y      := none
509
510  with prop.lift_n: prop → var → option prop
511  | (prop.term t) y        := none
512  | (prop.not P) y         := have P.sizeof < P.not.sizeof,
513                              from sizeof_prop_not,
514                              prop.not <$> P.lift_p y
515  | (prop.and P₁ P₂) y      := have P₁.sizeof < (P₁ ∧ P₂).sizeof,
516                              from sizeof_prop_and₁,
517                              have P₂.sizeof < (P₁ ∧ P₂).sizeof,
518                              from sizeof_prop_and₂,
```

166

```
519                               match P₁.lift_n y with
520                               | some P₁' := some (P₁' ∧ P₂)
521                               | none := (λP₂', P₁ ∧ P₂') <$> P₂.lift_n y
522                               end
523  | (prop.or P₁ P₂) y        := have P₁.sizeof < (P₁ ∨ P₂).sizeof,
524                               from sizeof_prop_or₁,
525                               have P₂.sizeof < (P₁ ∨ P₂).sizeof,
526                               from sizeof_prop_or₂,
527                               match P₁.lift_n y with
528                               | some P₁' := some (P₁' ∨ P₂)
529                               | none := (λP₂', P₁ ∨ P₂') <$> P₂.lift_n y
530                               end
531  | (prop.pre t₁ t₂) y       := none
532  | (prop.pre₁ op t) y       := none
533  | (prop.pre₂ op t₁ t₂) y   := none
534  | (prop.post t₁ t₂) y      := none
535  | (prop.call t) y          := none
536  | (prop.forallc x P) y     := none
537  | (prop.exis x P) y        := none
538
539  using_well_founded {
540    rel_tac := λ _ _, `[exact ⟨_, measure_wf $ λ s,
541    match s with
542    | psum.inl a := a.1.sizeof
543    | psum.inr a := a.1.sizeof
544    end⟩],
545    dec_tac := tactic.assumption
546  }
547
548  -- remaining definitions need some additional lemmas in qiaux.lean to prove
549  -- termination definitions continue in definitions2.lean
```

A.3 `definitions2.lean`

This file contains the remainder of the quantifier instantiation algorithm. Additionally, this files defines the evaluation relation for program expressions, an axiomatization of SMT Logic and the verification relation.

```
1   -- second part of definitions
2
3   import .definitions1 .qiaux
4
5   --   ################################
6   --   ### QUANTIFIER INSTANTIATION ###
7   --   ################################
8
9   -- first part is in definitions1.lean
10  -- the following definitions need some additional lemmas from qiaux.lean to prove
11  -- termination
12
13  -- lift_all(P) performs repeated lifting of quantifiers in positive
14  -- positions until there is no more quantifier to be lifted
15  def prop.lift_all: prop → prop
16  | P :=
17    let r := P.lift_p P.fresh_var in
18    let z := r in
19    have h: z = r, from rfl,
20    @option.cases_on prop (λr, (z = r) → prop) r (
21      assume : z = none,
22      P
23    ) (
24      assume P': prop,
25      assume : z = (some P'),
26      have r_id: r = (some P'), from eq.trans h this,
27      have P'.num_quantifiers < P.num_quantifiers,
28      from (lifted_prop_smaller P').left r_id,
29      prop.lift_all P'
```

```
30      ) rfl
31
32    using_well_founded {
33      rel_tac := λ _ _, `[exact ⟨_, measure_wf $ λ s, s.num_quantifiers ⟩],
34      dec_tac := tactic.assumption
35    }
36
37    -- erase_p(P) / erase_n(P) replaces all triggers and quantifiers
38    -- in either positive or negative position with 'true'
39    mutual def prop.erased_p, prop.erased_n
40
41    with prop.erased_p: prop → vc
42    | (prop.term t)        := vc.term t
43    | (prop.not P)         := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
44                              vc.not P.erased_n
45    | (prop.and P₁ P₂)     := have P₁.sizeof < (P₁ ∧ P₂).sizeof, from sizeof_prop_and₁,
46                              have P₂.sizeof < (P₁ ∧ P₂).sizeof, from sizeof_prop_and₂,
47                              P₁.erased_p ∧ P₂.erased_p
48    | (prop.or P₁ P₂)      := have P₁.sizeof < (P₁ ∨ P₂).sizeof, from sizeof_prop_or₁,
49                              have P₂.sizeof < (P₁ ∨ P₂).sizeof, from sizeof_prop_or₂,
50                              P₁.erased_p ∨ P₂.erased_p
51    | (prop.pre t₁ t₂)     := vc.pre t₁ t₂
52    | (prop.pre₁ op t)     := vc.pre₁ op t
53    | (prop.pre₂ op t₁ t₂) := vc.pre₂ op t₁ t₂
54    | (prop.post t₁ t₂)    := vc.post t₁ t₂
55    | (prop.call _)        := vc.term value.true
56    | (prop.forallc x P)   := vc.term value.true
57    | (prop.exis x P)      := have P.sizeof < (prop.exis x P).sizeof,
58                              from sizeof_prop_exis,
59                              vc.not (vc.univ x (vc.not P.erased_p))
60
61    with prop.erased_n: prop → vc
62    | (prop.term t)        := vc.term t
63    | (prop.not P)         := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
64                              vc.not P.erased_p
65    | (prop.and P₁ P₂)     := have P₁.sizeof < (P₁ ∧ P₂).sizeof, from sizeof_prop_and₁,
66                              have P₂.sizeof < (P₁ ∧ P₂).sizeof, from sizeof_prop_and₂,
67                              P₁.erased_n ∧ P₂.erased_n
```

169

```
68  | (prop.or P₁ P₂)       := have P₁.sizeof < (P₁ ∨ P₂).sizeof, from sizeof_prop_or₁,
69                              have P₂.sizeof < (P₁ ∨ P₂).sizeof, from sizeof_prop_or₂,
70                              P₁.erased_n ∨ P₂.erased_n
71  | (prop.pre t₁ t₂)     := vc.pre t₁ t₂
72  | (prop.pre₁ op t)     := vc.pre₁ op t
73  | (prop.pre₂ op t₁ t₂) := vc.pre₂ op t₁ t₂
74  | (prop.post t₁ t₂)    := vc.post t₁ t₂
75  | (prop.call _)        := vc.term value.true
76  | (prop.forallc x P)   := have P.sizeof < (prop.forallc x P).sizeof,
77                              from sizeof_prop_forall,
78                              vc.univ x P.erased_n
79  | (prop.exis x P)      := have P.sizeof < (prop.exis x P).sizeof,
80                              from sizeof_prop_exis,
81                              vc.not (vc.univ x (vc.not P.erased_n))
82
83  using_well_founded {
84    rel_tac := λ _ _, `[exact erased_measure],
85    dec_tac := tactic.assumption
86  }
87
88  -- given a call trigger t, inst_with_p(P, t) / inst_with_n(P, t) instantiates all
89  -- quantifiers in either positive or negative positions by adding a conjunction
90  -- where the quantified variable is replaced by the term in the given trigger
91  mutual def prop.instantiate_with_p, prop.instantiate_with_n
92
93  with prop.instantiate_with_p: prop → calltrigger → prop
94  | (prop.term t) _        := prop.term t
95  | (prop.not P) t         := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
96                              prop.not (P.instantiate_with_n t)
97  | (prop.and P₁ P₂) t     := have P₁.sizeof < (P₁ ∧ P₂).sizeof,
98                              from sizeof_prop_and₁,
99                              have P₂.sizeof < (P₁ ∧ P₂).sizeof,
100                             from sizeof_prop_and₂,
101                             P₁.instantiate_with_p t ∧ P₂.instantiate_with_p t
102 | (prop.or P₁ P₂) t      := have P₁.sizeof < (P₁ ∨ P₂).sizeof,
103                             from sizeof_prop_or₁,
104                             have P₂.sizeof < (P₁ ∨ P₂).sizeof,
105                             from sizeof_prop_or₂,
```

