CERTIFIED REASONING FOR FIXPOINTS AND CONCRETE EXECUTION IN MATCHING LOGIC

Draft: v10.24; 2024-10-24 at 11 : 12

BY

NISHANT RODRIGUES

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Grigore Rosu, Chair, Advisor
Professor José Meseguer
Professor Santiago Escobar
Professor Lingming Zhang

ABSTRACT

Abstract

ACKNOWLEDGMENTS

Acknowledgments

TABLE OF CONTENTS

Part I

Introduction

There is a need for a more systematic approach to programming language ecosystem development. Today, besides the obligatory compiler or interpreter, programming languages are expected to come with an ecosystem of supporting tools, ranging from syntax highlighters and automated refactoring engines, to static and dynamic analysis tools, and even deductive verifiers. To provide a *coherent* ecosystem, all but the simplest of these tools require a deep understanding of the language's semantics, besides the working of these various tools.

Typically, this consensus is formed around a natural language document describing the programming language's behaviour. This comes with the expected pitfalls—it is difficult to write an unambiguous, comprehensive natural language specification. And often, even this was only added ex post facto, sometimes only decades later [1].

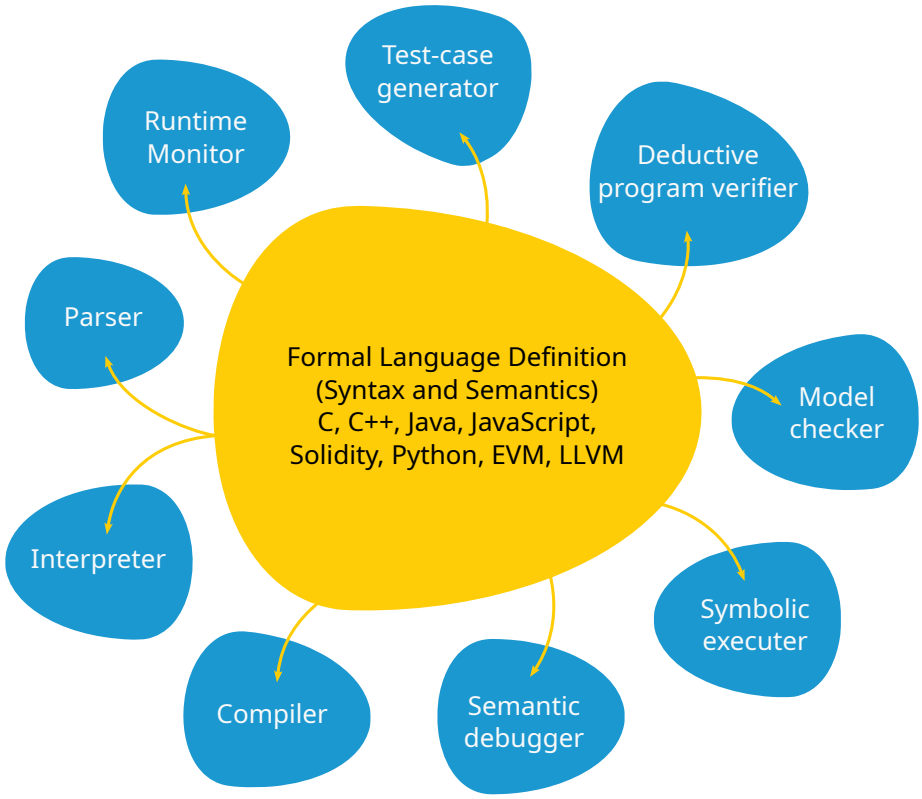


Figure 1: The ideal programming language framework as prescribed by the semantics-first approach.

A proposed solution to this quandary is the semantics-first approach, illustrated in Figure 1. This approach prescribes that the first step to programming language development is to define a *formal* mathematical semantics for the language. Next, each ecosystem tool is derived from this formal semantics.

Since each tool is derived from the same source of truth, confidence of correctness in one tool builds confidence in each of the others. For example, by running a large test suite against a language's interpreter, we gain confidence in its verification infrastructure. This is in contrast to the traditional method of software development where each tool must stand on its own in terms of correctness.

The $\mathbb{K}$ Framework attempts to implement such an ideal language framework. Many languages, including C [2], C++, JavaScript [3], and EVM [4] have their semantics defined in $\mathbb{K}$. Indeed, the C and the JavaScript semantics uncovered bugs in popular implementations of these languages, while the EVM semantics is commercial use, demonstrating the practicality of this approach. The framework generates a parser, interpreter, compiler, and deductive verifier among other tools.

$\mathbb{K}$'s success means that it has grown in both size and complexity, making it correspondingly more difficult to be sure of its correctness. Its implementation is currently more than 500,000 lines of code and employs five different languages—Java, C++, LLVM, Haskell, and Python. At the same time it has a fast pace of development—tens of commits a day. To compound the problem, each of $\mathbb{K}$'s tools is a blackbox, simply returning "yes" or "no" to a complex verification task.

Much of this complexity is necessary. For example, in order to implement fast concrete execution the framework compiles the formal language semantics into a native interpreter. This is done through a C++ tool that generates code in the LLVM intermediate representation that compiles into native machine code. This is the industry standard and recommended way of writing optimizing compilers.

While necessary, this complexity forces us to seriously reckon with the correctness of $\mathbb{K}$. Unfortunately, the scale of the problem makes full formal verification difficult. So we turn instead to a technique called *verified computing* to tackle it. In verified computing, a program is instrumented to emit an artifact that allows us to confirm the correctness of the result without re-running the entire algorithm.

$\mathbb{K}$ has its mathematical foundations in matching-μ logic, making it the natural choice to represent these proofs. Each language semantics written in $\mathbb{K}$ is an abstract representation of a formal matching logic theory. Programs and program states become terms in that logic, while tasks, such as execution, become proof obligations. For example, the semantics of a simple imperative language, IMP, may be captured by the matching logic theory Γ_{IMP}. The execution of a program SUM $\equiv$ `while(n > 0) { s = s + n; n--; }` in this language may

be encoded as the following matching logic formula:

$$\Gamma_{\text{IMP}} \vdash \langle\langle \text{SUM}, s \mapsto 0; n \mapsto 10\rangle\rangle \Rightarrow^* \langle\langle \text{noop}, s \mapsto 55; n \mapsto 0\rangle\rangle$$

Here, Γ_{IMP} represents the mathematical semantics of the programming language, defined in $\mathbb{K}$ and transformed into a matching logic theory. $\Rightarrow^*$ is a relation denoting that the left hand side program state reaches the right hand side in a finite number of steps. This is a formal statement, and can even be proved using matching logic's proof system.

Our long term goal is to instrument each of $\mathbb{K}$'s tools to produce formal proofs for each of these tasks. Producing proofs of execution involves detailed reasoning about program states, framing, domain reasoning, reasoning about constructors modulo associativity, commutativity, and much more. Proofs of reachability, and of program equivalence build on this, involving not just concrete but also *symbolic* execution, as well as potentially complex fixpoint reasoning. As a step towards this, the immediate goal of this thesis is to produce such certificates for concrete program execution and fixpoint reasoning.

The work described in this thesis is broken up into three parts:

1. Establishing the practicality of the $\mathbb{K}$ framework for developing language semantics. We do this by developing formal semantics for two languages.
2. Fixpoint reasoning in matching logic. Since much of $\mathbb{K}$ formal reasoning involves fixpoints, this is important to producing proofs for these.
3. Developing infrastructure for efficiently producing proofs of concrete execution. This builds on the work of [5], by improving the effiency of proof generation and checking.

Let us first review some preliminaries.

CHAPTER 1: PRELIMINARIES

Matching logic has three parts—a syntax of formulae, also called *patterns*; a semantics, defining a satisfaction relation $\models$; and a Hilbert-style proof system, defining a provablility relation $\vdash$.

1.1 SYNTAX

Matching logic formulae, or *patterns*, are built from propositional operators, symbol applications, variables, quantifiers, and a fixpoint binder.

Definition 1.1. Let EVar, SVar, Σ be disjoint countable sets. Here, EVar contains *element variables*, SVar contains *set variables* and the *signature* Σ is set of *symbols*. A Σ-pattern over Σ is defined by the grammar:

$$\varphi := \underbrace{\sigma \mid (\varphi_1\ \varphi_2)}_{\text{structure}} \mid \underbrace{\bot \mid \varphi_1 \to \varphi_2}_{\text{logic}} \mid \underbrace{x \mid \exists x.\, \varphi}_{\text{abstraction}} \mid \underbrace{X \mid \mu X.\, \varphi}_{\text{fixpoint}}$$

Notice that these operators are very low level, and would be tedious to work with. Fortunately, [6] shows us that we can easily build up to more convenient abstractions. For the rest of this work we will work with the following higher level of abstraction:

$$\varphi := \sigma(\varphi_1, \ldots, \varphi_n) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 = \varphi_2 \mid \varphi_1 \subseteq \varphi_2 \mid x \mid \exists x.\, \varphi \mid X \mid \mu X.\, \varphi$$

Multi-arity symbol application may be defined as nested binary application, and equality and subset may be defined in terms of the remaining operators. We assume the usual notation for operators such as $\top$, $\vee$, $\wedge$, $\forall$, ν etc. Here, ν is the greatest fixpoint operator, defined as $\nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$.

1.2 SEMANTICS

Matching logic formulae have a pattern matching semantics. Each pattern φ *matches* a set of elements $|\varphi|$ in the model, called its interpretation. As an example, consider the naturals $\mathbb{N}$ as a model with symbols zero and succ. Here, the pattern $\top$ matches every natural, whereas succ(x) matches $x+1$. Conjunctions and disjunctions behave as intersections and unions—the $\varphi \vee \psi$ matches every pattern that either φ or ψ match.

Unlike first-order logic, matching logic makes no distinction between terms and formulae. We may write $\mathrm{succ}(x \vee y)$ to match both $x + 1$ and $y + 1$. While unintuitive at first, this syntactic flexibility allows us to shallowly embed varied and diverse logics in matching logic with ease. Examples include first-order logic, temporal logics, separation logic, and many more [6], [7]. Formulae are embedded as patterns with little to no *representational distance*, quite often verbatim.

Patterns are not two valued as in first-order logic. We can restore the classic semantics by using the set M to indicate "true" and $\emptyset$ for "false". The operators $=$ and $\subseteq$ are *predicate patterns*—they are either true or false. For example, $x \subseteq \mathrm{succ}(\top)$ matches every natural if x is non-zero, and no element otherwise. This allows us to build *constrained patterns* of the form $\varphi_{\text{structure}} \wedge \varphi_{\text{constraints}}$. Here, $\varphi_{\text{structure}}$ defines the structure, while $\varphi_{\text{constraints}}$ places logical constraints on matched elements. For example, the pattern $x \wedge (x \subseteq \mathrm{succ}(\top))$ matches x, but only if it is the successor of some element—i.e. it is not zero.

Existential quantification works just as in first-order logic when working over predicate patterns. Over more general patterns, it behaves as the union over a set comprehension. For example, the pattern $\exists x.\, x \wedge (x \subseteq \mathrm{succ}(\top))$ matches *every* non-zero natural. Finally, the fixpoint operator allows us to inductively build sets, as in algebraic datatypes or inductive functions. For example, the pattern $\mu X.\, \mathrm{zero} \vee \mathrm{succ}(\mathrm{succ}(X))$ defines the set of even numbers.

1.3 PROOF SYSTEM

The third component to matching logic is its proof system, shown in Figure 1.1. It defines the provability relation, written $\Gamma \vdash \varphi$, meaning that φ can be proved with the proof system using the theory Γ as additional axioms. These proof rules fall into four categories: (1) *First-order reasoning:* the FOL rules provide complete FOL and propositional reasoning. (2) *Frame reasoning:* The (PROPAGATION) rules allow applications to commute through constructs with a "union" semantics, such as disjunction and existentials. (3) *Fixpoint reasoning:* The proof rule (KNASTER-TARSKI) is an embodiment of the Knaster-Tarski fixpoint theorem [8], and together with (PREFIXEDPOINT) corresponds to the Park induction rules of modal logic [9], [10]. (4) *Technical rules:* Finally, (EXISTENCE), (SINGLETON), and (SUBST) are technical rules, needed to work with variables.

(PROPOS$_1$)	$\varphi \to (\psi \to \varphi)$	(PROPAG$_\bot$)	$C[\bot] \to \bot$
(PROPOS$_2$)	$(\varphi \to (\psi \to \theta))$ $\to ((\varphi \to \psi) \to (\varphi \to \theta))$	(PROPAG$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
		(PROPAG$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ where $x \notin FV(C)$
(PROPOS$_3$)	$((\varphi \to \bot) \to \bot) \to \varphi$		
(MP)	$\dfrac{\varphi \qquad \varphi \to \psi}{\psi}$	(FRAMING)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
($\exists$-QUANT.)	$\varphi[y/x] \to \exists x.\, \varphi$	($\exists$-GEN.)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}$ where $x \notin FV(\psi)$
(PRE-FP)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$	(KT)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
(EXISTENCE)	$\exists x.\, x$	(SUBST)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(SINGLETON)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$		

Figure 1.1: Matching logic proof system. Here C, C_1, C_2 are application contexts, a pattern in which a distinguished element variable $\Box$ occurs exactly once, and only under applications. We use the notation $C[\varphi] \equiv C[\varphi/\Box]$.

1.4 NATURALS IN MATCHING LOGIC

In this document, we will use the natural numbers as a running example. For completeness let us describe its axiomatization in matching logic, Γ_{Nat}.

Signature: zero, succ(_).

$$\exists x.\, x = \text{zero} \quad (\text{FUNC}_0) \qquad \forall x.\, \text{succ}(x) \neq \text{zero} \quad (\text{NO-CONF}_{0\text{-}s})$$

$$\forall x.\, \exists y.\, x = \text{succ}(y) \quad (\text{FUNC}_s) \qquad \forall x, y.\, \text{succ}(x) = \text{succ}(y) \to x = y \quad (\text{NO-CONF}_s)$$

$$\mu X.\, \text{zero} \vee \text{succ}(X) \quad (\text{DOMAIN})$$

These axioms are not unlike Peano's axioms for the naturals. The axioms (FUNC$_0$), (FUNC$_s$)—corresponding to the first and sixth Peano axioms—give these symbols a *functional* interpretation. That is, they are single valued. These aren't needed in the first-order axiomatization of the naturals because in first-order logic, terms are defined to be interpreted as single-valued. The axioms (NO-CONF$_{0\text{-}s}$) and (NO-CONF$_s$) state that we should not "confuse" constructors—they are injective functions. These are similar to the eighth and seventh Peano axioms. Finally, (DOMAIN) is similar in spirit to Peano's ninth axiom, the Axiom of Induction. Note that it is more powerful than the first-order analog of this axiom,

and does not admit non-standard models. By Gödel's incompleteness theorem, this means that matching logic cannot be complete—there are valid statements that cannot be proved.

1.5 LINEAR TEMPORAL LOGIC

Here, we will review (infinite-trace) linear temporal logic (LTL).

Definition 1.2. Let A be a set of atomic propositions. Then, the syntax of LTL is given using the following grammar.

$$\varphi ::= a \in A \mid \bot \mid \varphi_1 \to \varphi_2$$
$$\mid \circ\varphi \mid \varphi_1 \, U \, \varphi_2$$

Other operators can be defined as syntactic sugar:

$$\top \equiv \bot \to \bot \qquad \Diamond\varphi \equiv \top \, U \, \varphi \qquad \Box\varphi \equiv \neg\Diamond\neg\varphi$$

The models of LTL are infinite traces over $\mathcal{P}(A)$—that is, infinite lists of elements labeled by a set of atomic propositions. We use $\pi = \varphi_0\varphi_1\varphi_2\ldots$ to denote a trace, where each $\varphi_i \subseteq A$. We use $\pi_{\geq i} \equiv \pi_i\pi_{i+1}\pi_{i+2}\ldots$ to denote a suffix.

Definition 1.3. The satisfaction relation $\pi \vDash_{\mathsf{LTL}} \varphi$ is defined as follows:

- $\pi \vDash_{\mathsf{LTL}} p$ iff $p \in \pi_0$
- $\pi \vDash_{\mathsf{LTL}} \bot$ does not hold.
- $\pi \vDash_{\mathsf{LTL}} \varphi_1 \to \varphi_2$ holds iff either $\pi \vDash_{\mathsf{LTL}} \varphi_1$ does not hold or, $\pi \vDash_{\mathsf{LTL}} \varphi_2$ holds
- $\pi \vDash_{\mathsf{LTL}} \circ\varphi$ iff $\pi_{\geq 1} \vDash_{\mathsf{LTL}} \varphi$
- $\pi \vDash_{\mathsf{LTL}} \varphi_1 \, U \, \varphi_2$ iff there exists a $j \geq 0$ such that $\pi_{\geq j} \vDash_{\mathsf{LTL}} \varphi_2$, and for every $i < j$, $\pi_{\geq i} \vDash_{\mathsf{LTL}} \varphi_1$.

1.5.1 Linear Temporal Logic in Matching Logic

An embedding of linear temporal logic over finite-traces was described in [11]. We present it below:

Signature: $\bullet\_$

Notation:

$$\circ\varphi \equiv \neg\bullet(\neg\varphi) \qquad \text{(NEXT)}$$
$$\Diamond\varphi \equiv \mu X.\, \varphi \vee \bullet X \qquad \text{(EVENTUALLY)}$$
$$\Box\varphi \equiv \nu X.\, \varphi \wedge \circ X \qquad \text{(ALWAYS)}$$
$$\varphi\, U\, \psi \equiv \mu X.\, \psi \vee (\varphi \wedge \bullet X) \qquad \text{(STRONG-UNTIL)}$$

Axioms:

$$\bullet\top \qquad \text{(INFINITE)}$$
$$\bullet\varphi \to \circ\varphi \qquad \text{(LINEAR)}$$

Here, the symbol *one-path next* $\bullet$, is the dual to *all-path next* $\circ$ defined as notation. In LTL, since traces are linear, these two concepts coincide however, they may diverge in say CTL*. For consistency we define both here. Eventually $\Diamond$, always $\Box$ and until U are defined as syntactic sugar using matching logic's fixpoint operators. The axiom (INFINITE) forces all states to have at least one successor, and thus all traces can be extended to infinite ones; (LINEAR) states that if some property holds for one next state, it must hold for all next states, effectively forcing a linear trace.

1.6 SEPARATION LOGIC

Separation Logic[12] (SL) is a logic for reasoning about programs with heaps. It is an extension of Hoare logic [13]. We will primarily deal with the static portion of separation logic, a specialization of bunched logic[14], This lets us specify assertions over *heaps*—finite maps between addresses and values. We will treat this component of separation logic with an extension, inductive definitions, as synonymous with the whole.

Definition 1.4. Let X be a set of variables, and P be an arity-indexed set of symbols called recursive predicates. The syntax of separation logic is given by the following grammar:

$$\mathsf{term} := x \in X \mid \mathsf{nil}$$
$$\varphi := \mathsf{emp} \mid x \mapsto \mathsf{term} \mid \varphi_1 * \varphi_2 \mid \varphi_1 \mathrel{-\!\!*} \varphi_2 \mid p(t_1, \ldots, t_n) \text{ for } p \in P_n$$

Each predicate p in P_n has a definition of the form $p(x_1, \ldots, x_n) =_{\mathsf{lfp}} \varphi$, where $\mathrm{free}(\varphi) \subseteq \bar{x}$, and p occurs positively. Informally, each formula φ is satisfied by a set of heaps or finite maps

from addresses to values. The formula emp denotes the empty heap, $x \mapsto t$ denotes a heap where the address x points to the term t. The separating conjunction denotes a heap that is the juxtaposition of two heaps with *disjoint* address spaces. The separating implication, also known as the magic wand operator $-\!\!*$ denotes the heaps that when conjuncted (via $*$) with φ_1 satisfy φ_2. Heap structures, such as list segments may be defined as a recursive predicate:

$$\mathsf{ll}(x, y) =_{\mathsf{lfp}} (x = y \wedge \mathsf{emp}) \vee (\exists t.\, x \mapsto t * \mathsf{ll}(t, y) \wedge x \neq y)$$

1.6.1 Separation Logic in Matching Logic

Let us recall the partial formalization of separation logic in matching logic in [15]. While this formalization is partial, it is enough for our application in Section 1.6.

Sorts: Heap, $\mathsf{Address}$, Value

Signature:

$$\mathsf{emp} : \mathsf{Heap}$$
$$\_ * \_ : \mathsf{Heap} \times \mathsf{Heap} \to \mathsf{Heap}, \text{ and}$$
$$\_ \mapsto \_ : \mathsf{Heap} \times \mathsf{Heap} \to \mathsf{Heap}$$

Axioms:

$$\forall h_1, h_2, h_3.\, h_1 * (h_2 * h_3) = (h_1 * h_2) * h_3 \qquad (\text{ASSOC}_*)$$
$$h_1 * h_2 = h_2 * h_1 \qquad (\text{COMM}_*)$$
$$\mathsf{emp} * h = h \qquad (\text{ID}_l)$$
$$a \mapsto v \to a \neq \mathsf{nil} \qquad (\text{ADDRESS-NON-NIL})$$
$$a_1 \mapsto v_1 * a_2 \mapsto v_2 \to a_1 \neq a_2 \qquad (\text{ADDRESS-DISTINCT})$$

Part II

Practicality of $\mathbb{K}$ for defining Programming Languages

Here, we will evaluate the $\mathbb{K}$ framework as a tool for developing programming languages. Is the additional cost of abstraction—via a formal mathematical formalization—prohibitive? Is it practical to develop a such a formalization? Does $\mathbb{K}$ offer the practical methods of abstraction?

We will tackle these questions through the implementation of two languages as $\mathbb{K}$ definitions. The first, the Etherium Virtual Machine, or EVM[16], is a stack machine, used as part of the Etherium blockchain[17].

The second, is the Boogie Intermediate Verification language (Boogie IVR)[18], an intermediate language used in the building verification infrastructure for other programming languages (such as Dafny[19], C[20], and more). This language is quite different from languages previously implemented in $\mathbb{K}$ since it requires evaluation of expressions with universal quantifiers, and verification of loop invariants and function contracts.

For both these, we will describe our implementations, evaluate them against test suites (taking advantage of the fact that $\mathbb{K}$ definitions are executable). We will discuss some of the pluses and minuses of our approach. These sections will also serve as a introduction to $\mathbb{K}$ by-example.

CHAPTER 2: SEMANTICS OF THE ETHEREUM VIRTUAL MACHINE

This project was joint work with Everett Hildenbrandt et al. Here, we focus on the areas of the project that I was involved in—for a more detailed description of the work please refer to [21]. It was completed in 2019, and so the evaluation and related work sections refer to that time.

Ethereum is a blockchain or "cryptocurrency". Quite informally, it is a cryptographically backed public ledger mapping user accounts to account balances, in a currency called Ether. Transactions between accounts allow the exchange of Ether, and are governed by small computer programs, or "smart contracts" written in a quasi-Turing complete language called the Ethereum Virtual Machine or EVM. These contracts may be used to facilitate a huge variety of contracts: from escrow services, decentralized currency exchanges[22], multi-signature authorization[23], gambling[24], and dark web markets[25].

The contracts are enforced by having their execution performed and verified through a consensus forming among the users of the blockchain, and thus do not require a trusted authority backing them. Assuming that the consensus protocol is sound, the code implementing the contract is the final word and transactions may not be disputed.