```
106                            P₁.instantiate_with_p t ∨ P₂.instantiate_with_p t
107  | (prop.pre t₁ t₂) _      := prop.pre t₁ t₂
108  | (prop.pre₁ op t) _      := prop.pre₁ op t
109  | (prop.pre₂ op t₁ t₂) _ := prop.pre₂ op t₁ t₂
110  | (prop.post t₁ t₂) _     := prop.post t₁ t₂
111  | (prop.call t) _         := prop.call t
112  | (prop.forallc x P) t    := prop.forallc x P ∧ P.substt x t.x -- instantiate
113  | (prop.exis x P) t       := prop.exis x P
114
115  with prop.instantiate_with_n: prop → calltrigger → prop
116  | (prop.term t) _         := prop.term t
117  | (prop.not P) t          := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
118                               prop.not (P.instantiate_with_p t)
119  | (prop.and P₁ P₂) t      := have P₁.sizeof < (P₁ ∧ P₂).sizeof,
120                                 from sizeof_prop_and₁,
121                               have P₂.sizeof < (P₁ ∧ P₂).sizeof,
122                                 from sizeof_prop_and₂,
123                               P₁.instantiate_with_n t ∧ P₂.instantiate_with_n t
124  | (prop.or P₁ P₂) t       := have P₁.sizeof < (P₁ ∨ P₂).sizeof,
125                                 from sizeof_prop_or₁,
126                               have P₂.sizeof < (P₁ ∨ P₂).sizeof,
127                                 from sizeof_prop_or₂,
128                               P₁.instantiate_with_n t ∨ P₂.instantiate_with_n t
129  | (prop.pre t₁ t₂) _      := prop.pre t₁ t₂
130  | (prop.pre₁ op t) _      := prop.pre₁ op t
131  | (prop.pre₂ op t₁ t₂) _ := prop.pre₂ op t₁ t₂
132  | (prop.post t₁ t₂) _     := prop.post t₁ t₂
133  | (prop.call t) _         := prop.call t
134  | (prop.forallc x P) t    := prop.forallc x P
135  | (prop.exis x P) t       := prop.exis x P
136
137  using_well_founded {
138    rel_tac := λ _ _, `[exact instantiate_with_measure],
139    dec_tac := tactic.assumption
140  }
141
142  -- finds all call triggers in either positive or negative positions and
143  -- returns these as list
```

```
144  mutual def prop.find_calls_p, prop.find_calls_n

145

146  with prop.find_calls_p: prop → list calltrigger
147  | (prop.term t)           := []
148  | (prop.not P)            := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
149                                 P.find_calls_n
150  | (prop.and $P_1$ $P_2$)       := have $P_1$.sizeof < ($P_1$ ∧ $P_2$).sizeof, from sizeof_prop_and$_1$,
151                                 have $P_2$.sizeof < ($P_1$ ∧ $P_2$).sizeof, from sizeof_prop_and$_2$,
152                                 $P_1$.find_calls_p ++ $P_2$.find_calls_p
153  | (prop.or $P_1$ $P_2$)        := have $P_1$.sizeof < ($P_1$ ∨ $P_2$).sizeof, from sizeof_prop_or$_1$,
154                                 have $P_2$.sizeof < ($P_1$ ∨ $P_2$).sizeof, from sizeof_prop_or$_2$,
155                                 $P_1$.find_calls_p ++ $P_2$.find_calls_p
156  | (prop.pre $t_1$ $t_2$)       := []
157  | (prop.pre$_1$ op t)        := []
158  | (prop.pre$_2$ op $t_1$ $t_2$)   := []
159  | (prop.post $t_1$ $t_2$)      := []
160  | (prop.call t)           := [ ⟨ t ⟩ ]
161  | (prop.forallc x P)      := []
162  | (prop.exis x P)         := []

163

164  with prop.find_calls_n: prop → list calltrigger
165  | (prop.term t)           := []
166  | (prop.not P)            := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
167                                 P.find_calls_p
168  | (prop.and $P_1$ $P_2$)       := have $P_1$.sizeof < ($P_1$ ∧ $P_2$).sizeof, from sizeof_prop_and$_1$,
169                                 have $P_2$.sizeof < ($P_1$ ∧ $P_2$).sizeof, from sizeof_prop_and$_2$,
170                                 $P_1$.find_calls_n ++ $P_2$.find_calls_n
171  | (prop.or $P_1$ $P_2$)        := have $P_1$.sizeof < ($P_1$ ∨ $P_2$).sizeof, from sizeof_prop_or$_1$,
172                                 have $P_2$.sizeof < ($P_1$ ∨ $P_2$).sizeof, from sizeof_prop_or$_2$,
173                                 $P_1$.find_calls_n ++ $P_2$.find_calls_n
174  | (prop.pre $t_1$ $t_2$)       := []
175  | (prop.pre$_1$ op t)        := []
176  | (prop.pre$_2$ op $t_1$ $t_2$)   := []
177  | (prop.post $t_1$ $t_2$)      := []
178  | (prop.call t)           := []
179  | (prop.forallc x P)      := []
180  | (prop.exis x P)         := []

181
```

```
182  using_well_founded {
183    rel_tac := λ _ _, `[exact find_calls_measure],
184    dec_tac := tactic.assumption
185  }
186
187  -- performs one full instantiation for each of the triggers in the provided list
188  def prop.instantiate_with_all: prop → list calltrigger → prop
189  | P list.nil        := P
190  | P (list.cons t r) := (P.instantiate_with_n t).instantiate_with_all r
191  using_well_founded {
192    rel_tac := λ _ _, `[exact ⟨_, measure_wf $ λ s, s.2.sizeof⟩]
193  }
194
195  -- performs n rounds of instantiations. each round also involves a repeated lifting.
196  -- once all rounds are complete, remaining quantifiers and triggers in
197  -- negative positions will be erased
198  def prop.instantiate_rep: prop → ℕ → vc
199  | P 0            := P.lift_all.erased_n
200  | P (nat.succ n) := have n < n + 1, from lt_of_add_one,
201                      (P.lift_all.instantiate_with_all P.lift_all.find_calls_n)
202                      .instantiate_rep n
203
204  using_well_founded {
205    rel_tac := λ _ _, `[exact ⟨_, measure_wf $ λ s, s.2⟩]
206  }
207
208  -- finds the maximum quantifier nesting level of a given proposition
209  def prop.max_nesting_level: prop → ℕ
210  | (prop.term t)     := 0
211  | (prop.not P)      := have P.sizeof < P.not.sizeof, from sizeof_prop_not,
212                         P.max_nesting_level
213  | (prop.and P₁ P₂)  := have P₁.sizeof < (P₁ ∧ P₂).sizeof, from sizeof_prop_and₁,
214                         have P₂.sizeof < (P₁ ∧ P₂).sizeof, from sizeof_prop_and₂,
215                         max P₁.max_nesting_level P₂.max_nesting_level
216  | (prop.or P₁ P₂)   := have P₁.sizeof < (P₁ ∨ P₂).sizeof, from sizeof_prop_or₁,
217                         have P₂.sizeof < (P₁ ∨ P₂).sizeof, from sizeof_prop_or₂,
218                         max P₁.max_nesting_level P₂.max_nesting_level
219  | (prop.pre t₁ t₂)  := 0
```

173

```
220 | (prop.pre₁ op t)      := 0
221 | (prop.pre₂ op t₁ t₂) := 0
222 | (prop.post t₁ t₂)    := 0
223 | (prop.call t)        := 0
224 | (prop.forallc x P)   := have P.sizeof < (prop.forallc x P).sizeof,
225                            from sizeof_prop_forall,
226                            nat.succ P.max_nesting_level
227 | (prop.exis x P)      := 0
228
229 using_well_founded {
230   rel_tac := λ _ _, `[exact ⟨_, measure_wf $ λ s, s.sizeof⟩],
231   dec_tac := tactic.assumption
232 }
233
234 -- the main instantiation algorithm performs n rounds of instantiations
235 -- where n is the maximum quantifier nesting level and
236 -- returns the erased proposition
237 def prop.instantiated_n (P: prop): vc := P.instantiate_rep P.max_nesting_level
238
239 --  ##############################
240 --  ### OPERATIONAL SEMANTICS ###
241 --  ##############################
242
243 -- semantics of unary operators
244 def unop.apply: unop → value → option value
245 | unop.not value.true               := value.false
246 | unop.not value.false              := value.true
247 | unop.isInt (value.num _)          := value.true
248 | unop.isInt _                      := value.false
249 | unop.isBool value.true            := value.true
250 | unop.isBool value.false           := value.true
251 | unop.isBool _                     := value.false
252 | unop.isFunc (value.func _ _ _ _ _ _) := value.true
253 | unop.isFunc _                     := value.false
254 | _ _                              := none
255
256
257
```

174

```
258  -- semantics of binary operators
259  def binop.apply: binop → value → value → option value
260  | binop.plus (value.num n₁) (value.num n₂)   := value.num (n₁ + n₂)
261  | binop.minus (value.num n₁) (value.num n₂)  := value.num (n₁ - n₂)
262  | binop.times (value.num n₁) (value.num n₂)  := value.num (n₁ * n₂)
263  | binop.div (value.num n₁) (value.num n₂)    := value.num (n₁ / n₂)
264  | binop.and value.true value.true            := value.true
265  | binop.and value.true value.false           := value.false
266  | binop.and value.false value.true           := value.false
267  | binop.and value.false value.false          := value.false
268  | binop.or value.true value.true             := value.true
269  | binop.or value.true value.false            := value.true
270  | binop.or value.false value.true            := value.true
271  | binop.or value.false value.false           := value.false
272  | binop.eq v₁ v₂                             := if v₁ = v₂ then value.true
273                                                             else value.false
274  | binop.lt (value.num n₁) (value.num n₂)     := if n₁ < n₂ then value.true
275                                                             else value.false
276  | _ _ _                                      := none
277
278  -- small-step stack-based semantics
279  inductive step : stack → stack → Prop
280  notation s₁ ⟶ s₂:100 := step s₁ s₂
281
282  | ctx {s s': stack} {σ: env} {y f x: var} {e: exp}:
283      (s⟶s') →
284      (s · [σ, letapp y = f[x] in e]⟶(s' · [σ, letapp y = f[x] in e]))
285
286  | tru {σ: env} {x: var} {e: exp}:
287      (σ, lett x = true in e)⟶(σ[x ↦ value.true], e)
288
289  | fals {σ: env} {x: var} {e: exp}:
290      (σ, letf x = false in e)⟶(σ[x ↦ value.false], e)
291
292  | num {σ: env} {x: var} {e: exp} {n: ℤ}:
293      (σ, letn x = n in e)⟶(σ[x ↦ value.num n], e)
294
295
```

175

```
296 | closure {σ: env} {R' R S: spec} {f x: var} {e₁ e₂: exp}:
297     (σ, letf f[x] req R ens S {e₁} in e₂)⟶(σ[f↦value.func f x R S e₁ σ], e₂)
298
299 | unop {op: unop} {σ: env} {x y: var} {e: exp} {v₁ v: value}:
300     (σ x = v₁) →
301     (unop.apply op v₁ = v) →
302     ((σ, letop y = op [x] in e)⟶(σ[y↦v], e))
303
304 | binop {op: binop} {σ: env} {x y z: var} {e: exp} {v₁ v₂ v: value}:
305     (σ x = v₁) →
306     (σ y = v₂) →
307     (binop.apply op v₁ v₂ = v) →
308     ((σ, letop2 z = op [x, y] in e)⟶(σ[z↦v], e))
309
310 | app {σ σ': env} {R S: spec} {f g x y z: var} {e e': exp} {v: value}:
311     (σ f = value.func g z R S e σ') →
312     (σ x = v) →
313     ((σ, letapp y = f[x] in e')⟶
314         ((σ'[g↦value.func g z R S e σ'][z↦v], e) · [σ, letapp y = f[x] in e']))
315
316 | return {σ₁ σ₂ σ₃: env} {f g gx x y z: var} {R S: spec} {e e': exp} {v vₓ: value}:
317     (σ₁ z = v) →
318     (σ₂ f = value.func g gx R S e σ₃) →
319     (σ₂ x = vₓ) →
320     ((σ₁, exp.return z) · [σ₂, letapp y = f[x] in e']⟶(σ₂[y↦v], e'))
321
322 | ite_true {σ: env} {e₁ e₂: exp} {x: var}:
323     (σ x = value.true) →
324     ((σ, exp.ite x e₁ e₂)⟶(σ, e₁))
325
326 | ite_false {σ: env} {e₁ e₂: exp} {x: var}:
327     (σ x = value.false) →
328     ((σ, exp.ite x e₁ e₂)⟶(σ, e₂))
329
330 notation s₁ `⟶` s₂:100 := step s₁ s₂
331
332
333
```

176

```
334  -- transitive closure
335  inductive trans_step : stack → stack → Prop
336  notation s `⟶∗ ` s':100 := trans_step s s'
337  | rfl {s: stack}           : s⟶∗ s
338  | trans {s s' s'': stack} : (s⟶∗ s') → (s'⟶s'') → (s⟶∗ s'')
339
340  notation s `⟶∗ ` s':100 := trans_step s s'
341
342  def is_value (s: stack) :=
343    ∃(σ: env) (x: var) (v: value), s = (σ, exp.return x) ∧ (σ x = v)
344
345  --   #######################################
346  --   ### VALIDTY OF LOGICAL PROPOSITIONS ###
347  --   #######################################
348
349  -- validity is axiomatized instead defined
350  -- see axioms below
351
352  constant valid : vc → Prop
353  notation `⊨` p: 20 := valid p
354  notation σ `⊨` p: 20 := ⊨ (vc.subst_env σ p)
355  notation `⟪` P `⟫`: 100 :=
356    ∀ (σ: env), closed_subst σ P → ⊨ (prop.subst_env σ P).instantiated_n
357
358  -- simple axioms for logical reasoning
359
360  axiom valid.tru:
361    ⊨ value.true
362
363  axiom valid.and {P Q: vc}:
364    (⊨ P) ∧ (⊨ Q)
365    ↔
366    ⊨ P ∧ Q
367
368  axiom valid.or.left {P Q: vc}:
369    (⊨ P) →
370    ⊨ P ∨ Q
371
```

```
372  axiom valid.or.right {P Q: vc}:
373    (⊨ Q) →
374    ⊨ P ⋁ Q
375
376  axiom valid.or.elim {P Q: vc}:
377    (⊨ P ⋁ Q)
378    →
379    (⊨ P) ⋁ (⊨ Q)
380
381  -- no contradictions
382  axiom valid.contradiction {P: vc}:
383    ¬ (⊨ P ⋀ P.not)
384
385  -- law of excluded middle
386  axiom valid.em {P: vc}:
387    (⊨ P ⋁ P.not)
388
389  -- a term is valid if it equals true
390  axiom valid.eq.true {t: term}:
391    ⊨ t
392    ↔
393    ⊨ value.true ≡ t
394
395  -- universal quantifier valid if true for all values
396  axiom valid.univ.mp {x: var} {P: vc}:
397    ∀(v, ⊨ vc.subst x v P)
398    →
399    ⊨ vc.univ x P
400
401  -- a free top-level variable is implicitly universally quantified
402  axiom valid.univ.free {x: var} {P: vc}:
403    (x ∈ FV P ⋀ ⊨ P)
404    →
405    ⊨ vc.univ x P
406
407
408
409
```

```
410  -- universal quantifier can be instantiated with any term
411  axiom valid.univ.mpr {x: var} {P: vc}:
412    (⊨ vc.univ x P)
413    →
414    ∀(t, ⊨ vc.substt x t P)
415
416  -- unary and binary operators are decidable,
417  -- so equalities with operators are decidable
418  axiom valid.unop {op: unop} {v_x v: value}:
419    unop.apply op v_x = some v
420    ↔
421    ⊨ v ≡ term.unop op v_x
422
423  axiom valid.binop {op: binop} {v_1 v_2 v: value}:
424    binop.apply op v_1 v_2 = some v
425    ↔
426    ⊨ v ≡ term.binop op v_1 v_2
427
428  -- can write pre_1 and pre_2 to check domain of operators
429
430  axiom valid.pre_1 {v_x: value} {op: unop}:
431    (⊨ vc.pre_1 op v_x)
432    →
433    option.is_some (unop.apply op v_x)
434
435  axiom valid.pre_2 {v_1 v_2: value} {op: binop}:
436    (⊨ vc.pre_2 op v_1 v_2)
437    →
438    option.is_some (binop.apply op v_1 v_2)
439
440  --  #####################################
441  --  ### VERIFICATION RELATION (VCGEN) ###
442  --  #####################################
443
444  reserve infix `⊢`:10
445
446  -- verification of expressions
447
```

```
448  inductive exp.vcgen : prop → exp → propctx → Prop
449  notation P `⊢` e `:` Q : 10 := exp.vcgen P e Q
450
451  | tru {P: prop} {x: var} {e: exp} {Q: propctx}:
452      x ∉ FV P →
453      (P ∧ x ≡ value.true ⊢ e : Q) →
454      (P ⊢ lett x = true in e : propctx.exis x (x ≡ value.true ∧ Q))
455
456  | fals {P: prop} {x: var} {e: exp} {Q: propctx}:
457      x ∉ FV P →
458      (P ∧ x ≡ value.false ⊢ e : Q) →
459      (P ⊢ letf x = false in e : propctx.exis x (x ≡ value.false ∧ Q))
460
461  | num {P: prop} {x: var} {n: ℕ} {e: exp} {Q: propctx}:
462      x ∉ FV P →
463      (P ∧ x ≡ value.num n ⊢ e : Q) →
464      (P ⊢ letn x = n in e : propctx.exis x (x ≡ value.num n ∧ Q))
465
466  | func {P: prop} {f x: var} {R S: spec} {e₁ e₂: exp} {Q₁ Q₂: propctx}:
467      f ∉ FV P →
468      x ∉ FV P →
469      f ≠ x →
470      x ∈ FV R.to_prop.to_vc →
471      FV R.to_prop ⊆ FV P ∪ { f, x } →
472      FV S.to_prop ⊆ FV P ∪ { f, x } →
473      (P ∧ spec.func f x R S ∧ R ⊢ e₁ : Q₁) →
474      (P ∧ prop.func f x R (Q₁ (term.app f x) ∧ S) ⊢ e₂ : Q₂) →
475      ⟨ prop.implies (P ∧ spec.func f x R S ∧ R ∧ Q₁ (term.app f x)) S ⟩ →
476      (P ⊢ letf f[x] req R ens S {e₁} in e₂ :
477          propctx.exis f (prop.func f x R (Q₁ (term.app f x) ∧ S) ∧ Q₂))
478
479  | unop {P: prop} {op: unop} {e: exp} {x y: var} {Q: propctx}:
480      x ∈ FV P →
481      y ∉ FV P →
482      (P ∧ y ≡ term.unop op x ⊢ e : Q) →
483      ⟨ prop.implies P (prop.pre₁ op x) ⟩ →
484      (P ⊢ letop y = op [x] in e : propctx.exis y (y ≡ term.unop op x ∧ Q))
485
```