Millions of dollars worth of Ether may be in the trust of these programs, and since transactions may be initiated by any member of the public, they are under heavy scrutiny. Bugs in such contracts can be financially devastating to the involved parties[26]. An example of such a catastrophe is the DAO attack [27], where 150 million USD worth of Ether was stolen, prompting an unprecedented hard fork of the Ethereum blockchain [28]. Worse still, the DAO is one of many smart contracts which did not execute as expected, inducing costly losses [29], [30], [31], [32]. A few of these are documented in Table 2, with many more in [26].

Table 2.1: Smart contract failures impacting $\geq$ 400k USD. Stars* indicate implementations of the ERC20 API [33]. (Ether price data from https://coinmarketcap.com.)

Contract name	**Value**	**Root cause**
Parity Multisig 1 [31]	$200M	Private function exposure
Parity Multisig 2 [32]	$165M	Private function exposure
The DAO* [27]	$150M	Re-entrancy
SmartBillions [34]	$500K	Broken caching mechanism
HackerGold (HKG)* [35]	$400K	Typo in code

To address these issues in a principled manner, we make use of the $\mathbb{K}$ Framework [36], [37] to formally specify EVM, and use its verification tools to check the correctness of smart contracts. Since these are small, deterministic programs with large financial stakes (often Ether worth in excess of 100M USD) and exploits are irreversible—one cannot dispute a transaction. Further, all contract code is public, and attackers can probe the system with full knowledge, testing and refining their attack privately before deploying it publicly.

2.1 THE ETHEREUM VIRTUAL MACHINE

The EVM is a stack machine. For the most part instructions operate over a stack of *words* and may additionally use empheral local memory. Additional instructions may interact with the network state for example, transfering Ether or mutating a global persistent memory.

```
 0: PUSH(0); PUSH(10);                 // s = 0; n = 10
 4: JUMPDEST;                          // loop:
 5: DUP(1); ISZERO; PUSH(21); JUMPI;   //   if n == 0: goto end;
 9: DUP(1); SWAP(2); ADD; SWAP(1);     //   s = s + n;
13: PUSH(1); SWAP(1); SUB;             //   n = n - 1
17: PUSH(4); JUMP;                     //   goto loop;
21: JUMPDEST;                          // end:
22: POP; PUSH(0); SSTORE;              //   store to network
```

Figure 2.1: A simple EVM program that adds the numbers from zero to ten. The left column indicates the program location, and comments indicate an equivalent program written in an imperative style.

In Figure 2.1 we see a simple EVM program for adding numbers from one to ten. The `PUSH` instuction simply pushes a value to the stack; `JUMPDEST` is a label, and acts as a target for unconditional jumps `JUMP`, and conditional jumps `JUMPI`. Each of these consumes an integer from the stack and jumps to that program location. `DUP(n)` duplicates the nth item from the stack and places it at the top, while `SWAP(n)` swaps it with the top; `ISZERO` checks if the top of the stack is zero, and replaces it with an integer indicating true or false. `ADD` and `SUB` implement integer arithmetic, modulo 2^{256}; finally, `SSTORE` stores the result to persistent network storage.

2.2 $\mathbb{K}$ SEMANTICS OF EVM

$\mathbb{K}$ represents program execution state using a *configuration*. The configuration is an unordered list of (potentially nested) *cells*, specified in using an XML-like notation.

When declaring transitions (as rewrites) over this state, any subset of the cells present in the configuration can be mentioned. This allows the user to specify only the necessary parts of the state for a given transition, letting $\mathbb{K}$ assume that the remaining parts of the configuration remain unchanged.

The KEVM configuration is split into two components: that of an active VM (for executing transactions and contracts), and the state of the network as a whole (such as, account information). We omit the full configuration, which contains 70+ cells.

2.2.1 VM state

Execution of EVM programs must maintain the executing account (`<id>` cell), the current program counter (`<pc>` cell), and the current program (as a map from program counters to opcodes in the `<program>` cell). The `<wordStack>` and `<localMem>` cells provide memory in the form of a bounded wordstack and scratchpad RAM, respectively. The `<gas>` cell maintains how much longer execution can continue before the VM forcibly terminates the program (to avoid DoS attacks). The comments to the right of the cells indicate the names used in the Yellow Paper[16] for these components of the state.

```
configuration
 <k> $PGM:EthereumSimulation </k>
 <evm>
  <id>         0           </id>         // I_a
  <program>    .Map        </program>    // I_b
  <pc>         0           </pc>         // \mu_pc
  <wordStack>  .WordStack  </wordStack>  // \mu_s
  <localMem>   .Map        </localMem>   // \mu_m
  <gas>        0           </gas>        // \mu_g
  ...
 </evm>
 ...
```

When declaring a configuration (using the keyword), the initial values are supplied. $\mathbb{K}$ replaces the placeholder with the code for the program being run, usually specified as a command line parameter or input file. In the case of the test suite, this is a JSON object describing the network state at the time of execution, the code for the program being executed, and a set of post conditions expected at the end of execution. Traditionally, the cell is used to drive execution and holds the next execution step of a $\mathbb{K}$ semantics.

2.2.2 Network state

The Ethereum blockchain forms a log of transactions on the network, which when replayed lead to the current world/network state. In our semantics, we choose to store the current world/network state over the append-only log of transactions leading to this state. In a given state, there may be any number of active accounts and pending transactions. Here we show part of the `<network>` sub-configuration, specifically the portion corresponding to account states.

```
configuration
  ...
  <network>
    <activeAccounts> .Map </activeAccounts>
    <accounts>
      <account multiplicity="*" type="Map">
        <acctID>  0          </acctID>
        <balance> 0          </balance>
        <code>    .WordStack </code>
        <storage> .Map       </storage>
        <nonce>   0          </nonce>
      </account>
    </accounts>
    ...
  </network>
```

The `<accounts>` cell holds information about the accounts on the blockchain. Each `<account>` holds the accounts associated `<balance>`, `<code>` (smart contract), `<storage>` (persistent memory), and `<nonce>`[1]. By adding attribute `multiplicity="*"`, we state that 0 or more `<account>` cells can exist at a time (that is, multiple accounts can exist at a time on the network). As an optimization, we additionally state that accounts can be treated internally as a map from their `<acctID>` (by specifying `type="Map"` on the `<account>` cell and listing `<acctID>` as the first sub-cell) This adds the extra requirement that any access of an `<account>` cell in a rule *must* mention the corresponding `<acctID>` cell.

2.2.3 Execution

Exception-based control-flow Exceptions are part of the low-level control-flow of the EVM—they may occur in case of invalid opcodes, `JUMP`s to PCs that haven't been marked as

[1]This nonce is a globally accessible monotonically increasing integer value that counts the number of transactions performed by this account.

`JUMPDEST`s, if there is insufficient gas to pay for execution, or in case of stack over/under-flow.

We built a simple imperative language for throwing/catching exceptions in KEVM. Exceptions consume anything following them on the `<k>` cell until they are caught.

```
    syntax KItem      ::= Exception
    syntax Exception ::= "#exception" | "#end"
 // ----------------------------------------
    rule <k> EX:Exception ~> (_:Int    => .) ... </k>
    rule <k> EX:Exception ~> (_:OpCode => .) ... </k>
```

The operator `~>` ships with distributions of $\mathbb{K}$ and acts as an associative binary sequencing operation (read as "followed by", and similar to the semicolon in many imperative languages).

Here, we show how exceptions consume any following `Int` or `OpCode` by rewriting them to the empty computation (`.`) (which is the empty/identity element of the operator `~>`). These rules can be read as '`when something of sort`Exception`is at the front of the`cell, `it dissolves anything of sort`l scope here is *local*: matching happens on the entire rule but the state change only happens inside the parentheses.

To use exceptions for control flow, we provide a branching choice operator `#?_:_?#` which chooses the first branch when no exception is thrown and the second when one is:

```
    syntax KItem ::= "#?" K ":" K "?#"
 // ----------------------------------
    rule                #? B1 : _  ?# => B1
    rule #exception ~> #? _  : B2 ?# => B2
```

The anonymous variable (`_`) is used to tell K that we do not care about the a subterm value when matching or rewriting.

Execution Cycle Execution in KEVM is driven by the internal operator `#next`, which loads and triggers execution of the next opcode. As described in section 9.4 of the Yellow Paper[16], execution of a single opcode follows these steps:

1. Perform quick checks for exceptional opcodes.
2. Execute the opcode if the checks passed.
3. Increment the program counter.
4. Revert state in case of any exceptions.

Here is the $\mathbb{K}$ rule which gives semantics to the `#next` operator, performing the above steps:

```
rule <k> #next
      => #pushCallStack ~> #exceptional? [ OP ]
```

```
                        ~> #exec          [ OP ]
                        ~> #pc            [ OP ]
      ~> #? #dropCallStack : #popCallStack ?#
    ...
    </k>
    <pc> PCOUNT </pc>
    <program> ... PCOUNT |-> OP ... </program>
```

Note that this rule reaches across multiple cells in the configuration (including the `<k>`, `<pc>`, and `<program>` cells). The ellipsis (..., called *structural frames*) in these rules are not omission of details, but syntax supported by $\mathbb{K}$ for abstracting uninteresting parts of the state. We match on the current program counter `PCOUNT` along with the corresponding key/value pair anywhere in the program map to retrieve the next opcode `OP`. The operator `_|->_` is $\mathbb{K}$'s built-in map-binding operator, which creates one key/value pair of the `Map` sort. Another rule in the semantics handles the case where the `PCOUNT` key is not present in the `<program>` map indicating that the program has run to termination.

Upon successfully finding the next opcode, first the current execution state is saved with internal operator `#pushCallStack`. Then, steps 1, 2, and 3 from above are performed using internal operators `#exceptional?`, `#exec`, and `#pc`. If an exception is thrown at any point during this, it consumes everything up to the choice operator `#?_:_?#` and takes the second branch which reverts the execution state with a call to `#popCallStack`. Given that no exception is thrown, the saved-off state is instead forgotten with a call to `#dropCallStack` (to save memory).

2.2.4 Example OpCodes

Opcodes are declared with sort corresponding to their arity to simplify the process of loading arguments from the `<wordStack>`. For example, `BinStackOp`s consume two `Word`s from the `<wordStack>`.

```
    syntax BinStackOp ::= "ADD" | "SUB"
 // ------------------------------------
    rule ADD W0 W1 => W0 +Word W1 ~> #push
    rule SUB W0 W1 => W0 -Word W1 ~> #push
```

`SUB` and `DIV` perform simple arithmetic on their arguments then use internal operator `#push` to push the result onto the `<wordStack>`. In contrast, the opcodes `SLOAD` and `SSTORE` access the current accounts `<storage>`. In both cases, the current account `ACCT` is matched so that the appropriate `<account>` cell is selected for matching.

Here, `SLOAD` grabs a single word at position `I` from the `<storage>` and `#push`es it onto the wordstack. A second rule (omitted here) specifies the behavior when the index `I` does not exist in the current account storage.

```
    syntax UnStackOp ::= "SLOAD"
 // ----------------------------
    rule <k> SLOAD I => V ~> #push ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> ... I |-> V ... </storage>
           ...
         </account>
```

`SSTORE` is used to write value `V` to index `I` in the current account `<storage>`. Notice that here, rewrite arrows are present in three cells: `<k>`, `<storage>`, and `<refund>`. This notational convenience allows users to specify rules more compactly, without having to duplicate parts of the configuration that remain unchanged on both the left and right-hand sides of the rule. The updates to state in each of the cells happens simultaneously, *only if* the overall left-hand side matches and the `requires` clause (if present) is met.

```
syntax BinStackOp ::= "SSTORE"
 // ------------------------------
    rule <k> SSTORE I V => . ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage>
           ... I |-> (OLD => V) ...
           </storage>
           ...
         </account>
         <refund> R =>
           #ifInt OLD =/=Int 0 andBool V ==Int 0
            #then R +Word Rsstoreclear < S >
            #else R
           #fi
         </refund>
         <schedule> S </schedule>
```

Here, a quirk of the EVM is also demonstrated with the `<refund>` cell. If the `OLD` value in `<storage>` is non-zero but the new value `V` is zero, the current executing account is refunded

some gas for freeing up memory. Notice that the refund amount, `Rsstoreclear`, is parametric over `S` (the current fee `<schedule>`) as explained further in Section 2.2.5.

2.2.5 Gas Semantics

Each execution step and memory expansion in EVM costs some state-dependent amount of gas, which ensures that all computations are terminating. KEVM mimics the Yellow Paper's gas calculation by providing several gas helper functions defined in Appendix G in [16]. For example, the function `Csstore` calculates the gas needed to store a value to an account's storage:

```
  syntax Int
    ::= Csstore (Schedule, Int, Int) [function]
//-------------------------------------------
  rule Csstore(SCHED, V, OLD)
    => #ifInt V =/=Int 0 andBool OLD ==Int 0
          #then Gsstoreset   < SCHED >
          #else Gsstorereset < SCHED >
       #fi
```

Note that the cost of storing to an account's memory depends on whether you are setting it for the first time (before it was zero, now it's not) or not. Beyond that, each gas-cost is parametric over a *fee schedule*. As Ethereum has evolved the fees for each opcode have been tweaked to disincentivize behavior expensive to the network and incentivize alternatives. This means that the same computation may consume different amounts of gas depending on when (in which block of the blockchain) it is executed. Since the blockchain requires that all past transactions be replayable, all EVM implementations must be aware of all previous schedules and not just the one currently in use. We abstract this information into *schedules*. Above, 'Gsstoreset' and 'Gsstorereset' are parametric over a 'Schedule'; the function `_<_>` allows making a schedule constant parametric over a schedule.

```
syntax Int
  ::= ScheduleConst "<" Schedule ">" [function]
//--------------------------------------------
```

Here we show some example fee schedules and schedule constants:

```
    syntax ScheduleConst
      ::= "Gzero"    | "Gbase"  | ...
        | "Gbalance" | "Gsload" | ...
 // --------------------------------
```

```
    syntax Schedule ::= "DEFAULT"
// ----------------------------
    rule Gzero < DEFAULT > => 0
    rule Gbase < DEFAULT > => 2

    syntax Schedule ::= "EIP150"
// ---------------------------
    rule Gbalance < EIP150 > => 400
    rule Gsload   < EIP150 > => 200
```

The fee schedule to use is set through the command line flag `-cSCHEDULE=<FEE_SCHEDULE>`. This allows us to execute and verify programs against any appropriate fee schedule.

2.3 EVALUATION

2.3.1 Correctness and Performance

As consensus-critical software, implementations of EVM are held to a high standard; past disagreements have caused accidental forks of the blockchain which leads to disparate world-views [38]. We based our semantics on the Yellow Paper [16], but found inconsistencies confirmed by its developers.

Table 2.2: Lem EVM vs KEVM vs cpp-ethereum

Test Set (no. tests)	**Lem EVM**	**KEVM**	**cpp-ethereum**
Lem (40665)	288:39	34:23	3:06
VMStress (18)	-	72:31	2:25
VMNormal (40665)	-	27:10	2:17
VMAll (40683)	-	99:41	4:42
GSNormal(22705)	-	35:00	1:30
GSQuad (250)	-	855:24	0:21
GSAll (22955)	-	889:00	1:51

Being an executable semantics, we are able to test our semantics against the exensive EVM test suite provided with the C++ reference implementation. Table 2.3.1 shows a performance comparison between KEVM, the Lem semantics [39], and the C++ reference implementation

distributed by the Ethereum foundation. The Lem semantics (discussed more in Section 2.8) is the only other executable formal specification of the EVM we are aware of.

All execution times are given as the full sequential CPU time (in MM:SS format) on an Intel i5-3330 processor (3GHz on 4 hardware threads) and 24 GB of RAM. By comparing to the C++ reference implementation, we show the feasibility of using the KEVM formal semantics for prototyping, development, and test-suite evaluation.

The row Lem indicates a run of all the tests that the Lem semantics can run (a subset of the VMTests suite). The row VMStress indicates a run of all 18 stress tests in the test-suite, to compare the performance of KEVM with the C++ implementation. The row VMNormal is a run of all the non-stress tests in the test-suite (*not* the same set of tests as Lem). VMAll is the addition of the second and third rows and is included for completeness. The last three rows indicate a runs of the GeneralStateTests; GSNormal are the non-stress tests, GSQuad are the stress tests, and GSAll is the addition of the two. Under the GeneralStateTests, our tools performs well except in the case of QuadraticStateTests (250 out of 22955).

As shown in the comparison, the automatically extracted interpreter for KEVM outperforms the currently available formal executable EVM semantics. KEVM compares favorably to the C++ implementation, performing under 30 times slower on the stress tests, roughly 20 times slower on all tests, and only 11 times slower on the Lem and VMNormal tests.

2.4 IMPLEMENTATION EFFORT

The time to develop the first release-quality KEVM was roughly 2 months of light activity by 2 developers followed by 3 months of heavy activity by 4 developers.

The total count of non-blank and non-literate lines of code for KEVM comes in at 2644. For comparison, the reference C++ implementation weighs in at 4588 lines of code. This measurement was taken on commit-hash ee0c6776c of https://github.com/ethereum/cpp-ethereum by counting non-blank lines of all and files in subdirectory .

We argue that these numbers are not atypical for implementing an interpreter for a small real-world programming language, not to mention the analysis tools and security provided along the way.

2.5 VERIFICATION OF SMART CONTRACTS

A primary motivation for this work has been to mitigate security failures as listed in the Chapter's introduction. Many such issues can be addressed via verification, i.e. proving a program conforms to a formal property. As mentioned previously, $\mathbb{K}$ generates a deductive verifier. We briefly demonstrate verification of a simple EVM program. Since the completion of this work many real-world smart contracts have been verified. See [40].

2.6 JELLO PAPER

While developing these semantics, a common problem we faced was interpreting the Yellow Paper, the English language specification of the EVM [16]. Often times, the Yellow Paper is unclear or underspecified, and in some exceptional cases even unfaithful to what actual implementations do.

For example, Section 9.4.2 of [16] describes exceptions as if they are all detectable prior to opcode execution. While it may be possible to implement EVM in this way, it is not clear that this is the simplest or best way (as it would lead to duplicating computation). No implementations seem to work this way, casting further doubt on this description; instead exceptions are thrown when they happen. Our original implementation tried to do it in this way, eventually necessitating a redesign.

In other cases, the expected behavior is underspecified. For example, it is not always explicit about what should happen when an opcode attempts to access a non-existent account's data. Another example is the appearance of "junk bytes" in a program's bytecode (which do not correspond to any opcodes); these can be used for loading long immutable strings of data into the VM. Though not originally addressed in the Yellow Paper, the community has reached agreement on these issues.

In some places the Yellow Paper is even inaccurate. The instruction `DELEGATECALL`, with semantics given in Appendix H of [16], describes the gas provided to the caller as equal to $\mu_s[0]$ (the top of the `<wordStack>`). This is clearly incorrect, since $\mu_s[0]$ is a user-provided value, and the user could set it equal to $2^{256} - 1$, leading to the user having near infinite gas. The test suites and other implementations indicate that the intended behavior is to use C_{callgas}, but with the *value*-transferred argument set to 0. For the same opcode, it describes the exceptional condition of not enough balance in terms of I_v, but in fact no value transfer occurs so this condition should never occur.

In the process of building an executable specification, all of these issues naturally arose when testing against the test-suite, as they did for other implementations. These problems make implementing tools and infrastructure for the Ethereum ecosystem needlessly error-prone and inefficient. Instead, we propose using our "developer" documentation, which is automatically generated from the KEVM semantics. This version of the semantics, called the Jello Paper, is available at https://jellopaper.org. We hope to continue improving the Jello Paper readability, and have been in communication with members of the Ethereum Foundation regarding establishing it as a reference specification for the EVM platform and an executable successor to [16].

2.7 GAS ANALYSIS TOOL

EVM programs are forced to always terminate to prevent malicious actors from mounting a DoS attack on the network. This is done by allotting gas for execution ahead of time and charging each VM operation some gas. If gas is exhausted before execution finishes, an exception is thrown and the state is reverted. For many contracts, functional correctness is dependent upon enough gas being supplied up front. To help users decide how much gas they should supply, we extended the semantics with a gas analysis tool.

The semantics was already designed with extensibility in mind; execution is parametric over an extra `<mode>` cell which controls how to interpret EVM programs. For example, in mode `VMTESTS`, execution of `CALL` and `CREATE` opcodes is not performed, as specified by the Ethereum Test Suite [41].

```
configuration ...
              <mode> $MODE:Mode </mode>
              ...

syntax Mode ::= "NORMAL" | "VMTESTS" | ...
```

The extension adds the execution mode `GASANALYZE`, that modifies how the `#next` operator works. In `GASANALYZE` mode, the `#next` operator executes normally until it hits a control-flow operator (eg. `JUMP`, `JUMPI`, or `JUMPDEST`), collecting the overall gas consumed to do so. At the control-flow operator, the overall gas consumed is recorded in the `<analysis>` cell (along with the starting and ending program counter for that basic block). Finally, the program counter is forcibly incremented past the control-flow operator, and the analysis is restarted.

In this way, each basic block is executed in isolation and the amount of gas used is collected and reported back to the user in the cell. Note that the current implementation only calculates

an approximation, but some engineering effort would result in a more accurate calculation.

This extension is a 1.8% increase in the size of the semantics (87 lines of literate code, roughly a day of work), demonstrating the flexibility of having a directly extendable executable specification of the EVM.

2.8 COMPARISON WITH RELATED WORK

There has been substantial practical interest in formally verifying properties of smart contracts. For example, the Solidity IDE incorporates Why3 [42] (a semi-automated theorem prover) to help verify smart contracts written in the higher-level Solidity language. In this section, we compare the practical artifacts derived from and generated by our work to existing efforts. A list compiled by Dr. Yoichi Hirai informs our comparison. We do not include or compare with any tools which operate over other languages (e.g., Solidity source analysis tools) exclusively, serve as implementations of an Ethereum network client, or are closed (eg. Securify).

The tools produced in the Ethereum community are meant to fill a variety of purposes, many of which are also able to be accomplished directly from our executable semantics. Table 2.8 shows an overview of the results of our comparison. We briefly describe each effort and compare it to the relevant KEVM artifact. The projects fit two categories: semantic specifications and smart contract analysis tools.

Table 2.3: Feature comparison of EVM semantics and other software quality tool efforts. **Spec.:** Suitable as a formal specification? **Exec.:** Executable on concrete tests? **Tests:** Passes the Ethereum test-suites? **Prover:** Serves as theorem prover for EVM? **Bugs:** Heuristic-based tools for finding bugs? **Gas:** Analyzes gas complexity of EVM programs?