```
486  | binop {P: prop} {op: binop} {e: exp} {x y z: var} {Q: propctx}:
487      x ∈ FV P →
488      y ∈ FV P →
489      z ∉ FV P →
490      (P ∧ z ≡ term.binop op x y ⊢ e : Q) →
491      ⟨ prop.implies P (prop.pre₂ op x y) ⟩ →
492      (P ⊢ letop2 z = op [x, y] in e : propctx.exis z (z ≡ term.binop op x y ∧ Q))
493
494  | app {P: prop} {e: exp} {y f x: var} {Q: propctx}:
495      f ∈ FV P →
496      x ∈ FV P →
497      y ∉ FV P →
498      (P ∧ prop.call x ∧ prop.post f x ∧ y ≡ term.app f x ⊢ e : Q) →
499      ⟨ prop.implies (P ∧ prop.call x) (term.unop unop.isFunc f ∧ prop.pre f x) ⟩ →
500      (P ⊢ letapp y = f [x] in e :
501          propctx.exis y (prop.call x ∧ prop.post f x ∧ y ≡ term.app f x ∧ Q))
502
503  | ite {P: prop} {e₁ e₂: exp} {x: var} {Q₁ Q₂: propctx}:
504      x ∈ FV P →
505      (P ∧ x ⊢ e₁ : Q₁) →
506      (P ∧ prop.not x ⊢ e₂ : Q₂) →
507      ⟨ prop.implies P (term.unop unop.isBool x) ⟩ →
508      (P ⊢ exp.ite x e₁ e₂ : propctx.implies x Q₁ ∧ propctx.implies (prop.not x) Q₂)
509
510  | return {P: prop} {x: var}:
511      x ∈ FV P →
512      (P ⊢ exp.return x : x ≣ ∘)
513
514  notation P `⊢` e `:` Q : 10 := exp.vcgen P e Q
515
516
517  -- verification of environments/translation into logic
518  inductive env.vcgen : env → prop → Prop
519  notation `⊢` σ `:` Q : 10 := env.vcgen σ Q
520
521  | empty:
522      ⊢ env.empty : value.true
523
```

181

```
524  | tru {σ: env} {x: var} {Q: prop}:
525      x ∉ σ →
526      (⊢ σ : Q) →
527      (⊢ (σ[x ↦ value.true]) : Q ∧ x ≡ value.true)
528
529  | fls {σ: env} {x: var} {Q: prop}:
530      x ∉ σ →
531      (⊢ σ : Q) →
532      (⊢ (σ[x ↦ value.false]) : Q ∧ x ≡ value.false)
533
534  | num {n: ℤ} {σ: env} {x: var} {Q: prop}:
535      x ∉ σ →
536      (⊢ σ : Q) →
537      (⊢ (σ[x ↦ value.num n]) : Q ∧ x ≡ value.num n)
538
539  | func {σ₁ σ₂: env} {f g x: var} {R S: spec} {e: exp} {Q₁ Q₂: prop} {Q₃: propctx}:
540      f ∉ σ₁ →
541      g ∉ σ₂ →
542      x ∉ σ₂ →
543      g ≠ x →
544      (⊢ σ₁ : Q₁) →
545      (⊢ σ₂ : Q₂) →
546      x ∈ FV R.to_prop.to_vc →
547      FV R.to_prop ⊆ FV Q₂ ∪ { g, x } →
548      FV S.to_prop ⊆ FV Q₂ ∪ { g, x } →
549      (Q₂ ∧ spec.func g x R S ∧ R ⊢ e : Q₃) →
550      ⟨ prop.implies (Q₂ ∧ spec.func g x R S ∧ R ∧ Q₃ (term.app g x)) S ⟩ →
551      (⊢ (σ₁[f ↦ value.func g x R S e σ₂]) :
552         (Q₁
553          ∧ f ≡ value.func g x R S e σ₂
554          ∧ prop.subst_env (σ₂[g ↦ value.func g x R S e σ₂])
555                          (prop.func g x R (Q₃ (term.app g x) ∧ S))))

557  notation `⊢` σ `:` Q : 10 := env.vcgen σ Q

559  --  ###############################
560  --  ### VERIFICATION WITHOUT QI ###
561  --  ###############################
```

182

```
562  -- verification conditions without quantifier instantiation algorithm
563
564  notation `⦃` P `⦄`: 100 := ∀ (σ: env), closed_subst σ P → σ ⊨ P.to_vc
565
566  reserve infix `⊩`:10
567
568  -- verification of expressions
569  inductive exp.dvcgen : prop → exp → propctx → Prop
570  notation P `⊩` e `:` Q : 10 := exp.dvcgen P e Q
571
572  | tru {P: prop} {x: var} {e: exp} {Q: propctx}:
573      x ∉ FV P →
574      (P ⋀ x ≡ value.true ⊩ e : Q) →
575      (P ⊩ lett x = true in e : propctx.exis x (x ≡ value.true ⋀ Q))
576
577  | fals {P: prop} {x: var} {e: exp} {Q: propctx}:
578      x ∉ FV P →
579      (P ⋀ x ≡ value.false ⊩ e : Q) →
580      (P ⊩ letf x = false in e : propctx.exis x (x ≡ value.false ⋀ Q))
581
582  | num {P: prop} {x: var} {n: ℕ} {e: exp} {Q: propctx}:
583      x ∉ FV P →
584      (P ⋀ x ≡ value.num n ⊩ e : Q) →
585      (P ⊩ letn x = n in e : propctx.exis x (x ≡ value.num n ⋀ Q))
586
587  | func {P: prop} {f x: var} {R S: spec} {e₁ e₂: exp} {Q₁ Q₂: propctx}:
588      f ∉ FV P →
589      x ∉ FV P →
590      f ≠ x →
591      x ∈ FV R.to_prop.to_vc →
592      FV R.to_prop ⊆ FV P ∪ { f, x } →
593      FV S.to_prop ⊆ FV P ∪ { f, x } →
594      (P ⋀ spec.func f x R S ⋀ R ⊩ e₁ : Q₁) →
595      (P ⋀ prop.func f x R (Q₁ (term.app f x) ⋀ S) ⊩ e₂ : Q₂) →
596      ⦃ prop.implies (P ⋀ spec.func f x R S ⋀ R ⋀ Q₁ (term.app f x)) S ⦄ →
597      (P ⊩ letf f[x] req R ens S {e₁} in e₂ :
598          propctx.exis f (prop.func f x R (Q₁ (term.app f x) ⋀ S) ⋀ Q₂))
599
```

```
600  | unop {P: prop} {op: unop} {e: exp} {x y: var} {Q: propctx}:
601      x ∈ FV P →
602      y ∉ FV P →
603      (P ⋀ y ≡ term.unop op x ⊩ e : Q) →
604      ⦃ prop.implies P (prop.pre₁ op x) ⦄ →
605      (P ⊩ letop y = op [x] in e : propctx.exis y (y ≡ term.unop op x ⋀ Q))
606
607  | binop {P: prop} {op: binop} {e: exp} {x y z: var} {Q: propctx}:
608      x ∈ FV P →
609      y ∈ FV P →
610      z ∉ FV P →
611      (P ⋀ z ≡ term.binop op x y ⊩ e : Q) →
612      ⦃ prop.implies P (prop.pre₂ op x y) ⦄ →
613      (P ⊩ letop2 z = op [x, y] in e : propctx.exis z (z ≡ term.binop op x y ⋀ Q))
614
615  | app {P: prop} {e: exp} {y f x: var} {Q: propctx}:
616      f ∈ FV P →
617      x ∈ FV P →
618      y ∉ FV P →
619      (P ⋀ prop.call x ⋀ prop.post f x ⋀ y ≡ term.app f x ⊩ e : Q) →
620      ⦃ prop.implies (P ⋀ prop.call x) (term.unop unop.isFunc f ⋀ prop.pre f x) ⦄ →
621      (P ⊩ letapp y = f [x] in e :
622          propctx.exis y (prop.call x ⋀ prop.post f x ⋀ y ≡ term.app f x ⋀ Q))
623
624  | ite {P: prop} {e₁ e₂: exp} {x: var} {Q₁ Q₂: propctx}:
625      x ∈ FV P →
626      (P ⋀ x ⊩ e₁ : Q₁) →
627      (P ⋀ prop.not x ⊩ e₂ : Q₂) →
628      ⦃ prop.implies P (term.unop unop.isBool x) ⦄ →
629      (P ⊩ exp.ite x e₁ e₂ : propctx.implies x Q₁ ⋀ propctx.implies (prop.not x) Q₂)
630
631  | return {P: prop} {x: var}:
632      x ∈ FV P →
633      (P ⊩ exp.return x : x ≣ ∘)
634
635  notation P `⊩` e `:` Q : 10 := exp.dvcgen P e Q
636
637  -- verification of environments/translation into logic
```

184