Tool	Spec.	Exec.	Tests	Prover	Bugs	Gas
Yellow Paper	✓	✗	✗	✗	✗	✗
cpp-ethereum	✗	✓	✓	✗	✗	✗
Lem spec	✓	✓	✓	✓	✗	✗
Oyente	✗	✓	✗	✗	✓	✓
hevm	✗	✓	✗	✗	✗	✗
Manticore	✗	✓	✗	✗	✓	✓
REMIX	✗	✓	✗	✗	✓	✓
Dr. Y's	✗	✓	✗	✓	✗	✗
F *	✗	✓	✗	✓	✓	✗
KEVM	✓	✓	✓	✓	✗	✓

Tool	Spec.	Exec.	Tests	Prover	Bugs	Gas

2.8.1 Semantic Specifications

Yellow Paper [16] The official document describing the execution of the EVM, as well as other data, algorithms, and parameters required to build consensus-compatible EVM clients and Ethereum implementations. It cannot be tested against the conformance test-suite; instead it serves as a guide for implementations to follow. Much of the machine definition is supplied as several mutually recursive functions from machine-state to machine-state. The Yellow Paper is occasionally unclear or incomplete about the exact operational behavior of the EVM; in these cases it is often easier to simply consult one of the executable implementations.

cpp-ethereum A C++ implementation that also serves as a de-facto semantics of the EVM. The Yellow Paper and the C++ implementation were developed by the same group early in the project, so the Yellow Paper conforms mostly to the C++ implementation. In addition, the conformance test-suite is generated from the C++ implementation. This means that if the Yellow Paper and the C++ implementation disagree, the C++ implementation is favored.

Lem semantics [39] A Lem ([43]) implementation of EVM provides an executable semantics of EVM for doing formal verification of smart contracts. Lem compiles to various interactive theorem provers, including Coq, Isabelle/HOL, and HOL4. The Lem semantics does not capture intercontract execution precisely as it models function calls as non-deterministic events with an external (speculated) relation dictating the "allowed non-determinism". This semantics is executable and passes all of the VMTests test-suite except for those dealing with more complicated intercontract execution, providing high levels of confidence in its correctness.

GMS small-step specification [44] A small-step specification of the EVM inspired by the EtherLite semantics of [45]. The specification is non-executable, but provides a precise guide for implementers of the EVM.

2.8.2 Smart Contract Analysis Tools

Oyente An EVM symbolic execution engine written in Python supporting most of the EVM. Many heuristics-based drivers of the engine are provided for bugfinding.

Manticore [46] A symbolic execution engine for virtual machines, including models for x86, x86_64, ARMv7, and EVM. This tool exports a Python API for specifying programs, driving symbolic execution, and checking assertions.

REMIX A JavaScript implementation of the EVM with a browser-based IDE for building and debugging smart contracts. Some static analysis is built into the tool, allowing it to catch pre-specified classes of bugs in smart contracts.

Dr.Y's Ethereum Contract Analyzer A symbolic execution engine for EVM to summarize the semantics of smart contracts. A debug mode allows step-by-step execution.

F * formalization of EVM [47] An implementation of the EVM in the F⋆language which passes roughly half of the VMTests at the time of writing. The same paper discusses an on-paper small-step specification of the EVM as well [44].

2.9 FUTURE WORK

We believe this rich ecosystem of tools, all generated programmatically from a single independent reference semantics, has the potential to be transformative in the development and deployment of secure smart contracts while avoiding a large class of potential losses and failures. With our existing semantics and EVM interpreter, we plan on finishing the work required to encourage the widespread adoption of our work as the reference semantics and interpreter for the EVM system.

Future EVM Hardforks As demonstrated in section 3.5, the current semantics is parametric over the selected fee schedule. The EVM will continue to evolve as new opcodes are added and gas prices are changed, and the specification will need to evolve with it. KEVM provides a solid tool for prototyping and testing updates to the EVM specification, allowing for a smoother protocol update process.

ABI-Level DSL Currently our ABI-level DSL only supports statically-sized ABI types, but the ABI contains several dynamically sized types as well. We plan to extend our DSL to support dynamic types too, allowing writing specifications over any ABI-compliant contract. Once the full ABI is supported, we will have produced an executable specification of the EVM ABI abstraction.

Verified EVM Libraries To further assist others in building high-assurance contracts, we intend to provide fully verified library contracts. Over time, we'll collect a body of high-assurance EVM code for the community.

2.10 CONCLUSION

In this section, we presented a formalization of the EVM using the Framework. This provides a specification of the EVM, a reference interpreter, and a suite of tools for program analysis and verification. KEVM is the first executable specification of the EVM that completely passes official test-suites.

Not only is our semantics complete and faithful, but performant. This is an important point if the semantics is intended to be used in a Continuous Integration environment where every change should be tested thoroughly. Changes to the EVM can realistically be prototyped on KEVM, yielding both an updated specification and implementation.

Beyond being a specification of the EVM, KEVM serves as a platform for building a wide range of analysis tools and other semantic extensions for EVM. We demonstrated this with three extensions: a full web-based version of our semantics, a gas analysis tool, and verification. Each of the extensions leveraged the existing semantics, meaning each required minimal code changes.

Already, KEVM has proved to be a useful tool for software developers working on smart contracts. Beyond that, KEVM is a valuable resource to Ethereum developers experimenting with evolving the EVM and protocol updates, as it streamlines the process of bringing an implementation in line with the specification for experimentation purposes. We hope this serves to signal the practicality of an *executable specification first* approach to programming language design, as well as the merits of separating the construction of programming language semantics from analysis tools. We believe the application of these tools can drastically increase the rigor and security of both currently deployed and future smart contracts.

CHAPTER 3: SEMANTICS OF BOOGIE

Boogie[18] is an intermediate verification language (IVL). It plays a similar role for program verifiers as LLVM[48] does in compilers. Similar to how compilers for a language may be built via translation to LLVM, program verifiers may be built via translation to Boogie.

Several frontends exist for Boogie. These include translations from Dafny[19], C (VCC[49] and HAVOC[50]), and Java (Joogie), GPUVerify (a verifier for GPU kernels), Spec# (a formal language for API contracts), among others. Dafny forms the core of larger verification efforts, such as IronFleet[51] and IronClad[52]. The IronClad project includes a verified kernel, verified drivers, verified system and crypto libraries including SHA, HMAC, and RSA; and applications built using these. There are also several backends. Besides the Microsoft's original Boogie verifier, Boogaloo[53] and Symbooglix[54] (symbolic execution engines), Corral[55] (a whole-program analysis tool), and Whoop[56] (a symbolic data race analyzer) all use the Boogie IVL as their input. Therefore, it is clear that the Boogie IVL forms the keystone in a large verification effort.

Yet, despite Boogie playing this foundational role, it does not have a formal semantics. All these tools depend on having a single notion of what a Boogie program means. The most complete semantics, "This is Boogie 2"[57], is now over a decade old and has not been updated as the Boogie IVL has evolved. Several new features (e.g. `lambda`s) have since been introduced, and other language features (such as `where` clauses and quantifiers) were glossed over. Being a natural language document, it is not always precise and unambiguous.

A second motivation for this semantics is that we believe that this "translate to an intemediate language" methodology for developing formal verification tools is not sustainable and error prone—the semantics-first approach is the "morally correct" way to go. We feel that the best way to demonstrate that this approach could work is to formally define the semantics of Boogie in $\mathbb{K}$, as opposed to providing formal semantics for each of the languages defined using Boogie.

The goals of this work are thus two-fold:

1. to provide a formal semantics to the Boogie IVL, and
2. to demonstrate that verification languages, just like other programming languages, may be given a formal semantics using the semantics-first approach.

3.1 THE BOOGIE IVL

Boogie is designed to be the translation target of verification conditions for imperative, object-oriented programs. It presents some unique languages paradigms that are not seen in other languages. These include `assume` statements (that allow adding arbitary path conditions), quantification, symbolic variable states, infinite maps and uninterpreted functions. These set Boogie apart from languages previously defined in $\mathbb{K}$.

For example, Boogie's `assume` statement allows adding constraints to a program's path condition without otherwise affecting the state. Looping constructs must be executed until the program state reaches a fixed point, and then stop. This involves keeping track of program points at which such looping may occur and the state at those points. See Section 3.6 for a broader discussion regarding this.

Another instance of this is that Boogie allows users to quantify over logical variables. At first glance, it may seem that these may simply be translated to SMT. However, the quantified expressions typically are a mix of logical constructs and program constructs (such as program variables)—they rely on program state, and must be reduced to a completely logical expressions before being sent to a solver.

In Figure 3.1, we show a example Boogie program. For the most part, Boogie programs look like typical imperative programs, including global and local variables and procedures. Each procedure specifies a constract, through its type signature, pre- and post-conditions as well as a list of global variables it may possibly modify. Unlike other imperative languages, procedures may have multiple implementations, each adhering to the specified contract.

Implementations may use imperative-style statements for assignments, control flow structures such as `if` and `while` statements, function calls. It also includes some unusual features such as symbolic values, non-deterministic `goto` statements, `assume` statements, quantification and more. We will describe the details of each of these constructs as we describe their semantics in $\mathbb{K}$ in the main body of this paper.

3.2 BOOGIE SEMANTICS

As mentioned in the previous chapter, $\mathbb{K}$ definitions have three major components: a grammar defining the syntax of the programming language via `syntax` declarations, forming a BNF grammar; `configuration` statements describing possible program state as a record-like structure; and a set of transition `rules` describing transitions between these states.

```
// Declare a constant 'K' and a function 'f' and postulate that 'K' is in the image of
const K: int;
function f(int) returns (int);
axiom (exists k: int :: f(k) == K);

// This procedure will find a domain value 'k' that 'f' maps to 'K'. It will
// do that by recursively enlarging the range where no such domain value exists.
// Note, Boogie does not prove termination.
procedure Find(a: int, b: int) returns (k: int)
  requires a <= b;
  requires (forall j: int :: a < j && j < b ==> f(j) != K);
  ensures f(k) == K;
{
  goto A, B, C;  // nondeterministically choose one of these 3 goto targets

  A:
    assume f(a) == K;  // assume we get here only if 'f' maps 'a' to 'K'
    k := a;
    return;

  B:
    assume f(b) == K;  // assume we get here only if 'f' maps 'b' to 'K'
    k := b;
    return;

  C:
    assume f(a) != K && f(b) != K;  // neither of the two above
    call k := Find(a-1, b+1);
    return;
}
```

Figure 3.1: An example Boogie program from https://github.com/boogie-org/boogie/blob/50ef2fb38f0c8/Test/textbook/Find.bpl

3.2.1 Imperative Core

To begin with we well describe Boogie's imperative core, focusing on how it differs from typical languages.

`assert` statements The `assert` statement prescribes a check on the current state. It takes a Boolean expression as argument. If the expression evaluates to `false` the program execution halts with failure, otherwise the statement becomes a no-op. We use $\mathbb{K}$'s `context` statement to ensure that the expression passed in is fully evaluated to a `Bool` value. This is done through $\mathbb{K}$'s heating and cooling mechanism.

```
syntax SimpleStmt ::= "assert" Expr ";"
context assert HOLE ; [result(Bool)]
rule <k> assert true ; => .K ... </k>
rule <k> (.K => #failure) ~> assert false; ... </k>
```

*`assume` statements** The `assume` construct is allows adding arbitrary constraints to the path condition. It allows us to make assumptions about the program state that may be used for the rest of the execution. States where these assumptions do not hold are ignored from future execution.

```
syntax SimpleStmt ::= "assume" Expr ";"
context assume HOLE ;
rule <k> assume true ; => .K      ... </k>
rule <k> assume false; => #Bottom ... </k>
```

As with `assert`, when the passed Boolean expression is `true` the statement evaluates to a no-op. The second `rule` states that when the expression evaluates to `false`, `assume` evaluates to `#Bottom`. `#Bottom` corresponds to matching logic's $\bot$ pattern. Semantically, this means that we no longer care about this particular trace in the execution—that is, this particular trace succeeds.

havoc statements This statement allows setting variables to an undefined value, possibly constrianted be a where clause. For example the following procedure simulates rolling two six-sided die, returning the sum of their faces.

```
procedure rollTwoDice() returns (r: int) {
    var d : int where d >= 1 && d <= 6;
    r = d;
    havoc d;
    r = r + d;
}
```

In our semantics, we implement this by assigning a fresh symbolic value to the variable, and then assuming the where clause. The construct `fresh(Type)` returns a fresh symbolic value subject to the constraints of that type.

```
rule <k> havoc X ; => X := fresh(type(X)) ~> assume where(X) ; ... </k>
```

call statements Calling a procedure is implemented very differently from typical languages. We may not assume any particular procedure implementation but only that the result returned adheres to the procedures contract. This can be implemented quite simply by first asserting that the pre-conditions stated in the procedures `requires` clause is met, replacing the variables with fresh value, similar to `#havoc`, and finally, `assume` that the post-conditions (via the `ensures` clause) are met.

```
context _ call X:IdList := ProcedureName:Id(HOLE);
rule <k> call X:IdList := ProcedureName:Id(ArgVals);
      => assert(lambda Args :: Requires)[ArgVals];
      ~> #freshen(X ++ Modifies)
      ~> assume(lambda (Args ++ Rets)::Ensures)[ArgVals ++ X];
         ...
     </k>
     <procName> ProcedureName </procName>
     <args> Args </args> <rets> Rets </rets>
     <requires> Requires </requires> <ensures> Ensures </ensures>
     <modifies> Modifies </modifies>
  requires isKResult(ArgVals)
```

`old` **expressions** In `ensures` clauses, it is possible to refer to the original value of the variable before the implementation is executed. This enables asserting that some relation holds between the old and the new value. This is exhibited in the following program.

```
procedure P() modifies x; ensures x == old(x) + 1; {
  start:
    x := 1 + x;
}
```

This is implemented using the follow rule, that replaces the global variable environment with the original one for the duration of the evaluation of the expression.

```
rule <k> old(E) => E ~> #restoreGlobals(Globals) ... </k>
     <globals> Globals => Olds </globals>
     <olds> Olds </olds>
```

`goto` statements Besides function calls, all of Boogies control flow statements are defined in terms of `goto`. Boogie's `goto` statement is non-deterministic, allowing execution to jump to any label from a list. This is implemented quite simply using two non-deterministic rules.

```
syntax Stmt ::= "goto" IdList ";"
rule <k> (goto L, Ls ; ~> _) => Stmts </k> <labels> L |-> Stmts ... </labels>
rule <k> goto L, Ls ; => goto Ls ; ... </k> requires Ls =/=K .IdList
```

3.2.2 Checking Loop Invariants

Boogie's `goto` statements complicate the checking of loop invariants. It may lead to complex looping behaviours that involve multiple labels, without being defined by explicit control flow structures.

To detect this looping behaviour, introduce a construct called a cutpoint. Each `cutpoint` represents a point in the programs execution where looping may occur. Each cutpoint is associated with an identifier and an invariant. The first time a trace reaches a cutpoint, we check that the invariant holds at that point, make a note that we have reached the cutpoint in the `<cutpoints>` cell. Then, we "forget" all dynamic program state besides that the invariant holds, similar to `havoc`, by replacing variables with unconstrained symbolic values via the `#freshen` construct. The next time the trace returns to the cutpoint, we first check that the program state is subsumed by the invariant using an `assert`ion. Now, since the invariant still holds, we know that this particular path maintains the invaraint.

```
rule <k> cutpoint(I, E) ;
      => assert(E) ;
      ~> #freshen(envToIds(Rho) ++ Modifiable)
      ~> assume(E) ;
         ...
     </k>
     <locals> Rho </locals>
     <modifies> Modifiable </modifies>
     <cutpoints> (.List => ListItem(I))
                 Cutpoints
     </cutpoints>
  requires notBool I in Cutpoints

rule <k> cutpoint(I, E) ; => assert(E) ; ~> assume (false); ... </k>
     <cutpoints> Cutpoints </cutpoints>
  requires I in Cutpoints
```

This is an approximation of Hoare logic's reasoning for while loops.

3.2.3 Quantification

Quantifiers in Boogie are a unique language feature. The interplay between logical and program variables and constructs gives them an interesting semantics. We treat existential quantification as the dual of universal quantification, and so will only talk about the latter in detail.

In an ideal world, we'd like to implement universal quantifiers using reachability claims: for each model, if on all paths the quantified expression reaches `true` then the quantifier evaluates to `true`; otherwise if on any path the quantified expression evaluates to `false` then the quantifier evaluates to `false`, where `=>*` symbolizes all-path reachability—the left-hand side reaches the right-hand side along all paths, and for all possible concrete instantiations of symbolic values:

```
rule <k> (forall X : T :: E ) => true ... </k>
  requires          <k> E[X/fresh(T)] =>* true <k>
rule <k> (forall X : T :: E ) => false ... </k>
  requires notBool(<k> E[X/fresh(T)] =>* true <k>)
```

However, $\mathbb{K}$ does not allow using statements about reachability in side-conditions. Indeed, such a feature would be impossible to implement—in general, it would require $\mathbb{K}$ to guess the right circularities to complete the proof. We may however, in the case of Boogie, make a simplifying assumption: all execution traces of the expression `E` terminate. This is because Boogie does not allow any looping constructs in the these expressions. Further, in calling a procedure we do not need to execute it's implementation but may instead jump over it by assuming it obeys it's contract.

With nested quantifiers in mind, the algorithm for evaluating these expressions is as follows: We execute the expression in the quantifier, in the current configuration replacing the previous contents of the `<k>` cell. Since the configuration is symbolic, it may reduce to a disjunction of configurations whose path conditions cover the original. Each disjunct may take one of the two forms: (1) it may encounter a nested quantifier, in which case, we recurse; (2) it may reduce to a boolean expression—`true`, `false` or a symbolic boolean value. In this case, we conjunct the result with the path condition and return it.

```
procedure execForall(config, X, Expr):
  results = symbolicExec(setKCell(config, E))
  resultPattern = true
```

```
  while(!empty(results)):
    result = pop(results)
    if (isBooleanExpression(head(kcell(config)))):
      resultPattern = resultPattern $\land$
         (pathCondition(config) $\limplies$
           head(kcell(config)))
    if (matches( kcell(config)
                , (forall X :: Expr) ~> Rest)):
      forallResult = execForall(config, X, Expr)
      results += symbolicExec(
        setKCell(config, forallResult ~> Rest))
  return resultExpression
```

3.3 OBSERVATIONS

We have observed a few surprising behaviours in Boogie programs, that cannot be given a trace-based semantics—a semantics by assuming each symbolic program state corresponds to a set of concrete programs. For example, consider the following:

```
procedure Q () {
  var x : int where x == 5;
  x := 6;
  start:

    goto start, end;
  end:
    assert x == 5;
    // fails as expected
}
```

```
procedure P () {
  var x : int where x == 5;
  x := 6;
  start:
    x := x;
    goto start, end;
  end:
    assert x == 5;
    // succeeds!
}
```

At first glance they appear to be semantically identical. However, the final assertion is triggered in one version, but not in the other. This is despite, the fact that when considering the semantics of individual statements, all traces result in the final assertion failing, and none in them succeeding. This behaviour of `where` clauses is not the result of the semantics of any individual statement, but a property of the global program structure. It is likely that Boogie uses static analysis to implement this behaviour, checking for updates to variables between cutpoints and applying `havoc` to them if so. This behaviour is difficult to justify in terms of execution traces, and so difficult to justify in a verification language where our goal

is to prove properties of these execution traces. `while` loops stop being the inductive control structures they are normally considered to be, and what it means for a program to be verified itself comes into question. We feel that this is not a desirable property for a verification language and decided to not support it. Other tools, such as Boogaloo, that use the Boogie IVL as their basis also make this choice. Symbooglix does not implement `where` clauses at all.

3.4 EVALUATION

Test file	Features tested	№ impls	№∀	Time taken
Arrays.bpl	maps, quantifiers, old exprs	16	8	26m44s
AssumeEnsures.bpl	assume, (free) ensures	9	0	0m52s
Axioms.bpl	quantification	3	1	1m20s
B.bpl	maps	4	0	1m06s
BadLineNumber.bpl		2	0	0m29s
Call.bpl	call	8	0	1m17s
ContractEvaluationOrder.bpl	ensures, requires	4	0	0m59s
CutBackEdge.bpl	assert and goto	6	0	1m07s
Ensures.bpl	maps, ensures	12	1	6m25s
False.bpl		2	0	0m34s
FormulaTerm.bpl	arithmetic & boolean exprs	15	0	3m32s
FormulaTerm2.bpl	functions	5	0	1m00s
FreeCall.bpl	free call	13	0	1m33s
IfThenElse1.bpl	if then else	4	0	1m00s
Lambda.bpl	lambda	8	11	43m44s
LoopInvAssume.bpl	cutpoints	1	0	0m33s
Old.bpl	old expressions	11	0	3m11s
Passification.bpl	control flow, maps	11	0	6m35s
TypeEncodingM.bpl	maps	1	0	0m39s

Figure 3.2: The tests run to evaluate our semantics, including the major features tested, number of implementations, number of quantifiers and time taken.

Our $\mathbb{K}$ specification consists of 1992 lines of literate $\mathbb{K}$ code and 270 semantic rules, compared with the 31,537 lines of C# code in the `Source/Core` directory of Boogie and 4811 lines of Haskell for Boogaloo, providing a concise and unambiguous semantics of the Boogie IVL.

As mentioned previously, one key advantage of using $\mathbb{K}$ is that it provides an executable formal semantics. We may use the generated interpreter to validate the specification against Boogie's regression tests. In Figure 3.2, we present a list of tests and the features of the semantics that they exercise. We run our formal specification against tests in the `test2/` directory of Boogie's test suite. These tests cover all the core features of Boogie. Note that since our implementation does not provide a type checker or perform any well-formedness

checking, we only run our implementation against tests that are well-formed. Further, there are a number of attributes that we do not yet support. Some, such as `{:verified_under ... }`, `{:selective_checking ... }` aren't documented. Others, such as `{:timelimit ... }`, aren't interesting to the core semantics of the prover. We have also as of yet not implemented polymorphic types and triggers (See future work).

Each of the run tests include a number of procedures and implementations, and are expected to produce a set of failing assertions. We check that the line numbers of the failures produced by Boogie are the same as by our semantics.

Most tests run in a few minutes. The two outliers, `Arrays.bpl` and `Lambda.bpl`, have a large number of quantifiers and implementations. For the most part the time taken is proportional to the number of procedures. However, the tests that use quantification are much slower. This is because the $\mathbb{K}$ backend does not understand quantifiers natively, and is forced to call into the SMT solver every time it takes a step. Further, when we use Boogie's invariant inferencer to pre-process the tests, it also converts `lambda` expressions into functions constrained by quantified expressions with triggers. Even so our semantics is able to execute these tests, albeit slowly, by ignoring the triggers. One possible way of mitigating this penalty would be to cache the results of SMT queries and counter examples as KLEE does[58].

3.5 RELATED WORK

3.5.1 This is Boogie 2

As mentioned previously, "This is Boogie 2" [57] is a natural language reference for the Boogie IVL. This document was quite useful in the definition of this semantics. Its main failings are that it is not sufficiently precise due it it being a natural language document and that it glosses over key parts of the language such as quantification and cutpoints. It has also not been kept up to date with respect to new features of the language.

3.5.2 Symbolic Execution of Boogie Programs

Boogaloo[53] is a symbolic execution engine for the Boogie language. In order to implement this engine, a formal (but non-executable) paper semantics for the Boogie IVL was defined in the form of a set of proof-rules. These rules were then used as a reference to implement a symbolic execution engine in Haskell. Symbooglix[54] is another symbolic execution engine for Boogie programs that supports a subset of Boogie syntax (e.g. it does not implement

`where` clauses).

3.5.3 Boogie-To-Why3

Boogie-To-Why3[59] provides a translation from Boogie to Why3, another IVL. This translation is useful in giving us an alternate implementation for the Boogie IVL allowing us to cross-check verification results. It also allows us to use the different backends provided by Why3, such as Alter-Ego, CVC4 and Z3 and to take advantage of Why3's interactive prover when automatic proofs fail. This translation does not, however, give us a formal semantics. It also does not encode core features of Boogie such as `goto` into Why3. More importantly, Why3, like Boogie is an IVL without a formal basis.

3.6 FUTURE WORK

There are several avenues for building on this work that we may take.

3.6.1 Completing the semantics of Boogie

There are some additional features we need to complete the operational semantics of Boogie. The most difficult of these are polymorphism and triggers. Completing these features would enable us to run programs translated from other languages, such as Dafny, unchanged. This would give us more confidence in our semantics. Running programs translated from C using the SMACK frontend also need support for bit vectors, that we haven't added yet.

3.6.1.1 Polymorphism

Boogies functions, maps, quantifiers, and procedures may be polymorphic – i.e. they may apply over a set of types rather than any particular type. However, this requires us to implement a procedure for type inference.

3.6.1.2 Triggers

Implementing triggers cleanly depends on functionality that does not as of yet exist in $\mathbb{K}$. Triggers restricts how quantified expressions are instantiated. For example, the statement `assume (forall x : int { f (x) } E )` should be instantiated only if `f(x)` is encountered sometime during execution. For example, in Figure 3.3, in the procedure `Trigger1` the final

assertion fails, despite that `g(3)` would evaluate to `true` if the `assume` statement did not include triggers.

Boogie implements this via translation to Z3. This cannot be done implemented in $\mathbb{K}$ by pattern matching, because, in truth, the semantics of triggers is "do whatever the SMT solver does". The `Trigger2` procedure demonstrates this. Even though the assumption is not triggered, it is used by the SMT solver to complete verification.

Unfortunately $\mathbb{K}$ does not yet provide an interface to specify triggers. The $\mathbb{K}$ team is considering adding this feature, and once such an interface is available, we plan on implementing triggers in KBoogie.

```
function f(int) : int ;
function g(int) : bool ;

procedure Triggers1() {
    assume (forall x : int :: { f(x) } g(x) );
    assert g(3);
    // ^^^ Assertion fails becase f(x) has not been
    //     triggered.
}

procedure Triggers2() {
    assume (forall x : int :: { f(x) } false );
    // ^^^ This assumtion is executed despite
    //     not being triggered. It is equivalent
    //     to `assume false;` and execution stops.
    assert false;
    // ^^^ Assertion not reached,
    //     and program verifies.
}
```

Figure 3.3: Triggers in Boogie

3.6.2 Invariant inference

Currently, we use Boogie to infer invariants for programs, before running them using KBoogie. It would be useful to implement this as a part of KBoogie. This can be done by changing the semantics of cutpoints so that each time the program reaches a particular cutpoint we record the current state. We take the disjunct of the current state with all previous states recorded for that cutpoint and apply an abstraction function. Using an abstraction lattice is of finite height, this process will eventually reach a fixed point and terminate.

3.6.3 Derive verifiers directly from the language semantics

Currently, program verifiers are implemented via translation to Boogie. This is a non-trivial effort. For example, Dafny's translation to Boogie in `Source/Dafny/Translator.cs` is 18852 lines of code. In comparison, it's compiler to C# (in `Source/Dafny/Compiler.cs` and `Source/Dafny/Compiler-Csharp.cs`) is 6577 lines of code. Implementing the translation to Boogie is clearly a non-trivial task, likely more complex than implementing the language itself. This should not be surprising, since both these translations do similar things – define the semantics of a Dafny in terms of a lower level language. Since each translation is independent, and essentially gives Dafny two different semantics. This is fertile ground for discrepancies and behaviour that differs between these two implementation.

$\mathbb{K}$ already provides a verifier, called `kprove`, for all language semantics. However, this prover is not as user-friendly as Dafny, VCC and other verifiers derived via Boogie. This is because `kprove` does not let users define invariants using the language's native syntax. They must instead use $\mathbb{K}$'s syntax. This means that the users must be aware not only of the programming language they are working in but also of $\mathbb{K}$, and the idiosyncrasies of the language's semantics in $\mathbb{K}$ as well.

In this work we provided justification for `cutpoint` and `call` through their correspondence with the proof rules of reachability logic. Instead of implementing `cutpoint` as a manual instantiation of these rules, `kprove` may directly apply these proof rules. `kprove` does not currently do this because it would require it to guess invariants. Creating a mechanism within $\mathbb{K}$ for sending hints to the prover from semantics would work around this, allowing us to derive much more user-friendly verification tools directly from the language semantics.

Part III

Certified Fixpoint Reasoning

CHAPTER 4: FIXPOINT REASONING IN MATCHING LOGIC

Fixpoint reasoning is of great practical interest to program verification and reasoning. Everything from a program's grammar, to the set of reachable states, to the inductive datatypes and functions it employs are defined in terms of fixpoints. Indeed, $\mathbb{K}$ draws from varied formalizations that employ fixpoints in the specification and reasoning of programming languages. These include order-sorted algebras, temporal logics, reachability logic, and separation logic to name a few. This makes understanding fixpoint reasoning critical to certifying implementations of $\mathbb{K}$'s tools.

Thus far, fixpoint reasoning in each of these domains has been implemented using domain-specific solvers. Besides the redundancy of effort, these isolated implementations make combined reasoning, as needed by $\mathbb{K}$, complex. $\mathbb{K}$ relies on being able to mix and match these techniques, using those most suited to a task. For example, when model checking a program, we may need fixpoint reasoning for both reachability analysis as well as for reasoning about inductive definitions.

Matching logic is uniquely situated to be the basis of a *unified* proof framework. It is a powerful and expressive logic, and many logical systems and programming languages have been defined as theories. Its base proof system includes components for fixpoint reasoning, in addition to standard FOL reasoning, domain reasoning, frame, and context reasoning.

We envision a *unified ecosystem* for proof reasoning with matching logic as its basis. Higher-level proof rules may be proved as lemmas on top of matching logic's proof system. These, along with embeddings of various formalisms will serve as the basis of automated provers. A simple theorem prover may be implemented as a search over a fixed set of these building blocks. A more complex one, may use a powerful decision procedure to guide this search.

Figure 4.1 illustrates this vision: a set of reasoning modules along with embeddings of the various logical systems are used as building blocks for prover. Modules developed for one domain may easily be reused in others, making extending the architecture to new domains easier. At the same time it enables formal reasoning across multiple domains. For example, to reason about a programs behaviour we may need to reason both about heaps and transition systems.

Let us discuss these reasoning modules in detail.

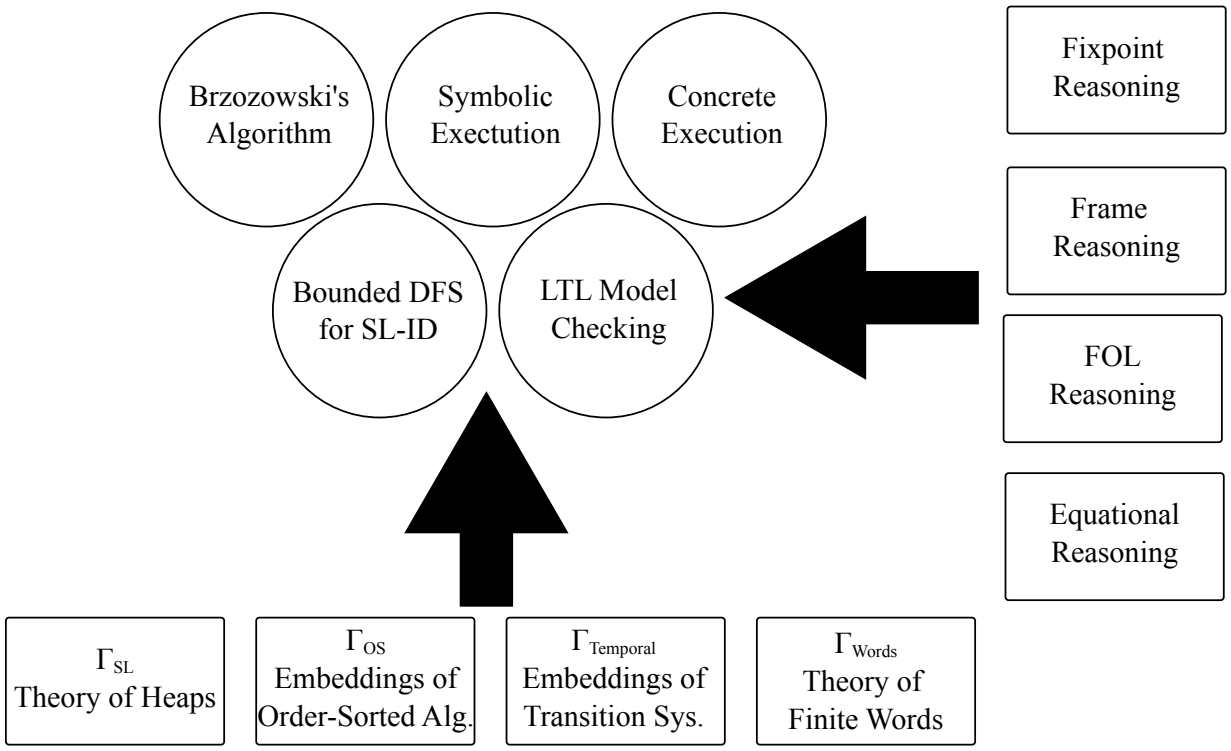


Figure 4.1: Our vision for a unified proof framework for program reasoning.

4.1 REASONING MODULES

The main challenge we faced while developing our framework was that the existing proof system of matching logic[60] is too fine-grained to be amenable for automation. For example, its (MODUS PONENS) proof rule "$\vdash \varphi \to \psi$ and $\vdash \varphi$ implies $\vdash \psi$" requires the prover to guess a premise φ, which does not bode well with automation.

Another example is the (KNASTER-TARSKI) proof rule (inspired by Park induction[61]) for fixpoint reasoning

$$\text{(KNASTER-TARSKI)} \quad \frac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$$

is limited to handling the cases where the LHS of the proof goal is a standalone least fixpoint. It cannot be directly applied to proof goals such as $\mathsf{list}(x, y) * \mathsf{list}(y) \to \mathsf{list}(x)$, where the LHS $C[\mathsf{list}(x, y)]$ contains the fixpoint $\mathsf{list}(x, y)$ within a context $C[\Box] \equiv \Box * \mathsf{list}(y)$. An indirect application is possible *in theory*, but it involves sophisticated, ad-hoc reasoning to eliminate the context C from the LHS, which cannot be efficiently automated.

4.2 SIMPLE FIXPOINT REASONING

Before diving into the higher-level fixpoint reasoning rules, let us first see take a closer look at the lower-level rules and see how they may be employed. Typically, fixpoints are defined in matching logic as a disjunction of clauses under the least fixpoint binder μ. For example, the naturals may be defined as $\text{Nat} \equiv \mu X.\, \text{zero} \lor \text{succ}(X)$. This is analogous to the typical definition of a recursive datatype `Nat = zero | succ(Nat)` in a programming language. Other examples include even numbers, $\text{even} \equiv \mu X.\, \text{zero} \lor \text{succ}(\text{succ}(X))$, and odd numbers, $\text{odd} \equiv \mu X.\, \text{succ}(\text{zero}) \lor \text{succ}(\text{succ}(X))$.

Together, the (KNASTER-TARSKI) and (PRE-FIXPOINT) rules form the basis of fixpoint reasoning in matching logic and allow us to reason about least fixpoint binders.

Informally, the (PRE-FIXPOINT) rule $\vdash \varphi[\mu X.\, \varphi/X] \to \mu X.\, \varphi$ is a logical embodiment of the fact that every least-fixpoint is a pre-fixpoint. It is useful for unfolding μ binders. For example, we may use it to prove:

Example 4.1. $\Gamma_{\text{Nat}} \vdash \text{succ}(\text{succ}(\text{even})) \to \text{even}$

Proof.

1.	$\text{zero} \lor \text{succ}(\text{succ}(\text{even})) \to \text{even}$	(PRE-FIXPOINT)
2.	$\text{succ}(\text{succ}(\text{even})) \to \text{even}$	Notation

QED.

The (KNASTER-TARSKI) rule is the logical incarnation of the Knaster-Tarski theorem[8]. Below, we use it to prove the induction principle for the naturals, also called the *axiom of induction* in the context of the Peano axioms. Informally, it may be stated as "if zero is in a set K, and for every natural in K its successor is also in K, then every natural is in K."

Example 4.2.

$$\frac{\Gamma_{\text{Nat}} \vdash \text{zero} \to \varphi \qquad \Gamma_{\text{Nat}} \vdash \text{succ}(\varphi) \to \varphi}{\Gamma_{\text{Nat}} \vdash \text{Nat} \to \varphi}$$

Proof.

1.	$\text{zero} \to \varphi$	Assumption
2.	$\text{succ}(\varphi) \to \varphi$	Assumption
3.	$\text{zero} \lor \text{succ}(\varphi) \to \varphi$	FOL Reasoning, 1, 2
4.	$\mu X.\, \text{zero} \lor \text{succ}(X) \to \varphi$	(KNASTER-TARSKI)
5.	$\text{Nat} \to \varphi$	Notation

QED.

4.3 FIXPOINTS WITHIN CONTEXTS

The (KNASTER-TARSKI) proof rule can only be applied when the left-hand side of the goal is a least fixpoint. For example, it is not of much help when trying to prove the following goal, where the + symbol is at the top of the pattern:

$$\Gamma_{\text{Nat}} \vdash \text{even} + \text{even} \to \text{even}$$

Here, the left-hand side is a larger pattern in which a fixpoint pattern even occurs. That is, even occurs within a *context*. Let $C[\Box] \equiv \Box + \text{even}$ be the *context pattern* in which $\Box$ is a distinguished element variable. We may rewrite the goal above as $C[\text{even}] \to \text{even}$. Clearly, (KNASTER-TARSKI) can only be applied when C is the *identity context*, i.e., $C_{id}[\Box] \equiv \Box$, but as we have seen, in practice recursive patterns often occur within a non-identity context. So a major challenge in applying (KNASTER-TARSKI) in automated fixpoint reasoning is to handle such non-identity contexts in a systematic way. The contextual reasoning module allows us extract a fixpoint from within a context, apply the necessary fixpoint reasoning, and plug it back in. This is enabled by the *contextual implication* operator, written $C \multimap \psi$, where C is a structural context defined below. Informally, it is a pattern that matches all elements that when plugged into context C match ψ. Formally, it is defined as the following notation:

$$C \multimap \psi \equiv \exists \Box.\, \Box \wedge (C \subseteq \psi)$$

To solve this challenge, we propose an important concept called *contextual implication*. In the following, we first give a formal definition of context patterns and contextual implications and then revisit the example.

Definition 4.1. A *context* C is a pattern with a distinguished variable denoted $\Box$, called the hole variable. We write $C[\varphi]$ as the substitution $C[\varphi/\Box]$. Given an ML theory Γ, we say that C is a *structural context* w.r.t. Γ if $C \equiv t \wedge \psi$ where t is a structure pattern and ψ is a predicate, and $\Box$ occurs exactly once in t within nested symbols (and not other logical constructs). All contexts considered in this chapter are structure contexts.

In other words, a context C is a structural context if the hole variable $\Box$ occurs only within nested applications. For example, $(\Box + \text{succ}(y)) \wedge y > 1$ is a structural context (w.r.t. $\Box$) because + is a matching logic symbol.

Let $C[\Box]$ be a structural context, and ψ be some property. We define *contextual implication* w.r.t. C and ψ as the pattern whose matching elements, if plugged into C, satisfy ψ. Formally,

Definition 4.2. We define *contextual implication* $C \multimap \psi \equiv \exists \Box.\, \Box \wedge (C[\Box] \subseteq \psi)$.

Recall from Chapter 1, that the semantics of $\exists$ is set union. Thus, $C \multimap \psi$ is the pattern matched by all elements $\Box$ such that $C[\Box] \subseteq \psi$ holds, i.e., by all elements that when plugged into C the result, $C[\Box]$, satisfies the property ψ. This is embodied by the following lemma:

Lemma 4.1.

$$\vdash C[C \multimap \psi] \to \psi \ (\textsc{ctximp-collapse})$$

A structural context C is *extensive* in the hole position. An element a matches $C[\varphi]$ where C if and only if there exists an element a_0 that matches φ and a matches $C[a_0]$—i.e. a_0 matches $C \multimap \varphi$. This is quite concisely captured by the introduction and elimination rules for contextual implication, that allow us to pull a pattern out of a context, and plug it back in.

Lemma 4.2. Let C be an application context, then

$$\frac{\vdash \varphi \to C \multimap \psi}{\vdash C[\varphi] \to \psi}\ (\textsc{ctximp-elim}) \qquad \frac{\vdash C[\varphi] \to \psi}{\vdash \varphi \to C \multimap \psi}\ (\textsc{ctximp-intro})$$

Now, let us revisit our example and prove it using these rules:

Example 4.3.

$$\Gamma_{\mathrm{Nat}} \vdash \mathrm{even} + \mathrm{even} \to \mathrm{even}$$

Proof.

1.	$\text{zero} + \text{even} \to \text{even}$	Domain reasoning
2.	$\text{zero} \to (\Box + \text{even} \multimap \text{even})$	(CTXIMP-INTRO)
3.	$(\Box + \text{even} \multimap \text{even}) + \text{even} \to \text{even}$	(CTXIMP-COLLAPSE)
4.	$\text{succ}(\text{succ}((\Box + \text{even} \multimap \text{even}) + \text{even})) \to \text{succ}(\text{succ}(\text{even}))$	(FRAME)
5.	$\text{succ}(\text{succ}((\Box + \text{even} \multimap \text{even}) + \text{even})) \to \text{even}$	By Ex. 4.1, trans. of $\to$
6.	$\text{succ}(\text{succ}((\Box + \text{even} \multimap \text{even}))) + \text{even} \to \text{even}$	By Ex. 4.1, ($\text{PROP}_\vee$)
7.	$\text{succ}(\text{succ}((\Box + \text{even} \multimap \text{even}))) \to (\Box + \text{even} \multimap \text{even})$	(CTXIMP-ELIM)
8.	$\text{zero} \vee \text{succ}(\text{succ}((\Box + \text{even} \multimap \text{even}))) \to (\Box + \text{even} \multimap \text{even})$	FOL, (2), (7)
9.	$(\mu X.\, \text{zero} \vee \text{succ}(\text{succ}(X))) \to (\Box + \text{even} \multimap \text{even})$	(KNASTER-TARSKI)
10.	$\text{even} \to (\Box + \text{even} \multimap \text{even})$	Notation
11.	$\text{even} + \text{even} \to \text{even}$	(CTXIMP-ELIM)

QED.

Here, the (FRAME) rule refers to matching logics's proof rule, that allows removing a wrapping context when it is not necessary.

$$(\text{FRAME})\frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$$

4.3.1 Fixpoints and Recursive Symbols

Although it is easy to define recursive *constant* symbols using just the μ binder, parametric definitions that take arguments are more complex. Consider the function

$$\begin{aligned}\text{plus}(x, y) := \ &\texttt{case} : (0, y) \Rightarrow y \\ &\mid \texttt{case} : (\text{succ}(t), y) \Rightarrow \text{succ}(\text{plus}(t, y))\end{aligned}$$

We may not define this merely using the fixpoint binder, because the pattern $\text{plus}(x, y)$ as a whole is not an recursive definition—we do not inductively build $\text{plus}(x, y)$ in terms of subsets of $\text{plus}(x, y)$ but rather in terms of plus called with other arguments. More precisely, the graph of the relation plus is recursively defined. So, as detailed in [11], it becomes natural to define the graph of plus as a 3-tuple, like so:

$$\text{plus} \equiv (\mu X.\, (\exists y.\, \langle \text{zero}, y, y\rangle) \vee (\exists t, y.\, \langle \text{succ}(t), y, \text{succ}(X(t, y))\rangle))$$

The first two arguments of each 3-tuple correspond to the arguments x and y, while the third corresponds to the result of relation. Here, the base cases, $\exists y.\, \langle \text{zero}, y, y\rangle$, give us the set of all triples with 0 as the first element and indentical integers in the second and third places (remember, existentials are evaluated to the union subpattern evaluations). That is, $\text{plus}(0, y)$ gives us y.

The inductive case, $\exists t, y.\, \langle \text{succ}(t), y, \text{succ}(X(t, y))\rangle$ uses the graph lookup notation, $\_(\_,\_)$. Given that a set variable X that matches a set of 3-tuples, it gives us a new graph adding one to both the first argument, and the result. That is $\text{plus}(\text{succ}(t), y)$ gives us $\text{succ}(\text{plus}(t, y))$.

In general, given a relation defined by conditional cases, where each φ_i and ψ_i are structural patterns and θ_i are predicates,

$$\begin{aligned} p(\bar{x}) :=& \mathtt{case}_0 : p(\bar{\varphi}_0(\bar{y})) \Rightarrow \psi_0(\bar{x}, \bar{y}) \ \mathtt{if}\ \theta_0(\bar{x}, \bar{y}) \\ &\vdots \\ &|\mathtt{case}_n : p(\bar{\varphi}_n(\bar{y})) \Rightarrow \psi_n(\bar{x}, \bar{y}) \ \mathtt{if}\ \theta_0(\bar{x}, \bar{y}) \end{aligned}$$

is captured by the following, where i ranges over the cases, and j ranges over the arguments $\bar{x}$ of p:

$$p \equiv \mu\, p. \bigvee \exists \bar{x}, \bar{y}.\, \langle \bar{x}, \psi_i(\bar{x}, \bar{y})\rangle \wedge \bigwedge x_j = \varphi_{i_j}(\bar{y}) \wedge \theta_i(\bar{x}, \bar{y})$$

When p is defined in such a way, we may abbreviate this as $p =_{\mathsf{lfp}} \bigvee \Phi_i(\bar{x})$ where each $\Phi_i(\bar{x}) = \exists \bar{y}.\, \langle \bar{x}, \psi_i(\bar{x}, \bar{y})\rangle \wedge \bigwedge x_j = \varphi_{i_j}(\bar{y}) \wedge \theta_i(\bar{x}, \bar{y})$ denotes a case, leaving $\bar{x}$ free.

While the definition of p itself is constructed directly using the μ binder, $p(\bar{x})$ for any values $\bar{x}$ wraps this in the graph lookup operator. Thus the usual application of (KNASTER-TARSKI) and (PREFIXEDPOINT) do not work. Instead we employ these higher-level analogues:

Lemma 4.3. Given a relation $p(\bar{x}) =_{\mathsf{lfp}} \bigvee \Phi_i(\bar{x})$, defined as above, we have

$$(\text{LFP-UNFOLD-R})\frac{\theta \to \exists \bar{y}.\, \Phi_i(\bar{t}, \bar{y})}{\theta \to p(\bar{t})} \text{ for any case } i$$

$$(\text{LFP-KT})\frac{\exists \bar{y}.\, \Phi_i(\bar{t}, \bar{y})[\psi/p] \to \psi[\bar{t}] \text{ for each case } i}{p(\bar{t}) \to \psi[\bar{t}]}$$

The example proofs where these lemmas are used against the naturals seem somewhat forced, and are more natural in the context of separation logic, where recursive definitions are many-valued. A background of separation logic, and its embedding in matching logic is given

in Chapter 1 and is necessary for the understanding of the following example.