```
638  inductive env.dvcgen : env → prop → Prop
639  notation `⊩` σ `:` Q : 10 := env.dvcgen σ Q
640
641  | empty:
642      ⊩ env.empty : value.true
643
644  | tru {σ: env} {x: var} {Q: prop}:
645      x ∉ σ →
646      (⊩ σ : Q) →
647      (⊩ (σ[x ↦ value.true]) : Q ∧ x ≡ value.true)
648
649  | fls {σ: env} {x: var} {Q: prop}:
650      x ∉ σ →
651      (⊩ σ : Q) →
652      (⊩ (σ[x ↦ value.false]) : Q ∧ x ≡ value.false)
653
654  | num {n: ℤ} {σ: env} {x: var} {Q: prop}:
655      x ∉ σ →
656      (⊩ σ : Q) →
657      (⊩ (σ[x ↦ value.num n]) : Q ∧ x ≡ value.num n)
658
659  | func {σ₁ σ₂: env} {f g x: var} {R S: spec} {e: exp} {Q₁ Q₂: prop} {Q₃: propctx}:
660      f ∉ σ₁ →
661      g ∉ σ₂ →
662      x ∉ σ₂ →
663      g ≠ x →
664      (⊩ σ₁ : Q₁) →
665      (⊩ σ₂ : Q₂) →
666      x ∈ FV R.to_prop.to_vc →
667      FV R.to_prop ⊆ FV Q₂ ∪ { g, x } →
668      FV S.to_prop ⊆ FV Q₂ ∪ { g, x } →
669      (Q₂ ∧ spec.func g x R S ∧ R ⊩ e : Q₃) →
670      ⫤ prop.implies (Q₂ ∧ spec.func g x R S ∧ R ∧ Q₃ (term.app g x)) S ⫣ →
671      (⊩ (σ₁[f ↦ value.func g x R S e σ₂]) :
672        (Q₁
673        ∧ f ≡ value.func g x R S e σ₂
674        ∧ prop.subst_env (σ₂[g ↦ value.func g x R S e σ₂])
675                          (prop.func g x R (Q₃ (term.app g x) ∧ S))))
```

185

```
676
677  notation `⊩` σ `:` Q : 10 := env.dvcgen σ Q
678
679  --   ####################################################################
680  --   ### AXIOMS ABOUT FUNCTION EXPRESSIONS, PRE and POSTCONDITIONS ###
681  --   ####################################################################
682
683  -- The following equality axiom is non-standard and makes validity undecidable.
684  -- It is only used in the preservation proof of e-return and in no other lemmas.
685  -- The logic treats `f(x)` uninterpreted, so there is no way to derive it naturally.
686  -- However, given a complete evaluation derivation of a function call, we can
687  -- add the equality `f(x)=y` as new axiom for closed values f, x, y and the
688  -- resulting set of axioms is still sound due to deterministic evaluation.
689  axiom valid.app {v_x v: value} {σ σ': env} {f x y: var} {R S: spec} {e: exp}:
690    (σ[f ↦ value.func f x R S e σ][x ↦ v_x], e)⟶* (σ', exp.return y) →
691    (σ' y = some v)
692    →
693    ⊨ v ≡ term.app (value.func f x R S e σ) v_x
694
695  -- can write pre and post to extract pre- and postcondition of function values
696
697  axiom valid.pre {v_x: value} {σ: env} {f x: var} {R S: spec} {e: exp}:
698    (σ[f ↦ value.func f x R S e σ][x ↦ v_x] ⊨ R.to_prop.to_vc)
699    ↔
700    ⊨ vc.pre (value.func f x R S e σ) v_x
701
702  axiom valid.post.mp {v_x: value} {σ: env} {Q: prop} {Q_2: propctx}
703                      {f x: var} {R S: spec} {e: exp}:
704    (⊩ σ : Q) →
705    (Q ∧ spec.func f x R S ∧ R ⊩ e : Q_2) →
706    (σ[f ↦ value.func f x R S e σ][x ↦ v_x] ⊨ (Q_2 (term.app f x) ∧ S.to_prop).to_vc)
707    →
708    (⊨ vc.post (value.func f x R S e σ) v_x)
709
710
711
712
713
```

186

```
714  axiom valid.post.mpr {vₓ: value} {σ: env} {Q: prop} {Q₂: propctx}
715                        {f x: var} {R S: spec} {e: exp}:
716     (⊩ σ : Q) →
717     (Q ∧ spec.func f x R S ∧ R ⊩ e : Q₂) →
718     (⊨ vc.post (value.func f x R S e σ) vₓ)
719     →
720     (σ[f ↦ value.func f x R S e σ][x ↦ vₓ] ⊨ (Q₂ (term.app f x) ∧ S.to_prop).to_vc)
```

A.4 `theorems.lean`

This file includes the soundness theorem for quantifier instantiation and the soundness theorem for the verification rules.

```
1  import .definitions2 .qi .soundness
2
3  -- This theorem states that any proposition `P` that is valid
4  -- with instantiations `⟪ P ⟫` is also a valid proposition
5  -- without quantifier instantiation `∥ P ∥`:
6  theorem vc_valid_without_instantiations (P: prop):
7     ⟪ P ⟫ → ∥ P ∥
8     := @vc_valid_from_inst_valid P -- actual proof in qi.lean
9
10
11 -- This theorem states that a verified source program `e` does not get stuck,
12 -- i.e. its evaluation always results either in a value or
13 -- in a runtime stack `s` that can be further evaluated.
14 -- The proof internally uses lemmas for progress and preservation.
15 theorem verification_safety (e: exp) (s: stack) (Q: propctx):
16    (value.true ⊢ e: Q) → ((env.empty, e)⟶* s) → (is_value s ∨ ∃s', s⟶s')
17    := @soundness_source_programs e s Q -- actual proof in soundness.lean
```

187

■ Appendix B

User Study Tutorial and Experiments

This appendix lists all tutorial steps and programming tasks that were part of the user study. Instructions, code and hints were displayed as a series of online web pages with a live web-based programming environment. An archived version of the user study including tutorial and programming tasks is available online at

`https://esverify.org/userstudy-archived`.

B.1   Tutorial 1: JavaScript Live Editing

Edit and run a simple JavaScript program

Instructions

This user study involves interactions with a programming environment. The source code can be edited directly and the program can be executed in the browser. Test the editor by fixing the JavaScript program such that it computes the correct area of a rectangle.

Provided Code

```
1  // This is a live demo, simply edit the code and click run
2
3  const height = 3;
4  const width = 4;
5  const area_of_rect = height * height;
6
7  // should print '12', but prints '9' instead
8  alert(area_of_rect)
```

Steps and Hints

1. Click the **run** button to see the result of the computation.

2. Change the source code to compute the correct area of a rectangle.

3. Click the **run** button again to test the code.

B.2 Tutorial 2: Program Verification With Pre- and Postconditions

Verify the given annotated `max` function and fix potential issues.

Instructions

ESVERIFY extends JavaScript with special syntax to annotate functions with pre- and postconditions. These are written as pseudo function calls that are skipped during evaluation. The following example includes an incorrect `max` function that should be fixed such that it returns the maximum of its arguments and verification succeeds.

Provided Code

```
1  // returns the maximum of the two provided numbers
2  function max(a, b) {
3    requires(typeof(a) === 'number');
4    requires(typeof(b) === 'number');
5    ensures(res => res >= a);
6    ensures(res => res >= b);// postcondition does not hold
7
8    if (a >= b) {
9      return a;
10   } else {
11     return a; // <- due to a bug in the implementation
12   }
13 }
```

Steps and Hints

1. Click the **verify** button to verify all assertions in the code.

2. The second postcondition does not hold due to a bug in the implementation.

3. Change the source code to return the correct maximum of a and b.

4. Click the **verify** button again to ensure that the new code verifies.

B.3   Tutorial 3: Interactive Verification Condition Inspector

Inspect a verification issue to understand and interactively explore assumptions and assertions.

Instructions

The following example includes a `max` function with missing preconditions. To better understand the problem, the ESVERIFY programming environment provides an interactive inspector for verification conditions that explains assumptions, assertions and counterexamples if available. This inspector also allows interactively adding assumptions and assertions.

Provided Code