Theorem 4.1. We prove $\vdash \mathsf{ll}(x, y) \to \mathsf{lr}(x, y)$, from [62] Example 2, where

$$\mathsf{ll}(x, y) =_{\mathsf{lfp}} (x = y \land \mathsf{emp}) \lor (\exists t.\, x \mapsto t * \mathsf{ll}(t, y) \land x \neq y)$$
$$\mathsf{lr}(x, y) =_{\mathsf{lfp}} (x = y \land \mathsf{emp}) \lor (\exists t.\, \mathsf{lr}(x, t) * t \mapsto y \land x \neq y)$$

Proof. Let $C \equiv x \mapsto z * \square \land x \neq y$ and $C'[\square] \equiv x \mapsto z * h \land x \neq w$.

1.	$\mathsf{lr}(x, w) * w \mapsto y \land z \neq y \land x \neq y \to x \neq y \land \mathsf{lr}(x, w) * w \mapsto y$	Clause subsumption
2.	$\mathsf{lr}(x, w) * w \mapsto y \land z \neq y \land x \neq y \to x \neq y \land \exists t.\, \mathsf{lr}(x, t) * t \mapsto y$	(RHS-INSTANTIATE)
3.	$\mathsf{lr}(x, w) * w \mapsto y \land z \neq y \land x \neq y \to \mathsf{lr}(x, y)$	(LFP-UNFOLD-R)
4.	$C'[C' \mathrel{-\!\!*} \mathsf{lr}(x, w)] * w \mapsto y \land z \neq y \land x \neq y \to \mathsf{lr}(x, y)$	(CTXIMP-COLLAPSE)
5.	$x \mapsto z * (C' \mathrel{-\!\!*} \mathsf{lr}(x, w)) * w \mapsto y \land z \neq y \land x \neq y \to \mathsf{lr}(x, y)$	SL Reasoning
6.	$(C' \mathrel{-\!\!*} \mathsf{lr}(x, w)) * w \mapsto y \land z \neq y \to (C \mathrel{-\!\!*} \mathsf{lr}(x, y))$	(CTXIMP-INTRO)
7.	$\exists w.\, (C' \mathrel{-\!\!*} \mathsf{lr}(x, w)) * w \mapsto y \land z \neq y \to (C \mathrel{-\!\!*} \mathsf{lr}(x, y))$	(ELIM-$\exists$)
8.	$\vdots$	Base case omitted
9.	$\mathsf{lr}(z, y) \to C \mathrel{-\!\!*} \mathsf{lr}(x, y)$	(LFP-KT)
10.	$x \mapsto z * \mathsf{lr}(z, y) \land x \neq y \to \mathsf{lr}(x, y)$	(CTXIMP-ELIM)
11.	$\exists z.\, x \mapsto z * \mathsf{lr}(z, y) \land x \neq y \to \mathsf{lr}(x, y)$	(ELIM-$\exists$)
12.	$\vdots$	Base case omitted
13.	$\mathsf{ll}(x, y) \to \mathsf{lr}(x, y)$	(LFP-KT)

QED.

Besides the fixpoints rules mentioned before, this proof additionally uses two key rules for first-order reasoning. Each case of a recursive definitions may have an existential quantifier. These are introduced onto the left-hand side of the goal by (LFP-KT), and may be eliminated by applying (ELIM-$\exists$). Similarly, when introduced to the right-hand side by (LFP-UNFOLD-R) it may be eliminated using (RHS-INSTANTIATE).

Lemma 4.4. In any theory, we can prove

$$(\textsc{elim-}\exists)\frac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi} \text{ if } x \notin \text{free}(\psi)$$

$$(\text{RHS-}\textsc{instantiate})\frac{\varphi \to C[\psi]}{\varphi \to C[\exists x.\, \psi]} \text{ where } C \text{ is a structural context}$$

Let us inspect this proof in greater detail. Since the LHS $\mathsf{ll}(x, y)$ is a recursive definition, we apply directly the (LFP-KT) rule and get two new proof goals. One goal, shown below, corresponds to the base case of the definition of $\mathsf{ll}(x, y)$:

$$\vdash (x = y \land \mathsf{emp}) \to \mathsf{lr}(x, y)$$

The other goal corresponds to the inductive case and is shown in Step 11. For clarity, we breakdown the steps in calculating the substitution $[\mathsf{lr}(x, y)/\mathsf{lr}]$ required by (LFP-KT) below:

$$\begin{array}{ll}
\vdash \mathsf{ll}(x, y) \to \mathsf{lr}(x, y) & \text{before (LFP-KT) is applied} \\
\vdash (\exists z.\, x \mapsto z * \mathsf{ll}(z, y) \land x \neq y) \to \mathsf{lr}(x, y) & \text{phantom step: unfolding to inductive case} \\
\vdash (\exists z.\, x \mapsto z * \mathsf{lr}(z, y) \land x \neq y) \to \mathsf{lr}(x, y) & \text{after (LFP-KT) is applied}
\end{array}$$

Now, the base case goal can be proved by applying (LFP-UNFOLD-R) to unfold the RHS $\mathsf{lr}(x, y)$ to its base case and then merely checking that the left- and right-hand side are identical. The inductive case (after eliminating $\exists z$ from LHS), $\vdash x \mapsto z * \mathsf{lr}(z, y) \land x \neq y \to \mathsf{lr}(x, y)$, contains a recursive pattern $\mathsf{lr}(z, y)$ within a context $C[h] = x \mapsto z * h \land x \neq y$. Therefore, we (CTXIMP-ELIM) the context and yield contextual implication $C \multimap \mathsf{lr}(x, y)$ on the RHS. Then (LFP) is applied, yielding two sub-goals, one for the base case and one for the inductive case. We omit the base case and show the following breakdown steps for the inductive case, for clarity:

$$\begin{array}{ll}
\vdash \mathsf{lr}(z, y) \to (C \multimap \mathsf{lr}(x, y)) & \text{before (LFP-KT) is applied} \\
\vdash (\exists w.\, \mathsf{lr}(z, w) * w \mapsto y \land z \neq y) \to (C \multimap \mathsf{lr}(x, y)) & \text{phantom step} \\
\vdash (\exists w.\, (C \multimap \mathsf{lr}(x, y))[w/y] * w \mapsto y \land z \neq y) \to (C \multimap \mathsf{lr}(x, y)) & \text{after (LFP-KT) is applied}
\end{array}$$

where $(C \multimap \mathsf{lr}(x, y))[w/y] = C' \multimap \mathsf{lr}(x, w)$ and $C'[h] = x \mapsto z * \square \land x \neq w$.

Now the proof proceeds by using (CTXIMP-INTRO) on the context C, and eliminating the

quantifier $\exists w$ by (ELIM-$\exists$). Then the goal becomes:

$$x \mapsto z * (C' \multimap \mathsf{lr}(x, w)) * w \mapsto y \land z \neq y \land x \neq y \to \mathsf{lr}(x, y)$$

At this point, we use the SL axiom $x_1 \mapsto y * x_2 \mapsto z \to x_1 \neq x_2$ given in [12]. to strengthen the left-hand side with the logical constraint $x \neq w$. This lets us make the outer context match one in the contextual implication and allows us to apply (CTXIMP-COLLAPSE). The proof can then be completed using (RHS-INSTANTIATE), and noticing that the clauses on the left-hand side are a subset of those on the right-hand side.

4.3.2 Controling instantiation of free variables

Often when using (LFP-KT), we have a goal $p(\bar{x}) \to \varphi(x, y)$, where $y \notin \bar{x}$. Immediate application results in an unquantified y on the left-hand side, and does not give us a choice in y's instantiation. Fortunately, we may relax this requirement by first *strengthening* the goal to $p(\bar{x}) \to \forall y.\, \varphi(x, y)$, and then applying (LFP-KT), resulting in a weaker overall goal—y is quantified on the left-hand side that may be latter instantiated arbitrarily.

Lemma 4.5.

$$(\text{INTRO-}\forall)\frac{p(\bar{x}) \to \forall \tilde{y}.\ (C \multimap \psi)}{p(\bar{x}) \to (C \multimap \psi)} \text{ where } \bar{y} = \text{free}(\psi) \setminus \bar{x}$$

Let us see how this is useful in practice. Consider the following slightly modified definitions of lr and lr that take a third argument s denoting the set of elements in the list segment:

Theorem 4.2. We may prove $\vdash \mathsf{ll}(x, y, s) \to \mathsf{lr}(x, y, s)$, where:

$$\mathsf{ll}(x, y, s) =_{\mathsf{lfp}} (x = y \land \mathsf{emp} \land s = \emptyset) \lor \exists x_1, s_1.\, x \mapsto x_1 * \mathsf{ll}(x_1, y, s_1) \land s = s_1 \cup \{x\} \land x \neq y$$
$$\mathsf{lr}(x, y, s) =_{\mathsf{lfp}} (x = y \land \mathsf{emp} \land s = \emptyset) \lor \exists y_1, s_1.\, \mathsf{lr}(x, y_1, s_1) * y_1 \mapsto y \land s{=}s_1 \cup \{y_1\} \land x \neq y$$

Its proof below is similar to the previous one, except that the use of rule (INTRO-$\forall$) is *necessary* for the proof to succeed, because we need to instantiate the quantifier $\forall s$ of goal with a fresh variable.

Proof. Let,

$$C[\square] \equiv x \mapsto z * \square \wedge s = s_1 \cup \{x\} \wedge x \neq y$$
$$C'[\square] \equiv C[\square][w/y, s_2/s_1] = x \mapsto z * \square \wedge s = s_2 \cup \{x\} \wedge x \neq w$$
$$\varphi \equiv w \mapsto y \wedge s_1 = s_2 \cup \{w\} \wedge z \neq y \wedge s = s_1 \cup \{x\} \wedge x \neq y$$

1.	$\mathsf{lr}(x, w, s_3) * \varphi \to \mathsf{lr}(x, w, s_3) * w \mapsto y \wedge s{=}s_3 \cup \{w\} \wedge x \neq y$	(Domain Reasoning)
2.	$\mathsf{lr}(x, w, s_3) * \varphi \to \exists t \exists s_4.\, \mathsf{lr}(x, t, s_4) * t \mapsto y \wedge s{=}s_4 \cup \{t\} \wedge x \neq y$	(RHS-INSTANTIATE)
3.	$\mathsf{lr}(x, w, s_3) * \varphi \to \mathsf{lr}(x, y, s)$	(LFP-UNFOLD-R)
4.	$x \mapsto z * (C'[s_3/s] \mathrel{-\!\!*} \mathsf{lr}(x, w, s_3))) * \varphi \to \mathsf{lr}(x, y, s)$	(CTXIMP-COLLAPSE)
5.	$x \mapsto z * (\forall x \forall s.\, (C' \mathrel{-\!\!*} \mathsf{lr}(x, w, s))) * \varphi \to \mathsf{lr}(x, y, s)$	(INSTANTIATE-LHS)
6.	$C[\exists w \exists s_2.\, (\forall x \forall s.\, (C' \mathrel{-\!\!*} \mathsf{lr}(x, w, s))) * \varphi] \to \mathsf{lr}(x, y, s)$	(ELIM-$\exists$)
7.	$\exists w \exists s_2.\, (\forall x \forall s.\, (C' \mathrel{-\!\!*} \mathsf{lr}(x, w, s))) * w \mapsto y \wedge s_1{=}s_2 \cup \{w\} \wedge z \neq y \to (C \mathrel{-\!\!*} \mathsf{lr}(x, y, s))$	(CTXIMP-INTRO)
8.	$\exists w \exists s_2.\, (\forall x \forall s.\, (C' \mathrel{-\!\!*} \mathsf{lr}(x, w, s))) * w \mapsto y \wedge s_1 = s_2 \cup \{w\} \wedge z \neq y \to \forall x \forall s.\, (C \mathrel{-\!\!*} \mathsf{lr}(x, y, s))$	(ELIM-$\forall$)
9.	$\vdots$	Base case omitted
10.	$\mathsf{lr}(z, y, s_1) \to \forall x \forall s.\, (C \mathrel{-\!\!*} \mathsf{lr}(x, y, s))$	(LFP-KT)
11.	$\mathsf{lr}(z, y, s_1) \to (C \mathrel{-\!\!*} \mathsf{lr}(x, y, s))$	(INTRO-$\forall$)
12.	$x \mapsto z * \mathsf{lr}(z, y, s_1) \wedge s = s_1 \cup \{x\} \wedge x \neq y \to \mathsf{lr}(x, y, s)$	(CTXIMP-ELIM)
13.	$\exists z, s_1.\, x \mapsto z * \mathsf{lr}(z, y, s_1) \wedge s{=}s_1 \cup \{x\} \wedge x \neq y \to \mathsf{lr}(x, y, s)$	(ELIM-$\exists$)
14.	$\vdots$	Base case omitted
15.	$\mathsf{ll}(x, y, s) \to \mathsf{lr}(x, y, s)$	(LFP-KT)

QED.

Suppose there is no application of rule (INTRO-$\forall$). Then, instead of having ‡, we will have

$$x \mapsto z * (C' \mathrel{-\!\!*} \mathsf{lr}(x, w, s)) * w \mapsto y \wedge s_1 = s_2 \cup \{w\} \wedge z \neq y \wedge s = s_1 \cup \{x\} \wedge x \neq y \to \mathsf{lr}(x, y, s)$$

where $C'[\square] = x \mapsto z * \square \wedge s = s_2 \cup \{x\} \wedge x \neq w$. So we cannot match $s{=}s_1 \cup \{x\} \wedge s_1{=}s_2 \cup \{w\}$

in the outer context with $s{=}s_2 \cup \{x\}$ in the inner context. In other words, we cannot eliminate the inner context, and so the proof will get stuck.

4.3.3 A Mutual Recursion Example

Mutually recursive definitions are in general defined as:

$$\begin{cases} p_1(\bar{y}_1) =_{\mathsf{lfp}} \exists \bar{x}_{11}.\, \varphi_{11}(\bar{y}_1, \bar{x}_{11}) \vee \cdots \vee \exists \bar{x}_{1m_1}.\, \varphi_{1m_1}(\bar{y}_1, \bar{x}_{1m_1}) \\ \qquad \cdots \\ p_k(\bar{y}_k) =_{\mathsf{lfp}} \exists \bar{x}_{k1}.\, \varphi_{k1}(\bar{y}_k, \bar{x}_{k1}) \vee \cdots \vee \exists \bar{x}_{km_k}.\, \varphi_{km_k}(\bar{y}_k, \bar{x}_{km_k}) \end{cases}$$

which simultaneously define k recursive definitions $p_1, \ldots, p_k$ to be the least that satisfies the equations. Our way of dealing with mutual recursion is to reduce it to several non-mutual, ordinary recursions. We use the following separation logic challenge test `qf_shid_entl/10.tst.smt2` from the SL-COMP'19 competition [63] as an example.

Theorem 4.3. Let

$$\begin{cases} \mathsf{IIO}(x, y) =_{\mathsf{lfp}} x \mapsto y \vee \exists t.\, x \mapsto t * \mathsf{IIE}(t, y) \\ \mathsf{IIE}(x, y) =_{\mathsf{lfp}} \exists t.\, x \mapsto t * \mathsf{IIO}(t, y) \end{cases}$$

Then: $\vdash \mathsf{IIO}(x, y) * \mathsf{IIO}(y, z) \to \mathsf{IIE}(x, z)$.

To proceed with the proof, we first reduce the mutual recursion definition into the following two non-mutual, simple recursion definitions, which can be obtained systematically by unfolding the other recursive symbols to exhaustion.

$$\begin{aligned} \mathsf{IIO}(x, y) &=_{\mathsf{lfp}} x \mapsto y \vee \exists t_1 \exists t_2.\, x \mapsto t_1 * t_1 \mapsto t_2 * \mathsf{IIO}(t_2, y) \\ \mathsf{IIE}(x, y) &=_{\mathsf{lfp}} \exists t_1 \exists t_2.\, x \mapsto t_1 * t_1 \mapsto t_2 * \mathsf{IIE}(t_2, y) \end{aligned}$$

Then, the proof can be carried out in the normal way, seen below.

Proof. Let $C[\Box] \equiv \Box * \mathsf{IIO}(y, z)$.

1.	$x \mapsto t_1 * t_1 \mapsto t_2 * \mathsf{IIE}(t_2, z) \to x \mapsto t_1 * t_1 \mapsto t_2 * \mathsf{IIE}(t_2, z)$	Clause subsumption
2.	$x \mapsto t_1 * t_1 \mapsto t_2 * \mathsf{IIE}(t_2, z) \to \exists u_1 \exists u_2.\, x \mapsto u_1 * u_1 \mapsto u_2 * \mathsf{IIE}(u_2, z)$	(INSTANTIATE-RHS)
3.	$x \mapsto t_1 * t_1 \mapsto t_2 * \mathsf{IIE}(t_2, z) \to \mathsf{IIE}(x, z)$	(UNFOLD-R)
4.	$x \mapsto t_1 * t_1 \mapsto t_2 * (C \multimap \mathsf{IIE}(t_2, z)) * \mathsf{IIO}(y, z) \to \mathsf{IIE}(x, z)$	(CTXIMP-COLLAPSE)
5.	$\exists t_1 \exists t_2.\, x \mapsto t_1 * t_1 \mapsto t_2 * (C \multimap \mathsf{IIE}(t_2, z)) * \mathsf{IIO}(y, z) \to \mathsf{IIE}(x, z)$	(ELIM-$\exists$)
6.	$\exists t_1 \exists t_2.\, x \mapsto t_1 * t_1 \mapsto t_2 * (C \multimap \mathsf{IIE}(t_2, z)) \to (C \multimap \mathsf{IIE}(x, z))$	(CTXIMP-ELIM)
7.	$\vdots$	Base case omitted
8.	$\mathsf{IIO}(x, y) \to C \multimap \mathsf{IIE}(x, z)$	(LFP-KT)
9.	$\mathsf{IIO}(x, y) * \mathsf{IIO}(y, z) \to \mathsf{IIE}(x, z)$	(CTXIMP-ELIM)

QED.

4.3.4 Greatest Fixed Points

For our final proof of the chapter, we will show a simple example that employs greatest fixpoints. Since the domain of both the naturals as well as separation logic do not involve greatest fixpoints, we will move to linear temporal logic. This has the additional purpose of demonstrating the generality of our framework.

We show an example of proving the induction proof rule of the sound and complete proof system of LTL[64], which uses the ML axiomatization for LTL given in Section 1.5 and the greatest fixpoint reasoning. In general, the rules for reasoning about greatest fixpoints are dual to that of least-fixpoints. The key dual rule (GFP-KT), is shown below.

Consider the following induction proof rule in the sound and complete LTL proof system: $\vdash p \wedge \Box(p \to \circ p) \to \Box p$. As defined in Chapter 1, the *always* temporal operator $\Box$ is a greatest fixpoint: $\Box\varphi =_{\text{gfp}} \varphi \wedge \circ\Box\varphi$. To reason about it, we need a set of proof rules dual to those shown previously, where the key rule, (GFP) dual to (LFP-KT), is shown below:

$$(\text{GFP})\frac{\varphi \to \psi_i[\varphi/q]}{\varphi \to q(\tilde{y})} \text{ where } q(\tilde{y}) =_{\text{gfp}} \bigvee \psi_i$$

(GFP) is used to discharge the RHS $\Box p$ of the proof goal. Note that during the proof we use the distributivity law provided by the ML theory Γ_{LTL}, denoted as proof step ($\circ\wedge$) below.

1. $p \wedge (p \to \circ p) \wedge \circ\Box(p \to \circ p) \to \circ p \wedge \circ\Box(p \to \circ p)$ (SMT)
2. $p \wedge \Box(p \to \circ p) \to \circ p \wedge \circ\Box(p \to \circ p)$ (UNFOLD-L)
3. $p \wedge \Box(p \to \circ p) \to \circ(p \wedge \Box(p \to \circ p))$ ($\circ\wedge$)
4. $p \wedge \Box(p \to \circ p) \to p \wedge \circ(p \wedge \Box(p \to \circ p))$ (PATTERN-MATCH)
5. $p \wedge \Box(p \to \circ p) \to \Box p$ (GFP-KT)

CHAPTER 5: A SIMPLE DFS AUTOMATION

In this section, we will describe the implementation of a simple prover targeted at separation logic with inductive definitions, reachability, as well as basic linear temporal logic proofs. The embeddings of each of these were seen in Chapter 1. The prover is built from building blocks described above and employs a simple, syntax-driven depth-first search over a small set of proof rules.

While the matching logic proof rules on their own are not amenable to automation, we found a small set of largely generic proof rules that were very effective for separation these logics. These proof rules are shown in the following, and are built from those mentioned in the previous chapter.

To evaluate our unified proof framework and prototype implementation, we consider four representative logical systems for fixpoint reasoning:

(1) separation logic extended with recursive definitions [65, p. CDNQ12], abbreviated SL;
(2) linear temporal logic [66], abbreviated LTL; and
(3) reachability logic [67], abbreviated RL,
(4) first-order logic extended with least fixpoints (LFP).

SL is the representative logic for reasoning about data-manipulating programs with pointers; LTL is the temporal logic of choice for model checkers of infinite-trace systems, e.g., SPIN [68]; RL is a language-parametric generalization of Hoare logic [69], where the programming language semantics is given as an input theory and partial correctness is specified and proved as a reachability rule $\varphi_{\text{pre}} \to \psi_{\text{post}}$. These logics therefore represent relevant instances of fixpoint reasoning across different and important domains. We believe that they form a good benchmark for evaluating a unified proof framework for fixpoint reasoning, so we set ourselves the long-term goal to support *all* of them. We will give special emphasis to separation logic (SL) here, however, because it gathered much attention in recent years that resulted in several automated SL provers and its own international competition SL-COMP'19 [63].

It would be unreasonable to hope at such an incipient stage that a generic automated prover can be superior to the state-of-the-art domain-specific provers and algorithmic decision procedures for all four logics, on all existing challenging benchmarks in their respective domains. Therefore, for each of the domains, we set ourselves a limited objective. For SL, the goal was to prove all the 280 benchmark properties collected by SL-COMP'19 in the problem set `qf_shid_entl` dedicated to inductive reasoning. For LTL, the goal was to prove

the axioms about the modal operators "always" $\Box\varphi$ and "until" $\varphi_1 \, U \, \varphi_2$ (whose semantics are defined as fixpoints) in its complete proof system. For RL, our goal was to verify a simple imperative program `sum` that computes the total of 1 to input n, and show that it returns the correct sum $n(n+1)/2$ on termination. We report what we have done in pushing towards the above goals, and discuss the difficulties that we met, and the lessons we learned.

This prover will be implemented as a simple, bounded depth-first search automating the application of the rules mentioned in the previous chapter.

5.1 AUTOMATION

To automate the application of these rules, there are still some things to take care of. For example, how do we choose instantiations when working with (INSTANTIATE-LHS) and (INSTANTIATE-RHS)? Does the ordering of unfolding and induction matter? With some simple heuristics, we were able to get a surprisingly effective prover.

5.1.1 Top-Level Proof Search Heuristic

The proof search heuristic is defined in a strategy language over the proof rules above. Each atomic strategy matches the current goal against the conclusion of the corresponding proof rule, and replaces it with goals corresponding to the set of hypotheses of the proof rule. Strategies may be composed sequentially using the sequential composition operator ".", or via the choice operator "|" where each argument is executed to completion in turn, giving us a depth-first search. The default solver strategy is as follows:

```
rule search(kt-bound: KTBOUND, unfold-bound: UNFOLDBOUND)
  => normalize
   . ( instantiate-domain-axioms
     . unfold-all(bound: UNFOLDBOUND) . normalize . match
     . substitute-equals-for-equals . structural-patterns-equal . smt
     | ( induct-each . search(kt-bound: KTBOUND -Int 1, unfold-bound: UNFOLDBOUND) )
     )
```

It begins by `normalize`-ing the goal into a standard form. For the most part this involves transforming each side of the goal into a conjunction of predicates and structural patterns. If there are disjunctions on either side they are split off into separate goals. It then tries two substrategies.