```
1  // returns the maximum of the two provided numbers
2  function max(a, b) {
3    ensures(res => res >= a);
4    ensures(res => res >= b);
5
6    if (a >= b) {
7      return a;
8    } else {
9      return b;
10   }
11 }
```

Steps and Hints

1. Click the **verify** button to verify all assertions in the code.

2. Click on the yellow triangle in front of line 3 to select the verification condition.

3. The panel on the right lists assumptions and assertions and the editor shows values for the counterexample as popup markers.

4. According to the editor popups, the postcondition does not hold if the arguments are not numbers. Check this hypothesis by entering **typeof** a === 'number' next to 'Assume:' and confirm this with by pressing the enter/return key.

191

5. Also add the assumption **typeof** b === 'number'.

6. With these assumptions, the postcondition can be verified.

B.4 Tutorial 4: Verification and Debugger Integration

Query the counterexample and step through the code.

Instructions

For each unverified verification condition, the counterexample values can be used to execute the code with an interactive debugger. The debugger shows variables in scope, the current call stack and allows step-by-step debugging. Additionally, the debugger can be queried with watch expressions.

Provided Code

```
1  // returns the maximum of the two provided numbers
2  function max(a, b) {
3    requires(typeof(a) === 'number');
4    requires(typeof(b) === 'number');
5    ensures(res => res >= a);
6    ensures(res => res >= b);
7
8    if (a > b) {
9      return a;
10   }
11   if (b > a) {
12     return b;
13   }
14 }
```

Steps and Hints

1. Click the **verify** button to verify all assertions in the code.

2. Click on the first incorrect verification condition in line 5.

3. The verification inspector in the panel on the right lists watch expressions, variables in scope and the call stack.

4. In this case, the counterexample uses `0` for both `a` and `b`.

5. To query the return value, enter `res` next to 'Watch:'.

6. It seems the function returns `undefined`.

7. To see the control flow, step through the code by clicking **Restart** and then clicking **Step Into** about eight times.

8. It seems none of the two **if** statements returned a value when stepping through the code with this counterexample.

B.5   Experiment 1: Factorial

Instructions

This first experiment involves an incorrect factorial function. This example can either be fixed by adding a stronger precondition or by changing the **if** statement. You can use the verification inspector and the integrated counterexample debugger. Click 'Next' if you fixed the example or if you want to move to the next experiment.

Provided Code

```
1  // returns the factorial of the provided argument
2  function factorial(n) {
3    requires(Number.isInteger(n));
4    ensures(res => res >= 1);
5
6    if (n === 0) {
7      return 1;
8    } else {
9      return factorial(n - 1) * n;
10   }
11 }
```

Steps and Hints

No steps or hints.

B.6   Experiment 2: Dice Rolls

Instructions

This experiment involves an function for rolling a six-sided dice. A correct implementa-
tion is given and the following assertions should be verifiable but the postconditions are
missing. The verification inspector is not available. Click 'Next' if you fixed the example
or if you want to move to the next experiment.

Provided Code

```
1  // Roll a six-sided dice
2  function rollDice () {
3    // missing annotations
4    // ensures(res => ...);
5
6    return Math.trunc(Math.random() * 6) + 1;
7  }
8
9  const r = rollDice() + rollDice() + rollDice();
10 assert(r >= 3);
11 assert(r <= 18);
```

Steps and Hints

1. Add missing postconditions with **ensures**(res => ...); in order to verify the assertions.

2. You can verify code but there is no verification inspector.

3. Click 'Next' if you fixed the example or if you want to move to the next experiment.

B.7   Experiment 3: Digital 24 Hour Clock

Instructions

This is the third and final experiment of this user study. Given the number of minutes since midnight, you should return time in a 24-hour digital clock format. You need to add an additional precondition and change the returned value. (Hint: Math.trunc rounds a number down to an integer.) You can use the verification inspector and the

editor counterexample popups. Click 'Next' if you fixed the example or if you want to finish the experiments.

Provided Code