In the first strategy, it tries proving the goal with only simple (non-inductive) unfolding on

the right-hand side, and immediately discharging the goal to the structural and domain solvers.

The `instantiate-domain-axioms` instantiates the domain axioms that are implied by the left hand side of the goal. For example, in SL this includes the following axioms that force conjoined heaps to have separate footprints:

$$x_1 \mapsto y * x_2 \mapsto z \rightarrow x_1 \neq x_2$$

`unfold-all` tries various combinations of unfoldings to see if that enables the later strategies to complete the proof. Since the order of unfolding doesn't matter, `unfold-all` lists out each possible combination and tries them in turn, using the `unfold()` below.

```
rule unfold(symbolList: P, Ps, bound: N)
  => normalize
   . substitute-equals-for-equals
   . ( ( match . structural-patterns-equal . smt )
     | ( unfold(P) . unfold(symbols: P, Ps, bound: N -Int 1) )
     | ( unfold(symbols: Ps, bound: N) )
     )
```

`match` directs the application of the (INSTANTIATE-RHS), and is described more in detail later. `structural-patterns-equal` removes structural patterns from the right-hand side that are also present on the left-hand side. If the left hand side of the goal includes an equality between two variables, `substitute-equals-for-equals` will substitute one for the other.

Finally, `smt` attempts to discharge the remaining non-structural clauses using SMT solvers such as Z3[70] and CVC4[71], where recursive symbols are treated as uninterpreted functions. Note that `smt` is the only proof rule that *finishes* the proof. In practice, goals that can be proved by `smt` are those about the common mathematical domains such as natural and integer numbers, and set membership.

In the second strategy, we attempt to prove the goal using induction on each recursive definition in turn. When applying the induction related rules, the ordering *does* matter, so `induct-each` picks a recursive symbol on the left-hand side of the goal and tries to prove the goal by inducting on that, and then recursively applying the search strategy with a lower bound.

```
rule induct(symbol: P)
  => ctximp-elim . intro-forall . lfp-kt
   . normalize
```

```
. exists-elim . ctximp-intro
. canonicalize . kt-collapse
```

5.1.2 Context Matching

(MATCH-CTX) deals with the (quantified) contextual implication $\forall \bar{y}.\,(C' \multimap \psi')$ on the LHS introduced by (LFP-KT) and is one of the most complicated proof rule in our proof system. Note that (LFP-KT) does the substitution $[\forall \bar{y}.\,(C \multimap \psi)/p]$, replacing each recursive occurrence $p(\bar{x}')$ (where $\bar{x}'$ might be different from the original argument $\bar{x}$) by $(\forall \bar{y}.\,(C \multimap \psi))[\bar{x}'/\bar{x}]$, whose result we denote as $\forall \bar{y}.\,(C' \multimap \psi')$. The number of contextual implications on the LHS is the same as the number of recursive occurrences of p in its definition. (MATCH-CTX) eliminates one contextual implication at a time, through a **c**ontext **m**atching algorithm.

$$(\text{MATCH-CTX})\frac{C_{rest}[\varphi'\theta] \to \psi}{C_o[\forall \tilde{y}.\,(C' \multimap \varphi')] \to \psi} \text{ where } (C_{rest}, \theta) = \texttt{cm}(C_o, C', \tilde{y})$$

In principle, (CTXIMP-COLLAPSE) can be used to handle contextual implication on the LHS. If contextual implication $C' \multimap \psi$ happens to occur within the same context C', then we can replace $C'[C' \multimap \psi]$ by ψ'. However, situations in practice are more complex. Firstly, contextual implication can be *quantified*, i.e., $\forall \bar{y}.\,(C' \multimap \psi')$, so we need to first instantiate it using a substitution θ, to $C'\theta \multimap \psi'\theta$. Secondly, the out-most context C_o might contain more than needed to match with $C'\theta$. So after matching, the remaining unmatched context, denoted C_{rest}, stays in the proof goal. The **c**ontext **m**atching algorithm `cm` implements heuristics to find a suitable substitution θ such that $C'\theta$ matches (a part of) the outer-most context C_o, and when succeeding, it returns θ and the remaining unmatched context C_{rest}.

Procedure `cm` is used to check whether the inner context can be matched with the outer context. For example, suppose we have the following proof goal:

$$\vdash C_{outer}[\forall \tilde{y}.\,(C' \multimap \varphi')] \to \psi \tag{5.1}$$

$\texttt{cm}(C_{outer}, C', \tilde{y})$ takes as inputs the outer context C_{outer}, the inner context C' and a list of quantifier variables $\tilde{y}$. To check if C' can be matched by (a part of) C_{outer}, it builds the following proof goal:

$$\vdash C_{outer} \to \exists \tilde{y}.\,C' \tag{5.2}$$

In (5.1), what we want is to initialize the universal variables $\forall \tilde{y}$ in order for the inner context C' to be matched with some part of C_{outer}. That is also the purpose of using the existential

variables $\exists\tilde{y}$ in (5.2). The change of the quantifier $\tilde{y}$ from universal to existential is because we have moved the inner context C' from LHS to RHS.

To deal with (5.2), `cm` will call the modified version of the `search()` strategy. The difference between the modified version and the `search()` strategy is only on the returning result. Specifically, when it succeeds the modified version additionally (1) returns the *remaining, unmatched* part of the LHS , denoted C_{rest}, after consuming all the matched constraints from the RHS, and (2) collects the instantiation of $\bar{y}$, denoted θ, when applying rule (PATTERN-MATCH). (Note that C_{rest} may contain structure patterns as we have seen in the SL examples.) Specifically, if we can prove (5.2), we have $\vdash C_{outer} \to C'\theta$. Furthermore, C_{rest} is the remaining part after removing the constraint of $C'\theta$ from C_{outer}, so we have $C_{rest}[C'\theta[C'\theta \multimap \varphi'\theta]] \to \psi$. As a result, we now can proceed to prove new proof goal $C_{rest}[\varphi'\theta] \to \psi$.

5.1.3 Pattern Matching

(PATTERN-MATCH) uses the pattern matching algorithm, `pm`, to instantiate the quantified variable(s) $\bar{y}$ on the RHS. The algorithm returns a match result as a substitution θ, which tells us how to instantiate the variables $\bar{y}$. If match succeeds, the instantiated proof goal $\varphi \to \psi\theta$ should be immediately proved by (SMT).

$$(\text{PATTERN-MATCH})\frac{\varphi \to \psi\theta}{\varphi \to \exists\tilde{y}.\,\psi} \text{ where } \theta \in \texttt{pm}(\varphi, \psi, \tilde{y}) \text{ matches } \varphi \text{ with } \psi$$

Note that the soundness of our proof framework does not rely on the correctness of the matching algorithm, because (PATTERN-MATCH) is basically a standard FOL proof rule and holds for any substitution θ. The matching algorithm is a heuristic to find a good θ. We rely on the external SMT solver to check the correctness of the match result given by the matching algorithm, through rule (SMT).

The combination of (PATTERN-MATCH) (based on the pattern matching algorithm `pm`) and (SMT) (based on SMT solvers) gives us the ability to do *static reasoning* about structural patterns. In separation logic (SL), for example, structural patterns correspond to *spatial formulas* built from the heap constructors emp, $\mapsto$, and $*$, whose behaviors are axiomatized as the algebraic specification given in Section 1.6 where $*$ is associative and commutative and emp is its unit. If the matching algorithm `pm` does not support matching modulo associativity (A), commutativity (C), and unit elements (U), then it cannot effectively discharge (separation logic) goals that are provable. In general, matching modulo any (given) set of equations is undecidable [72], so in this paper, we implement a naive matching

algorithm that supports matching modulo associativity (A-matching), and matching modulo associativity and commutativity (AC-matching), has been quite effective so far.

Procedure `pm`, used by rule (PATTERN-MATCH), implements a naive, brute-force algorithm that does matching modulo associativity and/or associativity-and-commutativity. Procedure `pm` takes as input

- a list of "pattern" patterns $[\psi_i]_1^m \equiv [\psi_1, \ldots, \psi_m]$;
- a list of "term" patterns $[\varphi_j]_1^m \equiv [\varphi_1, \ldots, \varphi_m]$;
- a set of existential variables EVar with $\text{EVar} \cap \bigcup_1^n \text{free}(\varphi_j) = \emptyset$ and $\text{EVar} \subseteq \bigcup_1^m \text{free}(\psi_i)$.

Then it returns Failure or the match result θ with $\text{domains}(\theta) \subseteq \text{EVar}$ and $\psi_i\theta \equiv \varphi_i$, for all i.

5.2 EVALUATION

We implemented our proof framework in the $\mathbb{K}$ framework (http://kframework.org). $\mathbb{K}$ has a modular notation for defining rewrite systems. Since our proof framework is essentially a rewriting system that *rewrites/reduces* proof goals to sub-goals, it is convenient to implement it in $\mathbb{K}$.

As discussed previously, we evaluated our prototype implementation using four representative logical systems for fixpoint reasoning: first-order logic extended with least fixpoints (LFP), separation logic (SL), linear temporal logic (LTL), and reachability logic (RL). Our evaluation plan is as follows: For SL, we used the 280 benchmark properties collected by the SL-COMP'19 competition [63]. These properties are entailment properties about various inductively-defined heap structures, including several hand-crafted, challenging structures. For LTL, we considered the (inductive) axioms in the complete LTL proof system (see, e.g., [64],journals/igpl/LichtensteinP00). For LFP and RL, we considered the program verification of a simple program `sum` that computes the total sum from 1 to a symbolic input n. We shall use two different encodings to capture the underlying transition relation: the LFP encoding defines it as a binary predicate and the RL encoding defines it as a reachability rule.

Before we discuss our evaluation results, we would like to point out that it would be unreasonable to expect that a unified proof framework can outperform the state-of-the-art provers and algorithms for all specialized domains from the first attempt. We believe that this *is* possible and within our reach in the near future, but it will likely take several years of sustained effort. We firmly believe that such effort will be worthwhile spending, because if successful then it will be transformative for the field of automated deduction and thus

program verification. Here, we focus on demonstrating the generality of our proof framework. We shall also report the difficulties that we experienced.

Our first evaluation is based on the standard separation logic benchmark set collected by SL-COMP'19 [63]. These benchmarks are considered challenging because they are related to heap-allocated data structures along with *user-defined* recursive predicates crafted by participants to challenge the competitors. Among the benchmarks, we focus on the qf_shid_entl division that contains entailment problems about inductive definitions. This division is considered the hardest, specifically because many of its tests require proofs by induction. As such, this division is a good case study for testing the generality of our generic proof framework. Furthermore, heap provers are currently considered to have the most powerful implementations of automated inductive reasoning, so we would be comparing our prototype with the state-of-the-art in automated inductive reasoning.

To set up our prover for the SL benchmarks, we *instantiate* it with the set Γ_{Heaps} of axioms that captures SL. Note that the associativity and commutativity of $\varphi_1 * \varphi_2$ are handled by the *built-in* pattern matching algorithms, so the most important axioms are the two specifying non-zero locations and no-overlapped heap unions. The experimental results show that our generic prover can prove 265 of the 280 benchmark tests, placing it third place among all participants.

Interestingly, we noted that (FRAME) is not necessary for most tests. Only 12 out of the 265 tests used (FRAME) reasoning. More experiments are needed to draw any firm conclusion, but it could be (FRAME) reasoning mostly improves performance, as it reduces the matching search space and thus proof search terminates faster, but does not necessarily increase the expressiveness of the prover. The 15 tests that our prover cannot handle come from the benchmarks of automata-based heap provers [74]. These benchmarks demand more sophisticated *SL-specific* reasoning that require more complex properties about heaps/maps than what our prover can naively derive from the Γ_{Heaps} theory with its current degree of automation; while we certainly plan to handle those as well in the near future, we would like to note that they are *not* related to fixpoint reasoning, but rather to reasoning about maps. The two provers that outperform our generic prover, Songbird and S2S, are both specialized for SL. Compared with generic provers such as Cyclist[75], our prover proves 13 more tests.

Table 5.1 illustrates some of the more interesting SL properties that our prover can verify automatically. These are common lemmas about heap structures that arise and are collected when verifying real-world heap-manipulating programs. For example, the property on the first line says that a sorted list is also a list, a typical verification condition arising in formal

Table 5.1: Selected separation logic properties, automatically proved by our prover

$\mathsf{sorted\_list}(x, \min) \to \mathsf{list}(x)$
$\mathsf{sorted\_list}_1(x, \mathrm{len}, \min) \to \mathsf{list}_1(x, \mathrm{len})$
$\mathsf{sorted\_list}_1(x, \mathrm{len}, \min) \to \mathsf{sorted\_list}(x, \min)$
$\mathsf{sorted\_ll}_2(x, y) * \mathsf{sorted\_list}_2(y) \to \mathsf{sorted\_list}_2(x)$
$\mathsf{sorted\_ll}(x, y, \min, \max) * \mathsf{sorted\_list}(y, min_2) \wedge\ max \leq min_2\ \to\ \mathsf{sorted\_list}(x, min)$
$\mathsf{lr}(x, y) * \mathsf{list}(y) \to \mathsf{list}(x)$
$\mathsf{lr}(x, y) \to \mathsf{ll}(x, y)$
$\mathsf{ll}(x, y) \to \mathsf{lr}(x, y)$
$\mathsf{ll}_1(x, y, \mathrm{len}_1) * \mathsf{ll}_1(y, z, \mathrm{len}_2) \to \mathsf{ll}_1(x, z, \mathrm{len}_1{+}\mathrm{len}_2)$
$\mathsf{lr}_1(x, y, \mathrm{len}_1) * \mathsf{list}_1(y, \mathrm{len}_2) \to \mathsf{list}_1(x, \mathrm{len}_1{+}\mathrm{len}_2)$
$\mathsf{ll}_1(x, \mathrm{last}, \mathrm{len}) * (\mathrm{last} \mapsto \mathrm{new}) \to \mathsf{ll}_1(x, \mathrm{new}, \mathrm{len}+1)$
$\mathsf{dlr}(x, y) * \mathsf{dlist}(y) \to \mathsf{dlist}(x)$
$\mathsf{dll}_1(x, y, len_1) * \mathsf{dll}_1(y, z, len_2) \to \mathsf{dll}_1(x, z, \mathrm{len}_1 + \mathrm{len}_2)$
$\mathsf{dlr}_1(x, y, len_1) * \mathsf{dlist}_1(y, len_2) \to \mathsf{dlist}_1(x, \mathrm{len}_1 + \mathrm{len}_2)$
$\mathsf{avl}(x, hgt, min, max, balance) \to \mathsf{bstree}(x, hgt, min, max)$
$\mathsf{bstree}(x, height, min, max) \to \mathsf{bintree}(x, height)$

verification. Table 5.1 also shows several proof goals about singly-linked lists and list segments (specified by predicates ls, list, ll, lr, etc.), doubly-linked lists and list segments (specified by predicates dls, dlist, etc.), and trees.

Our second study is to automatically prove the inductive axioms in the complete LTL proof system, whereas the proof tree of the most interesting of them, (IND), has been given in Section 1.5. Note that LTL is essentially a structure-less logic, as its formulas are only built from temporal operators and propositional connectives, and its models are infinite traces of states that have no internal structures and are modeled as "points". The structure-less-ness of LTL made fixpoint reasoning for it simpler, as no context reasoning or frame reasoning was needed.

Our final study considers a simple program `sum` that computes the total from 1 to a symbolic input n. We do the verification of `sum` following two approaches: RL and LFP. For LFP, program configurations are encoded as FOL terms and the program semantics is encoded as a binary FOL predicate that captures the transition relation. In particular, reachability is defined as a recursive predicate based on the semantics. Our prover then becomes *a (language-independent) program verifier*, different from Hoare-style verification (where a language-specific verification condition generator is required). Following a similar idea, we also considered the prototypical heap-manipulating program `reverse` that reverses a singly-linked list, whose complete proof tree we exile to [76].

We ran these tests on a single core virtual machine with 8GB of RAM. The SL-COMP'19 tests took a total 13 hours to finish, including two outliers that took approximately one and

three hours to complete. The two LTL tests took approximately three minutes, while the ten first order logic tests took seven minutes to complete and the sum program takes a minute to complete. To reiterate, we do not expect our prover to outperform specialized provers this early in its development. These results do, however, show that a unified, powerful and efficient proof framework is within reach.

In summary, we evaluated our generic proof framework using four different logical systems and demonstrated its generality with respect to fixpoint reasoning. We find it encouraging that our generic framework is comparable in terms of automation with specialized state-of-the-art inductive provers for SL, while at the same time also works with other, distinct domains, such as LTL and program verification of partial correctness. We have noticed that one major bottleneck of our implementation is its relatively weak support for non-fixpoint reasoning, such as pattern matching and standard FOL reasoning. We plan to investigate smarter proof search strategies, which shall improve its efficiency and performance.

5.3 RELATED WORK

We were inspired and challenged by work on automation of inductive proofs for separation logic [65], which resulted in several automatic separation logic provers; see [63] for those that participated in the recent SL-COMP'19 competition. Since separation logic is undecidable [77], many provers implement only decision procedures to decidable fragments [73], [78], [79], [80] or incomplete algorithms [74], [81], [82]. There is also work on decision procedures for other heap logics [83], [84], [85], [86], [87], [88], which achieve full automation but suffer from lack of expressiveness and generality. It is worth noting that significant performance improvements can be obtained by incorporating first-order theorem proving and SMT solvers [70], [71] into separation logic provers [89], [90].

Compared with our unified proof framework, the above provers are specialized to separation logic reasoning. Some are based on reductions from separation logic formulas to certain decidable computational domains, such as the satisfiability problem for monadic second-order logic on graphs with bounded tree width [74]. Others are based on separation logic proof trees, where the syntax of separation logic has been hardwired in the prover. For example, most separation logic provers require the following canonical form of separation logic formulas: $\varphi_1 * \cdots * \varphi_n \wedge \psi$ where $\varphi_1, \ldots, \varphi_n$ are basic spacial formulas built from singleton heaps $x \mapsto y$ or user-defined recursive structures such as $\mathsf{list}(x)$, and ψ is a FOL logical constraint. This built-in separation logic syntax limits the use of these provers to separation logic, even though the inductive proof rules proposed by the above provers might be more general. The major

advantage of our unified proof framework, which was the motivation fueling our effort, is that the inductive principle can be applied to any structures, not only those representing heap structures—we have shown the key elements of our proof system that supports the fixpoint reasoning for arbitrary structures.

Hoare-style formal verification represents another important but specialized approach to fixpoint reasoning, where the objects of study are program executions and the properties to prove are program correctness claims. There is a vast literature on verification tools based on classical logics and SMT solvers such as Dafny [91], VCC [20] and Verifast [92]. To use these tools, the users often need to provide annotations that explicitly express and manipulate frames, whose proofs are based on user-provided lemmas. The correctness of the lemmas is either taken for granted or manually proved using an interactive proof assistant (e.g., [20, Sec. 6] mentions several tools that are based on Coq [93] or Isabelle [94]). While it is acceptable for deductive verifiers to take additional annotations and/or program invariants, the use of manually-proved lemmas is not ideal.

There is recent work that considers inductive reasoning for more general data structures, beyond only heap structures [62], [75], [95], [96], [97], [98]. For example, [75] proposes Cyclist, a proof framework that implements a generic notion of *cyclic proof* as a "design pattern" about how to do inductive reasoning, which generalize the proof systems of LFP and SL. In Cyclist, inductive reasoning is achieved not by an explicit induction proof rule, but implicitly by cyclic proof trees with "back-links". In contrast, our unified proof framework uses one fixed logic (matching logic) and relies on an explicit induction proof rule . Therefore, Cyclist represents a different approach from ours but towards a similar goal of a unified framework for fixpoint reasoning.

Indeed, in the next chapter, we will see how we may deal with these cyclic proofs, or "proof with backlinks" in matching logic. As a first pass, we will tackle a simpler domain than separation logic—regular expressions and finite automata.

CHAPTER 6: REGULAR EXPRESSIONS AND FINITE AUTOMATA

In this chapter, we will show how, using the same building blocks, we may drive a more complex proof search. This is in contrast to the naïve depth-first search, presented in the previous chapter, Chapter 5. We may also think of this proof search, as decision procedure that in addition incrementally build a "certificate" guaranteeing the correctness of that particular run.

In this chapter, we will use Brzozowski's method for deciding equality of extended regular expressions as our example—we are certain that this technique carries over to other algorithms that manipulate automata in their implementation. This method has two phases. First, the equality is converted to finite automata by repeatedly taking its "derivative". Second, it checks that each node of that automata is accepting. Similarly, the generated proof will have two components—one that proves the regular expression is equivalent to a pattern representing the automata, and a second that shows that this pattern is valid—i.e. matches all words.

The key to this is our ability to capture finite automata as *structural-analoguous* patterns—patterns whose syntactical structure mimics the computational structure of the automata. For example, cycles in the automata's graph are captured using the fixpoint binder, whereas non-determinism is captured using logical disjunctions.

This mirroring makes it easy to capture algorithms that manipulate automata as part of their operation. Computational manipulations of an automata, such as moving the initial note to an adjacent one correspond directly to syntactical and logical operations on the pattern. We believe that this is a general technique that will work beyond finite automata, for example with push-down automata, tree automata and Büchi automata.

Let us first see how finite words, regular expressions, and automata may be captured in matching logic.

6.1 A MODEL OF FINITE WORDS

Here, we introduce a model $\mathbb{W}$ as the standard model of finite words, over a signature Σ_{Word} containing constants ϵ and a for each $a \in A$, and a binary symbol concat for concatenation. This model allows us to describe languages, including those of regular expressions and finite automata as patterns.

Definition 6.1. Let $\mathbb{W}$ be a model for the signature Σ_{Word} with universe the set of finite sequences over alphabet A, and the following interpretations of symbols:

- $\epsilon_{\mathbb{W}} := \{()\}$,
- for each letter a, $a_{\mathbb{W}} := \{(a)\}$, and
- $\text{concat}_{\mathbb{W}}(s_1, s_2) := \{s_1 \cdot s_2\}$.

Patterns interpreted in model $\mathbb{W}$ define languages. ϵ is interpreted as the singleton set containing the zero-length word, each letter as the singleton set containing the corresponding single-letter sequence, and finally, concat as the function mapping each pair of input words to the singleton containing their concatenation.

We may define the empty language simply as $\bot$. The concatenation of two patterns gives the concatenation of their languages. Matching logic's disjunction allows us to take the union for any languages, while negation gives us the complement. Finally, we may define the Kleene closure of a language using the fixpoint operator—$\mu X.\, \epsilon \lor \varphi \cdot X$ gives us the Kleene closure of the language of φ.

6.2 A SHALLOW EMBEDDING OF EXTENDED REGULAR EXPRESSIONS

It is easy to define regular expressions as patterns, once we have the following notation:

$$\emptyset \equiv \bot \qquad (\varphi + \psi) \equiv \varphi \lor \psi \qquad \varphi^* \equiv \mu X.\, \epsilon \lor (\varphi \cdot X)$$

Any ERE taken *verbatim* is interpreted in model $\mathbb{W}$ as its language.

Contrast this to the MSO translation of concatenation, shown in Section 6.7, and especially of Kleene star.

Theorem 6.1. Let α be an ERE. Then $\mathcal{L}(\alpha) = |\alpha|_{\mathbb{W}}$

6.3 EMBEDDING AUTOMATA

While it is obvious how to embed expressions the representation of automata, being computational rather than logical, is less clear. Here, we define a pattern $\mathfrak{pat}_{\mathcal{Q}}$ whose interpretation is the language of a finite automaton $\mathcal{Q}$, either deterministic or non-deterministic. Crucially, this pattern captures not just the language of the automaton, but also its *structure*—as shown in Table 6.1, structural elements of the automata map to syntactic elements of the pattern—non-determinism maps to logical disjunctions; cycles map to fixpoints. This allows

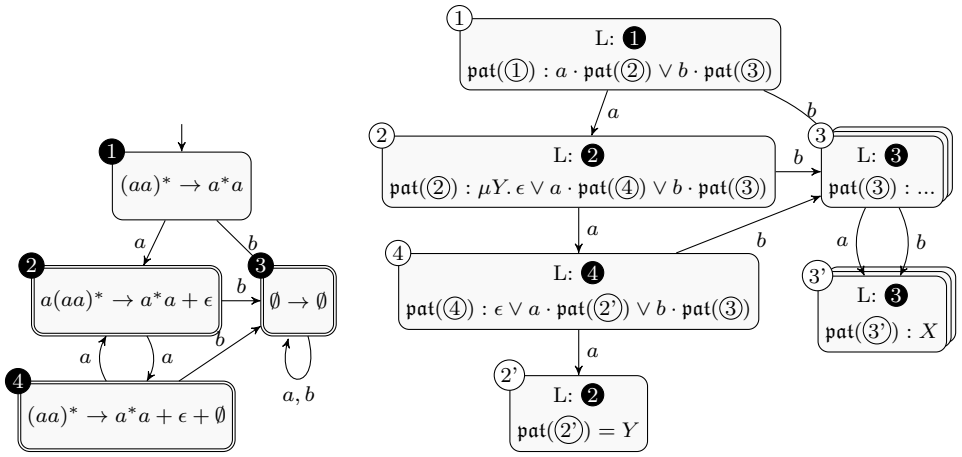


Figure 6.1: A DFA $\mathcal{Q}$ for the ERE $(aa)^* \to a^*a$, and its corresponding unfolding tree. Each node n shows its label $L(n)$, and the pattern $\mathfrak{pat}(n)$. Here $\mathfrak{pat}(③) \equiv \mu X.\, \epsilon \vee a \cdot \mathfrak{pat}(③') \vee b \cdot \mathfrak{pat}(③')$. The pattern for the automaton, $\mathfrak{pat}_{\mathcal{Q}}$, is that of the root node $\mathfrak{pat}(①)$. Observe that its structure closely mirrors that of Q. Accepting nodes include ϵ as a disjunct, whereas others do not. Starting a cycle in the graph introduces a fixpoint binder, whereas completing one employs the bound variable corresponding to that cycle. The major structural differences are due to duplicate states to allow backlinks and nodes reachable via muliple paths.

Computational aspect of $\mathcal{Q}$	Syntactic aspect of $\mathfrak{pat}_{\mathcal{Q}}$
Node n is accepting	ϵ is a subclause of $\mathfrak{pat}(n)$
Non-determinism, union of FAs	Logical union
Graph cycles	Fixpoint binder and its bound variable
Changing the initial node	Unfolding, framing

Table 6.1: Structural aspects of $\mathcal{Q}$ become syntactic aspects of $\mathfrak{pat}_Q$. This is crucial to capturing the traces of algorithms as proofs.

us to represent transformations of automata, such as making a transition, union, or complementation, as *logical manipulations* of this pattern in a proof system. This is imperative to capturing the execution of an algorithm employing these in a formal proof. To define $\mathfrak{pat}_Q$, we must first define the *unfolding tree* of the automaton $\mathcal{Q}$.

Definition 6.2. For a finite automaton $\mathcal{Q} = (Q, A, \delta, q_0, F)$, its *unfolding tree* is a labeled tree (N, E, L) where N is the set of nodes, $E \subseteq A \times N \times N$ is a labeled edge relation, and $L : N \to Q$ is a labeling function. It is the tree defined inductively:

- the root node has label q_0,
- if a node n has label q with no ancestors also labeled q, then for each $a \in A$ and $q' \in \delta(q, a)$, there is a node $n' \in N$ with $L(n') = q'$, and $(a, n, n') \in E_a$.

When $\mathcal{Q}$ is a DFA, we use n_a to denote the unique child of node n along edge a. All leaves in this tree are labeled by states that complete a cycle in the automaton. We define a secondary labeling function, $\mathfrak{pat} : N \to \mathsf{Pattern}$ over this tree.

Definition 6.3. Let (N, E, L) be an unfolding tree for $\mathcal{Q} = (Q, A, \delta, q_0, F)$. Let $X : Q \to \mathsf{SVar}$ be an injective function. Then, we define $\mathfrak{pat}$ recursively as follows:

1. For a leaf node n, $\mathfrak{pat}(n) := X(L(n))$.
2. For a non-leaf node,
   a. if n doesn't have a descendant with the same label, then:

$$\mathfrak{pat}(n) = \begin{cases} \epsilon \ \vee \bigvee_{(a,n,n') \in E} a \cdot \mathfrak{pat}(n') & \text{if } L(n) \text{ is accepting.} \\ \bigvee_{(a,n,n') \in E} a \cdot \mathfrak{pat}(n') & \text{otherwise.} \end{cases}$$

   b. if n has a descendant with the same label, then:

$$\mathfrak{pat}(n) = \begin{cases} \mu X(L(n)).\, \epsilon \ \vee \bigvee_{(a,n,n') \in E} a \cdot \mathfrak{pat}(n') & \text{if } L(n) \text{ is accepting.} \\ \mu X(L(n)). \ \bigvee_{(a,n,n') \in E} a \cdot \mathfrak{pat}(n') & \text{otherwise.} \end{cases}$$

Finally, define $\mathfrak{pat}_{\mathcal{Q}} := \mathfrak{pat}(\mathcal{R})$, where $\mathcal{R}$ is the root of this tree.

For nodes of the form (2b), we "name" them by binding the variable $X(L(n))$ using the fixpoint operator. When we return to that state we use the bound variable to complete a cycle. The use of fixpoints allows us to clearly embody the inductive structure as a pattern. Figure 6.1 shows an example of unfolding tree. The following theorem shows that this representation of automata is as expected.

Theorem 6.2. Let $\mathcal{Q}$ be a finite automaton. Then $\mathcal{L}(\mathcal{Q}) = |\mathfrak{pat}_{\mathcal{Q}}|_{\mathbb{W}}$

6.4 EMBEDDING BRZOZOWSKI'S DERIVATIVE

Besides regular languages, other important constructs may be defined using this model. Let us look at derivatives, needed to capture Brzozowski's method as a proof. The Brzozowski derivative of a language L w.r.t. a word w, is the set of words obtainable from a word in L by removing the prefix w. Defining this is quite simple in matching logic—for any word w and pattern ψ, we may define its Brzozowski derivative as the pattern $\delta_w(\psi) \equiv \exists x.\, x \wedge (w \cdot x \subseteq \psi)$.

This definition is quite interesting because it closely parallels the embedding of separation logic's magic wand in matching logic: $\varphi \mathrel{-\!\!*} \psi \equiv \exists x.\, x \wedge (\varphi * x \subseteq \psi)$. At first glance, this seems like a somewhat weak connection, but on closer inspection, magic wand and derivatives are semantically quite similar—we may think of magic wand as taking the derivative of one heap with respect to the other.

It is these connections between seemingly disparate areas of program verification that matching logic seeks to bring to the foreground. In fact, both derivatives and the magic wand generalize to a matching logic operator called *contextual implication*: $C \multimap \psi \equiv \exists \Box.\, \Box \wedge (C[\Box] \subseteq \psi)$ for any pattern ψ and application context C [99]. Using this notation, derivatives and magic wand become $\delta_w(\varphi) \equiv w \cdot \Box \multimap \varphi$ and $\varphi \mathrel{-\!\!*} \psi \equiv \varphi * \Box \multimap \varphi$ respectively. This operator has proven key to many techniques for fixpoint reasoning in matching logic, especially the derived rules (WRAP) and (UNWRAP) that enable applying Park induction within contexts [99]:

$$\vdash C[\varphi] \to \psi \qquad \underset{\text{(WRAP)}}{\overset{\text{(UNWRAP)}}{\rightleftharpoons}} \qquad \vdash \varphi \to (C \multimap \psi)$$

6.5 OTHER LANGUAGES DEFINABLE IN Γ_{Word}

While this work has focused on regular languages in Γ_{Word}, we can define more languages. For example, the context-free language $\{a^n \cdot b^n | n \in \mathbb{N}\}$ may be defined as $a^n \cdot b^n \equiv \mu X.\, \epsilon \vee a \cdot X \cdot b$. Extending this, we may define $a^n \cdot b^n \cdot c^i$, and $a^i \cdot b^n \cdot c^n$ for $n, i \in \mathbb{N}$ as the patterns $a^n \cdot b^n \cdot c^*$ and $a^* \cdot b^n \cdot c^n$ respectively. Finally, since patterns are closed under intersection we may define the *context-sensitive* language $a^n \cdot b^n \cdot c^n \equiv (a^n \cdot b^n \cdot c^*) \wedge (a^* \cdot b^n \cdot c^n)$. Extensive research has been done regarding languages definable in fragments of MSO. A corresponding effort for matching logic would be interesting. Likely, quantifiers and fixpoint operators will allow defining most computable languages.

6.6 A THEORY OF FINITE WORDS

We may use a theory Γ, a set of patterns called *axioms*, to restrict the models we consider to those in which every axiom is "true". We say a pattern φ *holds* in a model M, or that φ is *valid* in M, written $M \vDash \varphi$ if its interpretation is M under all evaluations. For a theory Γ, we write $M \vDash \Gamma$ if every axiom in Γ is valid in M. For a pattern ψ, we write $\Gamma \vDash \psi$ if for every model where $M \vDash \Gamma$ we have $M \vDash \psi$. These axioms also extend the provability relation $\vdash$ defined by the proof system, allowing us to proof additional theorems. The soundness of

Signature: ϵ, $\_ \cdot \_$, and a for each $a \in A$.

Axioms:

For each $a \in A$, For each distinct $a, b \in A$,

$\exists w.\, a = w$	(FUNC$_a$)	$a \neq b$	(NO-CONF$_a$)
$\exists w.\, \epsilon = w$	(FUNC$_\epsilon$)	$\epsilon \not\subseteq a \lor b$	(NO-CONF$_\epsilon$)
$\forall u, v.\, \exists w.\, u \cdot v = w$	(FUNC$_\bullet$)	$\forall u, v. \epsilon = u \cdot v \;\rightarrow$	
$\forall u, v, w.\, (u \cdot v) \cdot w = u \cdot (v \cdot w)$	(ASSOC)	$u = \epsilon \land v = \epsilon$	(NO-CONF$_\bullet$-1)
		$\forall x, y : \text{Letter}. \forall u, v.$	
$\forall x.\, (\epsilon \cdot x) = x$	(ID$_L$)	$x \cdot u = y \cdot v \rightarrow x = y \land u = v$	(NO-CONF$_\bullet$-2)
$\forall x.\, (x \cdot \epsilon) = x$	(ID$_R$)	$\mu X. \epsilon \lor \bigvee_{a \in A} a \cdot X$	(DOMAIN)

Figure 6.2: Γ_{Word}: A theory of finite words in matching logic. This theory is complete for proving equivalence between representations of both automata and extended regular expressions. Here, $\forall x : \text{Letter}.\, \varphi$ is notation for $\forall x.\, x \in (\bigvee_{a \in A} a) \rightarrow \varphi$.

matching logic guarantees that each proved theorem holds in every model of the theory.

Figure 6.2 defines a theory, Γ_{Word}, of finite words. The first set of the axioms in Γ_{Word}, (FUNC$_\sigma$), gives each symbol a functional interpretation: for an n-ary symbol σ, the axiom $\forall x_1, \ldots, x_n.\, \exists y.\, \sigma(x_1, \ldots, x_n) = y$, forces the interpretation σ_M to return a single output for any input. This is because element variables are always interpreted as singleton sets. Next, the (NO-CONF) axioms ensure that interpretations of symbols are injective modulo AU—they have distinct interpretations unless their arguments are equal modulo associativity of concatenation with unit ϵ. Here, $\forall x : \text{Letter}.\, \varphi$ is notation for $\forall x.\, x \in (\bigvee_{a \in A} a) \rightarrow \varphi$, i.e. we quantify over letters. The axioms (ASSOC), and (ID$_L$), and (ID$_R$) enforce the corresponding properties and allow their use in proofs. The final axiom (DOMAIN) defines our domain to be inductively constructed from ϵ, concatenation and letters. It is easy to see the standard model $\mathbb{W}$ satisfies these axioms, giving us the theorem, proved in the appendix [100]:

Theorem 6.3. $\mathbb{W} \models \Gamma_{\text{Word}}$

The rest of this section is dedicated to showing that Γ_{Word} is complete with respect to both equivalence of automata and EREs—if two automata or expressions have the same language their representations are provably equivalent.

6.7 A COMPARISON WITH OTHER EMBEDDINGS

Monadic Second-order-logic (MSO) over Words There is a well-known connection between MSO and regular languages. Büchi, Elgot, and Trakhtenbrot showed that MSO

formulae and regular expressions are equally expressive [101], [102], [103]. Moreover, the transformation from expressions to formulae and back is easily computable [104]. Models are sets of labeled positions, representing a word. The set of models that satisfy a formula give us its language—e.g. the MSO formula $\bot$ defines the empty language—no word satisfies it, while $\exists x.\, P_a x \wedge \forall y.\, x = y$ defines the language containing the word a. Here, $P_a x$ indicates the letter at position x is a. The concatenation of languages may be defined as:

$$\exists X.\, \forall y, z.\, ((y \in X \wedge z \notin X) \rightarrow y < z) \wedge [\varphi_\alpha]_{x \in X} \wedge [\varphi_\beta]_{x \notin X}$$

Here, $[\varphi]_{\psi(x)}$ denotes the relativization of the formula φ to the formula ψ, a transformation that forces it to apply to a particular subdomain of the model. The translation of Kleene star is even more complex. This connection has been used, e.g. in the verification of MSO formulae [105].

One concern about this connection between MSO and regular expressions is that the translation of expressions is quite involved, including complex auxiliary clauses and quantification, as well as the relativization transformation. Our goal here is to define a shallow embedding, rather than a translation—regular expressions are directly embedded as matching logic with minimal *representational distance*.

Salomaa's Axiomatization In [106] Salomaa provides a complete axiomatization of regular expressions that may be used to prove equivalences. This axiomatization is specific to unextended regular expressions and does not support other representations such as negations in EREs, and finite automata.

Deep Embeddings There are several existing formalizations of regular expressions and automata using mechanical theorem provers, such as Coq [107] and Isabelle [108]. To the best of our knowledge, all these formalizations use deep embeddings. In [107], the authors formalize regular expressions and Brzozowski derivatives, with the denotations of regular expressions defined using a membership predicate. Besides proving the soundness of Brzozowski's method, the authors also prove that the process of taking derivatives terminates through a notion of finiteness called *inductively finite sets*. This is something that is not likely provable in a shallow embedding like ours.

6.8 PROVING EQUIVALENCE BETWEEN ERES

We are now ready to demonstrate our proof generation method. We will use it to capture equivalence of expressions using matching logic's Hilbert-style proof system. Brzozowski's

method consists of two parts—converting an ERE into a DFA $\mathcal{Q}$, and checking that $\mathcal{Q}$ is total. Mirroring this, the proof for equivalence between EREs $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$ has two parts. First, we prove that $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \to (\alpha \leftrightarrow \beta)$—the language of $\alpha \leftrightarrow \beta$ subsumes that of $\mathcal{Q}$. Second, that $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}}$—the language of $\mathcal{Q}$ is total. We put these together using (MODUS-PONENS), giving us $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$—the EREs are provably equivalent.

6.8.1 Proving $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \to (\alpha \leftrightarrow \beta)$

To prove this, we prove a more general, inductive, lemma:

Lemma 6.1. Let n be a node in the unfolding tree of the DFA $\mathcal{Q}$ of the regular expression $\alpha \leftrightarrow \beta$, where α and β have the same language. Then,

$$\Gamma \vdash \mathfrak{pat}(n)[\Lambda_n] \to \delta_{\text{path}(n)}(\alpha \leftrightarrow \beta)$$

where,

$$\Lambda_n = \begin{cases} \lambda, \text{ the empty substitution} & \text{if } n \text{ is the root node} \\ \Lambda_p[\delta_{\text{path}(p)}(\alpha \leftrightarrow \beta)/X(p)] & \text{if } n \text{ has parent } p, \text{ and } \mathfrak{pat}(p) \text{ binds } X(p) \\ \Lambda_p & \text{otherwise.} \end{cases}$$

The substitution Λ_n provides the inductive hypothesis—as we use the (KNASTER-TARSKI) rule on each μ-binder in $\mathfrak{pat}_{\mathcal{Q}}$, it replaces the bound variable with the right-hand side of the goal. The left-hand side then becomes a disjuntion of the form $\epsilon \vee a \cdot \mathfrak{pat}(n_a)[\Lambda_{n_a}] \vee b \cdot \mathfrak{pat}(n_b)[\Lambda_{n_b}]$. We decompose the right-hand side into a similar structure using an important property of derivatives, proved in Γ_{Word}:

Lemma 6.2. For any pattern φ, $\Gamma_{\text{Word}} \vdash \varphi = ((\epsilon \wedge \varphi) \vee \bigvee_{a \in A} a \cdot \delta_a(\varphi))$

The derivatives are reduced to expressions using proved syntactic simplifications:

Lemma 6.3. For EREs α, β and distinct letters a and b, the following hold:

- $\Gamma_{\text{Word}} \vdash \delta_a(\emptyset) = \emptyset$; $\Gamma_{\text{Word}} \vdash \delta_a(\epsilon) = \emptyset$;
- $\Gamma_{\text{Word}} \vdash \delta_a(b) = \emptyset$; $\Gamma_{\text{Word}} \vdash \delta_a(a) = \epsilon$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\alpha_1 + \alpha_2) = \delta_a(\alpha_1) + \delta_a(\alpha_1)$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\alpha_1 \cdot \alpha_2) = \delta_a(\alpha_1) \cdot \alpha_2 + (\alpha_1 \wedge \epsilon) \cdot \delta_a(\alpha_2)$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\neg\alpha) = \neg\delta_a(\alpha)$;
- $\Gamma_{\text{Word}} \vdash \delta_a(\alpha^*) = \delta_a(\alpha) \cdot \alpha^*$.

6.8.2 Proving $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}}$

The next part of the proof is a bit more technical, requiring us to exploit the equivalence $\Gamma_{\text{Word}} \vdash (\mu X.\, \epsilon \vee X \cdot \varphi) \leftrightarrow (\mu X.\, \epsilon \vee \varphi \cdot X)$, and induct using the (DOMAIN) axiom. This reduces our goal to $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \cdot (\bigvee_{a \in A} a) \to \mathfrak{pat}_{\mathcal{Q}}$, a consequence of the following inductive lemma:

Lemma 6.4. Let n be a node in the unfolding tree of a total DFA $\mathcal{Q}$. Then,

$$\Gamma_{\text{Word}} \vdash \mathfrak{pat}(n)[\Theta_n] \cdot (\bigvee_{a \in A} a) \to \mathfrak{pat}(n)[\mathrm{U}_n]$$

where,

$$\Theta_n = \begin{cases} \lambda, \text{ the empty substitution} & \text{if } n \text{ is the root node} \\ \Theta_p[\Psi_p / X_{L(p)}] & \text{if } n \text{ has parent } p, \text{ and } \mathfrak{pat}(p) \text{ binds } X(p) \\ \Theta_p & \text{otherwise} \end{cases}$$

$$\Psi_p = \Box \cdot (\bigvee_{a \in A} a) \multimap \mathfrak{pat}(p)[\mathrm{U}_p]$$

$$\mathrm{U}_n = \begin{cases} \lambda, \text{ the empty substitution} & \text{if } n \text{ is the root node} \\ \mathrm{U}_p[\mathfrak{pat}(p)[\mathrm{U}_p] / X_{L(p)}] & \text{if } n \text{ has parent } p, \text{ and } \mathfrak{pat}(p) \text{ binds } X(p) \\ \mathrm{U}_p & \text{otherwise} \end{cases}$$

Again, Θ_n gives us the inductive hypothesis, this time in the form of a contextual implication. To apply it, we leverage a general property about contextual implications: $\vdash C[C \multimap \varphi] \to \varphi$, allowing us combine framing with Park induction. This gives us our main theorem, showing that our axiomatization is complete with respect to extended regular expressions:

Theorem 6.4. For any EREs α and β with the same language, $\Gamma_{\text{Word}} \vdash \alpha \leftrightarrow \beta$.

6.9 FROM EXPRESSIONS TO AUTOMATA

Our uniform treatment of automata and expressions as patterns allows us to apply Brzozowski's method not just to EREs but also to more general patterns. For example, it can be used to determinize NFAs, or take the complement, union, or intersection of DFAs. The general principle is the same as above, except instead of $\alpha \leftrightarrow \beta$, we use a pattern corresponding to the operation we wish to perform. For example, to prove that the DFA $\mathcal{Q}$ has the same language as the intersection of those of $\mathcal{A}$ and $\mathcal{B}$, we prove $\Gamma_{\text{Word}} \vdash \mathfrak{pat}_{\mathcal{Q}} \leftrightarrow (\mathfrak{pat}_{\mathcal{A}} \wedge \mathfrak{pat}_{\mathcal{B}})$. All we need is the ability to take the derivative of arbitrary fixpoint patterns enabled by the

```
def checkValid(φ: Regex, prev: set[Regex] = ∅) → bool:
    if φ ∈ prev: return True
    if ¬hasEWP(φ): return False
    return checkValid(canonicalize(δ_a(φ), prev ∪ {φ}))
       and checkValid(canonicalize(δ_b(φ), prev ∪ {φ}))
```

Figure 6.3: The algorithm instrumented to generate proofs. The `canonicalize` function reduces the pattern $\delta_a(\varphi)$ to an ERE, simplifying it to a form where choice is left-associative, and the idemopotency and unit identities have been applied.

equivalence $\Gamma_{\text{Word}} \vdash \delta_a(\mu X.\, \varphi) \leftrightarrow \delta_a(\varphi[\mu X.\, \varphi / X])$.

6.10 IMPLEMENTATION AND EVALUATION

In this section, we describe one implementation of our method—for producing proofs in Metamath Zero. Later in this document, we will describe improvements gained from generating proofs in an alternate format, one tailor-made for Matching Logic.

The algorithm implemented is shown in Figure 6.3. It recursively checks that the expression and its derivatives have the empty-word property, keeping track of when it has already visited an expression. Here, $\delta_a(\varphi)$ represents a *pattern* using the derivative notation, and not the fully simplified regular expression. This notation is simplified away in the (also instrumented) `canonicalize` function that also normalizes the choice operator to be left-associative and commutes subterms into lexicographic order, allowing the application of the idemopotency and unit axioms. This results in a canonical representation of expressions modulo similarity.