```
1   // Given the number of minutes since midnight,
2   // returns the current hour and minute as object
3   // in a { h: 0-23, m: 0-59 } format
4   function clock_24 (min) {
5     requires(Number.isInteger(min) && 0 <= min);
6
7     ensures(res => res instanceof Object &&
8             'h' in res && 'm' in res);
9     ensures(res => Number.isInteger(res.h) &&
10            0 <= res.h && res.h < 24);
11    ensures(res => Number.isInteger(res.m) &&
12            0 <= res.m && res.m < 60);
13
14    return {
15      h: min / 60,
16      m: min % 60
17    };
18  }
```

Steps and Hints

1. You need to add an additional precondition and change the returned value. (Hint: `Math.trunc` rounds a number down to an integer.)

2. You can use the verification inspector and the editor counterexample popups.

3. Click 'Next' if you fixed the example or if you want to finish the experiments.

■ Appendix C

User Study Survey Answers

This appendix lists all survey answers by participants in the user study. Test subjects were given a series of online tutorials and programming tasks as listed in Appendix B, and then filled out an online survey. For features of the programming environment, participants could either select a given response or type their own answer.

Participant 1

JavaScript experience: 5/5   **Program verification experience:** No

Verification Inspector	Did not use it	It impairs the development process
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Unsuccessfully tried using it	The feature is helpful
Comments:		

Participant 2

JavaScript experience: 5/5   **Program verification experience:** Yes

Verification Inspector	Used this feature in experiments	The feature is helpful
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Used this feature in experiments	It could be helpful with different UI

Comments: I liked clicking on particular postconditions and being able to see counterexamples and add preconditions in the "scratch pad" area.

Participant 3

JavaScript experience: 4/5   **Program verification experience:** Yes

Verification Inspector	Used this feature in experiments	The feature is helpful
Counterexample Popups	Did not use it	It could be helpful with different UI
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments: JS is most useful on front end development. But how do you make assumptions/assertions for those UI/Networking related things?

Participant 4

JavaScript experience: 3/5   **Program verification experience:** Yes

Verification Inspector	Used this feature in experiments	It could be helpful with different UI
Counterexample Popups	Did not use it	It could be helpful with different UI
Integrated Debugger	Did not use it	The feature is helpful

Comments: For a while I couldn't even tell that the debugging window was there. Perhaps have it always visible but only populated when something is selected.

Participant 5

JavaScript experience: 2/5   **Program verification experience:** Yes

Verification Inspector	I tried to but was not successful.	It impairs the development process
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Used this feature in experiments	The feature is helpful

Comments: I've never worked with assume or ensure before, and your modal text felt very jargon-y to me.

Participant 6

JavaScript experience: 5/5   **Program verification experience:** Yes

Verification Inspector	I tried to but was not successful.	The feature is helpful
Counterexample Popups	Did not use it	It is not useful for development
Integrated Debugger	Did not use it	For more complex examples, but I can't judge the development experience as I did not use it.

Comments: I broke the interface when I tried to enter an assumption.

Participant 7

JavaScript experience: 5/5   **Program verification experience:** No

Verification Inspector	I tried to but was not successful.	It could be helpful with different UI
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Used this feature in experiments	It could be helpful with different UI

Comments: Keep up the good work!

Participant 8

JavaScript experience: 3/5   **Program verification experience:** Yes

Verification Inspector	I tried to but was not successful.	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	It could be helpful with different UI
Integrated Debugger	Unsuccessfully tried using it	It could be helpful with different UI

Comments: The tutorial window often covered parts of the interface that I was required to interact with and there was no way to dismiss it which made it challenging to use the interface particularly in the example using the debugger. Entering assumptions in the right panel does not seem to add them or update the code in the editor which was confusing and makes the interface not seem very useful if they need to be entered in two separate places.

Participant 9

JavaScript experience: 5/5   **Program verification experience:** Yes

Verification Inspector	Did not use it	The feature is helpful
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	The feature is helpful

Comments: counterexample of 1499 for min was super helpful!!

Participant 10

JavaScript experience: 1/5   **Program verification experience:** Yes

Verification Inspector	I entered the ones from the tutorial as suggested	Yes, but it also calls for more logical foundation in introductory classes.
Counterexample Popups	I would have liked a cheat sheet for the parts that were not a tutorial. Maybe I'm a little tired and can't figure this out right now. Getting the answer like in the Tutorial would give me something to chew on.	The feature is helpful
Integrated Debugger	Used this feature in experiments	The feature is helpful

Comments: What I want to know as a lay person is will there be some AI auto-pilot who not only can tell me something is wrong, as this does, but pretty much gives me the right answer every time. That's what I want, a glorified spellchecker.

Participant 11

JavaScript experience: 2/5   **Program verification experience:** No

Verification Inspector	Used this feature in experiments	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments: I didn't like the font in the UI. For instance the difference between $==$ and $===$ wasn't immediately obvious. For someone who doesn't program in JavaScript much this made the tutorial more difficult to follow. But overall it's an interesting prototype and interactive development of assertions along with hints from assertion editor seems like a useful tool for learning how assertions work and adding them in the code.

Participant 12

JavaScript experience: 5/5   **Program verification experience:** Yes

Verification Inspector	Did not use it	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments: I had some UI difficulty with the specific implementation of adding assumptions (e.g. having to add them in multiple places) and another with the interface for the debugger being very narrow and requiring scrolling to see the whole thing. I did find this tool very compelling, although the hints to some extent obviated the need for the tool. It is very difficult to evaluate stuff like this.

Participant 13

JavaScript experience: 4/5   **Program verification experience:** No

Verification Inspector	Did not use it	The feature is helpful
Counterexample Popups	Unsuccessfully tried using it	It could be helpful with different UI
Integrated Debugger	Did not use it	It could be helpful with different UI

Comments:

Participant 14

JavaScript experience: 4/5   **Program verification experience:** Yes

Verification Inspector	Did not use it	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Unsuccessfully tried using it	The feature is helpful

Comments:

Participant 15

JavaScript experience: 4/5   **Program verification experience:** No

Verification Inspector	I think I did on one of them but don't remember.	It is not useful for development
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	The feature is helpful

Comments: The counterexamples would be the most helpful contribution of this project to my own programming practice. It makes it easier and quicker to comprehensively test a function and catch edge cases, because I don't have to come up with the test values or edge cases myself. And it feels better to me to code pre and post conditions explicitly, rather than write them in comments and reference comments or other documentation whenever I use the function in a sort of new way. I would want to write requires() and ensures() statements in my own Javascript programs.

Participant 16

JavaScript experience: 3/5   **Program verification experience:** No

Verification Inspector	Unsuccessfully tried using it	It could be helpful with different UI
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Used this feature in experiments	It could be helpful with different UI

Comments:

Participant 17

JavaScript experience: 4/5   **Program verification experience:** No

Verification Inspector	They weren't visible clearly since the instructions/help panel was covering it	The feature is helpful
Counterexample Popups	Used this feature in experiments	The feature is helpful
Integrated Debugger	Did not use it	It impairs the development process

Comments: Great tool!

Participant 18

JavaScript experience: 4/5   **Program verification experience:** No

Verification Inspector	Used this feature in experiments	It could be helpful with different UI
Counterexample Popups	Unsuccessfully tried using it	The feature is helpful
Integrated Debugger	Unsuccessfully tried using it	The feature is helpful

Comments:

Bibliography

[1]   Umut A. Acar. "Self-adjusting Computation: (an Overview)". In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '09. Savannah, GA, USA: ACM, 2009, pp. 1–6. ISBN: 978-1-60558-327-3. DOI: 10.1145/1480945.1480946.

[2]   Amal Ahmed et al. "Blame for All". In: *POPL '11*. 2011.

[3]   Rajeev Alur et al. "Syntax-guided synthesis". In: *Dependable Software Systems Engineering* (2015).

[4]   Esben Andreasen and Anders Møller. "Determinacy in Static Analysis for jQuery". In: *OOPSLA '14*. 2014.

[5]   Ryoya Arai, Shigeyuki Sato, and Hideya Iwasaki. "A Debugger-Cooperative Higher-Order Contract System in Python". In: *Programming Languages and Systems*. 2016.

[6]   Stephan Arlt et al. "The Gradual Verifier". In: *NASA Formal Methods*. 2014. ISBN: 978-3-319-06200-6.

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

[8]   Clark Barrett et al. "CVC4". In: *CAV'11*. 2011.

[9]   Samuel Baxter et al. "Putting in All the Stops: Execution Control for JavaScript". In: *PLDI'18*.

[10]   Bernhard Beckert, Sarah Grebing, and Alexander Weigl. "Debugging Program Verification Proof Scripts (Tool Paper)". In: *CoRR* (2018).

[11]   Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. 2007.

[12]   Gavin Bierman, Martin Abadi, and Mads Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281.

[13]   Elisa Gonzalez Boix et al. "Object-oriented Reactive Programming is Not Reactive Object-oriented Programming". In: *REM'13* (2013).

[14] Sebastian Burckhardt et al. "It's Alive! Continuous Feedback in UI Programming". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 95–104. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462170.

[15] Miguel Campusano, Alexandre Bergel, and Johan Fabry. "Does Live Programming Help Program Comprehension?" In: (2016).

[16] Adam Chlipala. "Ur/Web: A Simple Model for Programming the Web". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 153–165. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2677004.

[17] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An Interpolating SMT Solver". In: *SPIN'12*. Oxford, UK, 2012.

[18] M. Christakis, P. Müller, and V. Wüstholz. "Guiding Dynamic Symbolic Execution toward Unverified Program Executions". In: *ICSE'16*. 2016.

[19] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Collaborative Verification and Testing with Explicit Assumptions". In: *FM'12*. 2012.

[20] Maria Christakis et al. "Integrated Environment for Diagnosing Verification Errors". In: *TACAS'16*. 2016.

[21] Ravi Chugh, David Herman, and Ranjit Jhala. "Dependent Types for JavaScript". In: *OOPSLA '12*. 2012.

[22] Ravi Chugh et al. "Programmatic and Direct Manipulation, Together at Last". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, June 2016.

[23] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64.

[24] Simon Cruanes and Jasmin Blanchette. "Extending Nunchaku to Dependent Type Theory". In: *Hammers for Type Theories (HaTT 2016)*. Vol. 210. 2016, pp. 3–12.

[25] Evan Czaplicki and Stephen Chong. "Asynchronous Functional Reactive Programming for GUIs". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 411–422. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462161.

[26] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. "Just Test What You Cannot Verify!" In: *Fundamental Approaches to Software Engineering*. 2015.

[27] Claire Dross et al. "Adding decision procedures to SMT solvers using axioms with triggers". In: *Journal of Automated Reasoning* (2016).

[28] ECMA-262. *ECMAScript 2017 Language Specification*. 6 / 2017. 2017.

[29] Jonathan Edwards. "Subtext: Uncovering the Simplicity of Programming". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 505–518. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094851.

[30] M.D. Ernst et al. "Dynamically discovering likely program invariants to support program evolution". In: *Software Engineering, IEEE Transactions on* 27.2 (Feb. 2001), pp. 99–123. ISSN: 0098-5589. DOI: 10.1109/32.908957.

[31] R. S. Fabry. "How to Design a System in Which Modules Can Be Changed on the Fly". In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 470–476.

[32] Cormac Flanagan et al. "Extended Static Checking for Java". In: *PLDI'02*. 2002.

[33] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *PLDI '93*. 1993.

[34] José Fragoso Santos et al. "JaVerT: JavaScript verification toolchain". In: *POPL'18* (2018).

[35] Carlo Alberto Furia and Bertrand Meyer. "Inferring Loop Invariants Using Postconditions". In: *Fields of Logic and Computation*. 2010.

[36] Joel Galenson et al. "CodeHint: Dynamic and Interactive Synthesis of Code Snippets". In: *ICSE 2014*. 2014.

[37] X. Ge et al. "DyTa: dynamic symbolic execution guided with static verification results". In: *ICSE'11*. 2011.

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

[39] Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang. "Analyzing Software Updates: Should You Build a Dynamic Updating Infrastructure?" In: *Fundamental Approaches to Software Engineering (FASE)*. Saarbrücken, Germany, Apr. 2011.

[40] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6.

[41] Sumit Gulwani. "Automating String Processing in Spreadsheets Using Input-output Examples". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0.

[42] Philip J. Guo. "Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: ACM, 2013, pp. 579–584.

[43]   Christopher Michael Hancock. "Real-time Programming and the Big Ideas of Computational Literacy". AAI0805688. PhD thesis. Cambridge, MA, USA, 2003.

[44]   Phillip Heidegger and Peter Thiemann. "Contract-Driven Testing of JavaScript Code". In: *Objects, Models, Components, Patterns*. 2010.

[45]   Johannes Henkel and Amer Diwan. "Discovering Algebraic Specifications from Java Classes". English. In: *ECOOP 2003 – Object-Oriented Programming*. Ed. by Luca Cardelli. Vol. 2743. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 431–456. ISBN: 978-3-540-40531-3. DOI: 10.1007/978-3-540-45070-2_19.

[46]   Martin Hentschel, Richard Bubel, and Reiner Hähnle. "The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more". In: *International Journal on Software Tools for Technology Transfer* (2018).

[47]   Michael Hicks, Jonathan T. Moore, and Scott Nettles. "Dynamic Software Updating". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, 2001, pp. 13–23. ISBN: 1-58113-414-2.

[48]   Robert Hirschfeld et al. "Dynamic Contract Layers". In: *SAC '10*. 2010.

[49]   Stefan Huster et al. "Using Robustness Testing to Handle Incomplete Verification Results When Combining Verification and Testing Techniques". In: *Testing Software and Systems*. 2017.

[50]   R Jiménez-Peris et al. "Towards Truly Educational Programming Environments". In: *Computer science education in the 21st century*. Springer, 2000, pp. 81–111.

[51]   David H. Jonassen et al. *Learning to Solve Problems with Technology: A Constructivist Perspective (2nd Edition)*. Prentice Hall, Aug. 2002. ISBN: 0130484032. URL: http://www.worldcat.org/isbn/0130484032.

[52]   Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. ACM. New York, NY, USA: ACM, 2013. ISBN: 978-1-4503-2041-2. DOI: 10.1145/2489798.2489802.

[53]   Matthias Keil and Peter Thiemann. "Blame Assignment for Higher-order Contracts with Intersection and Union". In: *ICFP'15*. 2015.

[54]   Caitlin Kelleher and Randy Pausch. "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers". In: *ACM Comput. Surv.* 37.2 (June 2005), pp. 83–137. ISSN: 0360-0300. DOI: 10.1145/1089733.1089734. URL: http://doi.acm.org/10.1145/1089733.1089734.

[55]   Casey Klein, Matthew Flatt, and Robert Bruce Findler. "Random Testing for Higher-order, Stateful Programs". In: *OOPSLA '10*. 2010.

[56] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[57] Kenneth Knowles and Cormac Flanagan. "Hybrid Type Checking". In: *TOPLAS* (2010).

[58] GE Kransner and S Pope. "Cookbook for using the Model-View-Controller User Interface paradigm". In: *Object Oriented Programming* (1988), pp. 26–49.

[59] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. "The Boogie Verification Debugger (Tool Paper)". In: *Software Engineering and Formal Methods*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. 2011.

[60] K. Rustan M. Leino. "Accessible Software Verification with Dafny". In: *IEEE Software* (2017).

[61] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR'10*. 2010.

[62] K. Rustan M. Leino. "Developing Verified Programs with Dafny". In: *ICSE'13*. 2013.

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

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

[65] K. Rustan M. Leino and Valentin Wüstholz. "The Dafny Integrated Development Environment". In: *Workshop on Formal Integrated Development Environment, F-IDE 2014*. 2014.

[66] Remo Lemma and Michele Lanza. "Co-evolution As the Key for Live Programming". In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE '13. San Francisco, California: IEEE Press, 2013, pp. 9–10. ISBN: 978-1-4673-6265-8.

[67] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communication of the ACM* 52.7 (July 2009), pp. 107–115.

[68] Bil Lewis and Mireille Ducasse. "Using events to debug Java programs backwards in time". In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '03. Anaheim, CA, USA: ACM, 2003, pp. 96–97. ISBN: 1-58113-751-6.

[69] Tom Lieber, Joel R. Brandt, and Rob C. Miller. "Addressing Misconceptions About Code with Always-on Programming Visualizations". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. Toronto, Ontario, Canada: ACM, 2014, pp. 2481–2490. ISBN: 978-1-4503-2473-1. DOI: 10.1145/2556288.2557409.

[70] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[71] Henry Lieberman and Christopher Fry. "Bridging the Gulf Between Code and Behavior in Programming". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '95. Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 480–486. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223969.

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

[73] John H. Maloney and Randall B. Smith. "Directness and Liveness in the Morphic User Interface Construction Environment". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 21–28. ISBN: 0-89791-709-X.

[74] Sean McDirmid. "Living It Up with a Live Programming Language". In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 623–638. ISBN: 978-1-59593-786-5.

[75] Sean McDirmid. "Usable Live Programming". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 53–62. ISBN: 978-1-4503-2472-4.

[76] Sean McDirmid and Jonathan Edwards. "Programming with Managed Time". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. Portland, Oregon, USA: ACM, 2014, pp. 1–10. ISBN: 978-1-4503-3210-1.

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

[78] Leonardo de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. 2015, pp. 378–388.

[79] Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". In: *TOPLAS* (1979).

[80] Hoang Duong Thien Nguyen et al. "SemFix: Program Repair via Semantic Analysis". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 772–781. ISBN: 978-1-4673-3076-3.

[81] Phuc C. Nguyen and David Van Horn. "Relatively Complete Counterexamples for Higher-order Programs". In: *PLDI '15*. 2015.

[82] Phuc C. Nguyen et al. "Soft Contract Verification for Higher-order Stateful Programs". In: *POPL'17* (2017).

[83] Jens Nicolay et al. "Detecting Function Purity in JavaScript". In: *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. SCAM '15. Bremen, DE, Sept. 2015.

[84] Oscar Marius Nierstrasz et al. "Deep Into Pharo: Versioning Your Code with Monticello". In: (2013).

[85] Donald A Norman. *Cognitive engineering*. 1986.

[86] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

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

[88] Patrick Rein et al. "Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness". In: *The Art, Science, and Engineering of Programming* 3 (2018).

[89] Bob Reynders, Dominique Devriese, and Frank Piessens. "Multi-tier Functional Reactive Programming for the Web". In: *Onward! 2014*. ACM, Oct. 2014, pp. 55–68.

[90] Andrew Reynolds et al. "Quantifier Instantiation Techniques for Finite Model Finding in SMT". In: *CADE'13*. 2013.

[91] Jay W Roberts. *Beyond Learning by Doing: Theoretical Currents in Experiential Education*. ERIC, 2011.

[92] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland: ACM, 2014, pp. 25–36. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577083.

[93] Erik Sandewall. "Programming in an Interactive Environment: The "Lisp" Experience". In: *ACM Comput. Surv.* 10.1 (Mar. 1978), pp. 35–71. ISSN: 0360-0300. DOI: 10.1145/356715.356719.

[94] Christopher Schuster, Tim Disney, and Cormac Flanagan. "Macrofication: Refactoring by Reverse Macro Expansion". In: *Programming Languages and Systems: 25th European Symposium on Programming*. ESOP 2016. Eindhoven, NL, Apr. 2016. URL: http://dx.doi.org/10.1007/978-3-662-49498-1_25.

[95] Christopher Schuster and Cormac Flanagan. "A Light-Weight Effect System for JavaScript". In: *Proceedings of the 2015 Scripts to Programs Workshop*. STOP '15. Prague, CZ, July 2015.

[96] Christopher Schuster and Cormac Flanagan. "Reactive Programming with Reactive Variables". In: *Constrained and Reactive Objects Workshop, MODULARITY Companion 2016*. CROW 2016. Malaga, Spain, Mar. 2016. URL: http://doi.acm.org/10.1145/2892664.2892666.

[97] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. "Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong)". In: *ICFP'16*. 2016.

[98] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. "A survey of dynamic software updating". In: *Journal of Software: Evolution and Process* 25.5 (2013), pp. 535–568. ISSN: 2047-7481. DOI: 10.1002/smr.1556.

[99] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081750.

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

[101] B. Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages". In: *Computer* 16.8 (Aug. 1983), pp. 57–69. ISSN: 0018-9162.

[102] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages". In: *Workshop on Scheme and Functional Programming*. 2006.

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

[104] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. "From Program Verification to Program Synthesis". In: *POPL '10: Proceedings of the 37th ACM SIGACT-SIGPLAN conference on Principles of Programming Languages*. 2010.

[105] Andre Staltz. *Unidirectional User Interface Architectures*. http://staltz.com/unidirectional-user-interface-architectures.html. Blog. 2015.

[106] Ofer Strichman and Rachel Tzoref-Brill, eds. *Haifa Verification Conference, HVC 2017*. 2017.

[107] A. J. Summers and P. Müller. "Automating Deductive Verification for Weak-Memory Programs". In: *TACAS'2018*. 2018.

[108] Suresh Thummalapenta et al. "Synthesizing Method Sequences for High-coverage Testing". In: *OOPSLA '11*. 2011.

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

[110] Nikolai Tillmann and Wolfram Schulte. "Parameterized Unit Tests". In: *ESEC/FSE-13*. 2005.

[111] Emina Torlak and Rastislav Bodik. "Growing Solver-aided Languages with Rosette". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 135–152. ISBN: 978-1-4503-2472-4.

[112] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. "JIT Feedback—what Experienced Developers like about Static Analysis". In: *International Conference on Program Comprehension* (2018).

[113] Tom Van Cutsem and Mark S. Miller. "Proxies: Design Principles for Robust Object-oriented Intercession APIs". In: *Proceedings of the 6th Symposium on Dynamic Languages*. DLS '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 59–72. ISBN: 978-1-4503-0405-4.

[114] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. "A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq". In: 2017.

[115] Niki Vazou et al. "Refinement Reflection: Complete Verification with SMT". In: *POPL'18*. 2018.

[116] Niki Vazou et al. "Refinement Types for Haskell". In: *ICFP'14*. 2014.

[117] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Refinement Types for TypeScript". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, 2016, pp. 310–325.

[118] Xiaoyin Wang et al. "Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 2012, 16:1–16:11. ISBN: 978-1-4503-1614-9.

[119] Xinyu Wang, Isil Dillig, and Rishabh Singh. "Program Synthesis Using Abstraction Refinement". In: *POPL'18* (2018).

[120] Greta Yorsh, Thomas Ball, and Mooly Sagiv. "Testing, Abstraction, Theorem Proving: Better Together!" In: *ISSTA '06*. 2006.