The intrumentation of successful runs of this method produces a *proof-hint*, an example of which is shown in Figure 6.4. A proof-hint is an informal artifact containing all the information necessary to produce a formal proof. It is a term defined by the following grammar.

$$\begin{aligned} \text{Node} := {} & (\texttt{backlink}\ \text{Pattern}) \\ & \mid (\texttt{der}\ \text{Pattern}, a : \text{Node}, b : \text{Node}) \\ & \mid (\texttt{simpl}\ \text{Pattern}, \text{LemmaID}, \text{Context}, \text{Subst}, \text{Node}) \end{aligned}$$

These terms are more detailed structures than unfolding trees—if we ignore the simplification nodes, we get an unfolding tree. Each `backlink` and `der` node is labeled by a regular expression, and correspond to the leaf and interior nodes of an unfolding tree. In addition,

$(\texttt{der}\ (a+b)^*,$

$a:\ (\texttt{simpl}\ \delta_a((a+b)^*),$ $\quad \texttt{der-}*,\ \square,\ \alpha \mapsto (a+b); l \mapsto a,$

$\quad (\texttt{simpl}\ (\delta_a((a+b)) \cdot (a+b)^*$ $\quad \texttt{der-}\vee,\ \square \cdot (a+b)^*,\ \ldots,$

$\quad (\texttt{simpl}\ (\delta_a(a)+\delta_a(b))(a+b)^*,$ $\quad \texttt{der-same-letter},\ (\square+\delta_a(b)) \cdot (a+b)^*,\ \ldots,$

$\quad (\texttt{simpl}\ (\epsilon+\delta_a(b))(a+b)^*,$ $\quad \texttt{der-diff-letter}(\ \epsilon+\square) \cdot (a+b)^*,\ \ldots,$

$\quad (\texttt{simpl}\ (\epsilon+\bot)(a+b)^*,$ $\quad \texttt{choice-identity-right},\ \square \cdot (a+b)^*,\ \ldots,$

$\quad (\texttt{simpl}\ \epsilon \cdot (a+b)^*,$ $\quad \texttt{concat-identity-left},\ \square,\ \ldots,$

$\quad (\texttt{backlink}\ (a+b)^*)))))))),$

$b:\ (\texttt{simpl}\ \delta_b((a+b)^*), \ldots))$

Figure 6.4: A snippet of a proof-hint for expression $(a+b)^*$ produced by the instrumentation. Most substitutions are omitted for brevity. The lemma id `der-`$*$ corresponds to the metamath theorem for $\Gamma_{\text{Word}} \vdash \delta_l(\alpha^*) = \delta_l(\alpha) \cdot \alpha^*$

`der` nodes have child nodes labeled by the patterns $\delta_a(\varphi)$ and $\delta_b(\varphi)$. Note that these are patterns and *not* regular expressions—they use the matching logic notation for derivatives, and are distinct from the fully simplified EREs. Each `simpl` node keeps track of equational simplifications needed to reduce the derivative notation, and employs associativity, commutativity, and idempotency of choice to reduce the expression into a canonical form, allowing the construction of unfolding tree to terminate. The `simpl` nodes contain the name of the simplification applied, the context in which it was applied, as well as the substitutions with which it was applied. The `LemmaID` corresponds to a hand-proved lemma in the Metamath Zero formalization.

To produce the proof of validity, proof-hints are used in three contexts. First, to produce the pattern $\mathfrak{pat}_Q$; next, to produce an instance of Lemma 6.1; and finally, to produce an instance of Lemma 6.4. For each lemma, we inductively build up the proof from two manually proven Metamath Zero theorems, one for the `backlink` node case, and another for the `der` node case. In the case of Lemma 6.1, the `simpl` nodes are ignored. In the case of Lemma 6.4 we use them to reduce the patterns to their canonical form. This is done by lifting a manually proven theorem corresponding to the `LemmaId` into the context, and applying the substitution, all supplied by the `simpl` node.

Benchmark	Nodes	`.mmb` size	Gen. time	Check time
Manual Lemmas		307		3
$(a+b)^*$	3	2	64	3
$a^{**} \to a^*$	5	4	82	3
$(aa)^* \to a^*a + \epsilon$	9	15	179	3
$\neg(\top \cdot a \cdot \top) + \neg(b^*)$	5	5	90	3
$\mathsf{match}_l(2)$ / $\mathsf{match}_l(8)$	19 /43	13 / 266	273 / 27483	3 / 4
$\mathsf{match}_r(2)$ / $\mathsf{match}_r(8)$	19 /43	13 / 228	337 / 21085	3 / 4
$\mathsf{eq}_l(2)$ / $\mathsf{eq}_l(8)$	13 /37	15 / 446	374 / 91661	3 / 5
$\mathsf{eq}_r(2)$ / $\mathsf{eq}_r(8)$	13 /37	15 / 330	368 / 31489	3 / 5

Table 6.2: Statistics for certificate generation. Sizes are in KiB, times in milliseconds. We show the unfolding tree nodes, proof size, generation and checking time.

6.10.1 Trust Base

Our trust base consists of the Metamath Zero formalization of matching logic proof system, including its syntax and meta-operations for its sound application such as substitution, freshness (272 lines); the theory of words instantiated with $A = \{a, b\}$, (13 lines); and the Metamath Zero proof checker, `mm0-c`. Each of these are defined in `.mm0` files in our repository [109], [110]. From these, we prove by hand 354 supporting general theorems and 163 specific to Γ_{Word}, such as Lemmas 6.1 and 6.4, and those about derivatives and their simplification.

6.10.2 Evaluation

We have evaluated our work against handcrafted tests, as well as standard benchmarks for deciding equivalence presented in [111]. Some statistics are shown in Table 6.2. Each $\mathsf{match}_{\{l,r\}}(n)$ test, by [112], is an ERE asserting that a^n matches $(a + \epsilon) \cdot a^n$, that is, $a^n \to (a + \epsilon) \cdot a^n$. Here α^n indicates n-fold concatenation of α, with the l version using concatenation from the left, and the r version on the right. That is, α^3 may be either $((\alpha \cdot \alpha) \cdot \alpha)$ or $(\alpha \cdot (\alpha \cdot \alpha))$. Each $\mathsf{eq}_{\{l,r\}}(n)$ test, by [113], checks if a^* and $(a^0 + \cdots + a^n) \cdot (a^n)^*$ are equivalent. We also include property testing using the Hypothesis testing framework. We randomly generate an ERE α, and check that $\alpha \to \alpha$. Our procedure does not optimize for this, so it allows testing correctness for a variety of expressions, augmenting the few handcrafted ones, and the structurally monotonous benchmarks.

6.10.3 Performance

In this work, our goal was to prove that this process is feasible—we have not focused on performance. In fact, we find the performance numbers here are quite poor. There are a number of reasons for this.

First, we made some poor implementation choices with reference to instrumentation. The prototype uses Maude and its meta-level to produce the instrumentation. While Maude's `search` command *collects* all the information needed for the proof hint, it does not make it accessible. This forced us to repeatedly enter and exit the meta-level to collect this information, bringing the running time of, e.g., $\mathsf{match}_l(8)$ to 27 seconds, compared to 3ms when implemented idiomatically.

Another reason is that we targeted simplicity, rather than even the most basic optimizations. For example, when multiple identical nodes occur in an unfolding tree, we do not reuse the subproofs for identical notes in the derivative tree, and instead re-prove the result each time. This causes a significant blow up in proof size. We believe that a relatively small engineering effort would greatly improve performance both in terms of proof size and generation time.

Another issue is that handling machine generated proofs is not one of Metamath Zero's design goals. It is intended as a human-readable language, for human-written proofs. We would rather output a succinct binary representation of proofs. Although Metamath Zero does allow generation of proofs directly in the `mmb` format, at the time of writing, this wasn't the most welcoming option as we were warned that the format was subject to change, and had not been formally specified.

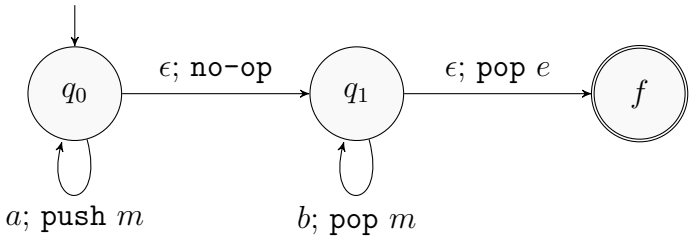
6.11 FUTURE APPLICATIONS

The technique shown in the previous sections, of using logical analogues of computational structures, may be extended beyond finite automata. In this section, we will consider push-down automata and Büchi automata and tree automata and its application to order-sorted theories.

6.11.1 Push-Down Automata

Push-down automata augment finite state machines with a stack that may be manipulated at each push. For example, the automaton shown recognizes the language $a^n b^b$—the language segments of only as followed by an equal length segment of only bs. It operates over an input

alphabet $\{a, b\}$ and a stack alphabet $\{e, m\}$. The stack is initialized containing just the letter e, and the initial state is q_0.



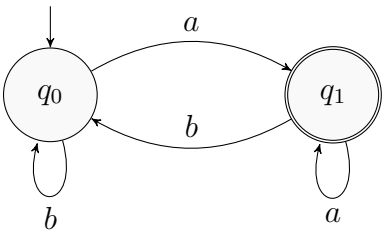
As mentioned previously, while not regular, this particular language is easily representable in Γ_{Word} as the matching logic pattern $\mu X.\, a \cdot X \cdot b$. This pattern, however, does not meet the criterion of being a structural analog of the automata shown. Could we get such a representation that would allow us to certify algorithms that manipulate push-down automata?

It seems that with a small amount of infrastructure this is indeed possible. If we augment Γ_{Word} with pairs, we may capture the automaton with this pattern:

$$\mathsf{first}\Big(\mu Q_0.\, \Big(\bullet(a, \mathsf{push}\ m, Q_0) \vee \Big(\mu Q_1.\, \bullet(b, \mathsf{pop}\ m, Q_1) \vee \bullet(\epsilon, \mathsf{pop}\ e, \langle \epsilon, e \rangle)\Big)\Big)\Big)$$

6.11.2 Büchi Automata

Büchi automata are an extension to finite automata that deals with ω-words, strings of infinite length, instead of finite strings. They are therefore very useful in the verification of LTL formulae, which deal with linear execution traces—easily represented as strings of program states. Indeed, many procedures for checking LTL formulae are via Büchi automata.



The above automaton accepts ω-words in which a appears infinitely often. In a theory of ω-words, $\Gamma_{\omega\text{Word}}$, the following pattern would give us its language. This theory is quite similar to Γ_{Word}, but would include a different (DOMAIN) axiom employing ν.

$$\mu L_0.\, \nu G_0.\, b \cdot L_0 \vee a \cdot (\mu L_1.\, \nu G_1.\, a \cdot G_1 \vee b \cdot G_0)$$

Part IV

Efficient Proof Generation

As mentioned in the introduction, to gain assuredness of the correctness of the $\mathbb{K}$ framework, we propose instrumenting each of its tools, so that we can generate formal proofs for each task. To do saw, we will augment the framework with an *untrusted* proof generator that consumes this instrumentation, and a *trusted* proof checker that verifies the correctness of the generated proof.

The *trusted* proof checker takes as input a proof unit, while the *untrusted* proof generator produces one. A proof unit is a set of axioms called a theory, a set of claimed theorems, and a representation of a proof for those claims, and verifies that the proof supplied actually proves the claims.

$$\underbrace{\text{List}\,\{\text{Pattern}\}}_{\text{Theory}} \times \underbrace{\text{List}\,\{\text{Pattern}\}}_{\text{Claims}} \times \text{List}\,\{\text{Proof}\}$$

In matching logic, the statement

$$\Gamma_{\text{IMP}} \vdash \langle\langle \text{SUM}, s \mapsto 0; n \mapsto 10\rangle\rangle \Rightarrow^* \langle\langle \text{noop}, s \mapsto 55; n \mapsto 0\rangle\rangle$$

where SUM $\equiv$ `while(n > 0) { s = s + n; n--; }`, states the result of execution of the `SUM` program. This would be encoded as a proof-unit with Γ_{SUM} as the list of axioms, a single claim with the right-hand side of the turnstile, and an encoding for the proof of the claim. We allow multiple claims and proofs in a proof unit so we may gain efficiencies by proving several claims without having to re-encode the axioms again and again. In addition, they may share common lemmas.

Since programming languages are complex beasts, proof generation and checking needs to be reasonably efficient. Another consideration is streaming—can we generate proofs for long-running executions? We obviously cannot keep an entire proof in memory for a program executions of arbitrary lengths. We must therefore be able to emit proofs without accumulating state, and eventually running out of memory. In this part, we will describe how we deal with some issues observed with [5] in terms of performance and scalabilty of proof generation and checking.

- A first issue we faced with [5] was the scalability of the proof-generation architecture. The implementation constructed the entire proof in memory before outputting it to disk. This lead to emitting the proof to disk.
- Proof hint generation is that the instrumentation had large holes in them. In particular, it did not produce proofs for functional simplifications. We worked with the K team to develop a proof-hint format that serialized most aspects of execution (some holes in

the proof format are hooked operations such as those over integers, sets, and maps).

- One of the goals of the Pi Squared **?** is to be able to certify the validity of these proofs cryptographically using zero-knowledge techniques[114]. Proof checking in these formats is orders of magnitude slower than native execution so another goal was to make proof checking as efficient as possible.

The rest of this part will explain how we deal with these issues, starting with the proof checker, then moving on to proof generation. For proof generation, we will use regular expressions and finite automata example in Chapter 6 as test case before moving on to proof generation for the more complex proof generation for $\mathbb{K}$'s concrete execution.

CHAPTER 7: A PROOF LANGUAGE FOR MATCHING LOGIC

In this chapter, we talk our efforts towards a more efficient representation of matching logic proofs. As mentioned previously, this is necessary to represent the large proofs needed for proofs of execution.

7.1 WHY ANOTHER PROOF FORMAT?

Previous attempts of generating proofs of concrete execution in Metamath ran into scalability and performance issues. There are two major reasons for this, both stemming from us not using Metamath as it is intended to be used.

First, *Metamath is intended as a human-readable language for hand-written proofs*, perhaps with a small amount of mechanical assistance—as opposed to large machine generated ones. This means that each Metamath file is highly structured, and requires somewhat involved parsing. For example, each lemma must include a human readable statement of each lemma proved.

While this may seem harmless at first, it forces checkers to implement a second mechanism for constructing patterns. The other, more fundamental, mechanism is used to construct *provably* well-formed patterns used in proofs and instantiations of lemmas. Further, it is impossible to share these patterns, or sub-patterns of them between lemmas since they are rigidly scoped to a single statement.

The rigid structure of Metamath files means that lemmas of interest to a localized section of the proof must be kept around for the entire duration of the proof's lifetime, despite it being known at generation time that it will not be re-used. This can cause scalability issues for long-running proofs.

Another example of the awkwardness caused by this human-oriented format is the Metamath's compressed proof sub-format—it encodes proofs as an RPN using a hybrid base-20 and base-5 binary to ASCII encoding scheme [115]. This is a kludge aimed at getting somewhat better performance for larger proofs while at the same time keeping the proof in an ASCII format.

This introduce a mechanism for sharing subproofs, but it is somewhat limited—one can only use it to share subproofs within the proof of a single lemma. This is especially a problem when considering the sharing of notations between the statements of different axioms, and claims.

In fact, each formalization must introduce such a mechanism for the definition of notation since it is not built into Metamath. Notation is typically defined as trusted axioms in Metamath, (both in the Metamath matching logic formalization, and `set.mm` a formalization of ZFC in Metamath) and make it easy to introduce ambiguities in their definition. The `set.mm` Metamath database among others deal with this using external tools to make sure notation and grammar are well-formed. Unfortunately, this makes these tools part of our trust base. A matching logic formalization in Metamath would additionally need to verify other aspects of well-formedness such as that set variables only occur positively in defined notation to ensure soundness.

Second, *Metamath is built to be logic-agnostic*. This means that each many operations that are taken for granted in matching logic must be explicitly and verbosely proved. Some experiments show that a large proportion of the proof are these uninteresting deterministic checks that must be repeatedly checked again and again. These could be easily handled internally by the prover at little cost to complexity.

Listing 7.1: The anatomy of a Metamath proof

```
...
$( Matching Logic Axiomatization $)
$( ph0, ph1, and ph2 are metavariables representing well-formed patterns. $)
$v ph0$. ph0-is-pattern $f #Pattern ph0 $. ph0-is-pattern $f #Pattern ph0 $.
$v ph1$. ph1-is-pattern $f #Pattern ph1 $. ph1-is-pattern $f #Pattern ph1 $.
$v ph2$. ph1-is-pattern $f #Pattern ph2 $. ph1-is-pattern $f #Pattern ph2 $.
$( Given two patterns (ph0 \limplies ph1) is a well-formed pattern. $)
imp-is-pattern $a #Pattern ( \imp ph0 ph1 ) $.
$( Relevant propositional axioms schemas $)
proof-rule-prop-1 $a |- ( \imp ph0 ( \imp ph1 ph0 ) ) $.
proof-rule-prop-2 $a |- ( \imp ( \imp ph0 ( \imp ph1 ph2 ) )
                                ( \imp ( \imp ph0 ph1 )
                                       ( \imp ph0 ph2 ) ) ) $.
$( Modus Ponens $)
${
    proof-rule-mp.0 $e |- ( \imp ph0 ph1 ) $.
    proof-rule-mp.1 $e |- ph0 $.
    proof-rule-mp   $a |- ph1 $.
$}
$( Proof of reflexivity of implication, $\phi_0 \limplies \phi_0$ $)
imp-reflexivity $p |- ( \imp ph0 ph0 ) $=
     ( imp-is-pattern proof-rule-prop-1 proof-rule-mp proof-rule-prop-2 )
     AAABBAABAAABABBAAABBAABBAAABAEAAABCDAACD $.
     $( ASCII representation of proof RPN $)
```

```
$( A = ph0-is-pattern
   B = imp-is-pattern  (2 patterns as args)
   C = propositional axiom 1 (2 patterns as args)
   D = ModusPonens (2 patterns and 2 proofs as args)
   E = propositional axiom 2 (3 patterns as args)
$)
```

TODO: Annotate this https://tex.stackexchange.com/questions/657375/how-to-use-listing-tikz-and-tikzmark-packages-together-to-annotate-source-code

- `$p` Statement includes human readable statement of proof.] This is as involved as a proof of well-formedness, but cannot be shared with other parts of the proof. It is yet another mechanism for constructing patterns
- Metamath's compressed proof are a bolted on hack to improve the efficency of the format.

7.2 THE TAML AND BAML PROOF FORMATS

The *binary format for applicative matching logic* (BAML) aims to solve some of these problems by giving up on human-readabilty and off-loading those objectives to secondary tools such as pretty-printers and DSLs for proof generation. In addition, we define an isomorphic *textual format for applicative matching logic* (TAML) to help specifying the BAML format in this document.

These may be thought of as a reverse polish notation (RPN) for constructing Gentzen-style proof trees for matching logic. The construction of patterns and the proofs of theorems are treated uniformly—they are both something that needs to be "proved"—and we use the same mechanisms for both. This leads to some nice simplifying congruences between the two—for example, notations and lemmas become instances of the same mechanism—the ability to share subproofs.

7.3 A $\mathbb{K}$ SPECIFICATION FOR TAML

In the rest of this sub-section we define a K specification of the TAML format. Being a textual format it is easier to give a formal semantics to. In the following section, we will describe the binary encoding of this format.

We have operations or `Instruction`s for each constructor in matching logic's syntax.

```
syntax Instruction ::=
        "Bot"         | "Implies"    // Logic
      | "Symbol" Int  | "App"        // Structure
      | "EVar" Int    | "Exists" Int // First-order quantification
      | "SVar" Int    | "Mu" Int     // Fixpoint quantification
```

For example, the expression `Bot Bot Implies` constructs the matching logic pattern $\bot \to \bot$. Notice that some of these constructors are parametric, taking a symbol or variable identifier. We use integer identifiers for these—it is the responsibility of external tool to cross-reference this with the relevant metadata to produce a human readable identifier for pretty printing.

To make proof generation practical, we enrich the syntax of matching logic with meta-operators. The `MetaVar n` instruction allows us to construct *pattern-schemas* that may then be instantiated using the `Instantiate` instruction. Since we cannot perform substitutions over meta-variables, we must also have an explicit syntactic construct to represent them.

```
syntax Instruction ::=
      "MetaVar" Int ConstrList
    | "ESubst" Int | "SSubst" Int
    | "Instantiate" IntList
```

In addition, we may constrain how each `MetaVar` is instantiated—in particular, we may add constraints on how set and element variables may occur. It is convenient to have a `ConstrName` for when a variable is not constrained. For example, we may construct the well-formed pattern schema pattern $\mu X_0.\, \varphi_0$, where X_0 may only occur positively, using the instructions:

```
MetaVar 0 positive=0
Mu 0
```

The constraints `eFresh` and `sFresh` require that an element or set variable do *not* occur free in any instantiation; `positive` and `negative` require that a *set* variable occurs either positively or negatively in an instantiation (i.e. under the left-hand side of an even or odd number of implications respectively); and `app_ctx_hole` requires that an element variable occurs as a hole in an application context—i.e. exactly once, and only under applications.

For each meta-variable–variable pair, we allow up to one constraint. This is sufficient because the conjunct of every pair of constraints either has an equivalent single constraint (e.g. if a set variable satisfies both `positive` and `negative`, then equivalently it satisfies `sFresh`), or unsatisfiable (no element variable can satisfy both `app_ctx_hole` and `eFresh`).

```
syntax ConstrList ::= List{Constr, ""}
syntax ConstrName ::= SConstrName | EConstrName
```

```
syntax EConstrName ::= "eFresh" | "app_ctx_hole"
syntax SConstrName ::= "sFresh" | "positive" | "negative"
syntax Constr ::= ConstrName "=" IntList
```

Next, we have instructions for each of matching logics inference rules and axiom schemas.

```
syntax Instruction ::=
      "Prop1" | "Prop2" | "Prop3" | "ModusPonens"
    | "Generalization" Int
    | "Framing" Int
    | "PreFixpoint" Int | "KnasterTarski" Int
```

The `Save` and `Load n` instructions allow sharing common sub-expressions of both patterns and proofs—it generalizes the RPN to allow representing acyclic graphs rather than just trees. These two instructions give us a mechanism to define not just lemmas, but also notation and let-expressions.

```
syntax Instruction ::= "Load" Int | "Save"
```

The `Publish` instruction allows marking certain patterns and theorems as of some kind of special interest—for example, it allows us to mark them as axioms, or as being a proof of one of the main claims made by this proof unit.

```
syntax Instruction ::= "Publish"
```

The `Version` instruction lets us check that the proof-checker and generator are in agreement regarding the version of this protocol supported.

```
syntax Instruction ::= "Version" Int "." Int "." Int
```

Finally, the `STOP` instruction is for debugging, allow us to stop execution in its tracks and inspect the internal state of the checker.

```
syntax Instruction ::= "STOP"
```

The proof file is broken up into three sections, each containing a list of instructions. the *gamma* section defines a list of axioms or assumptions, the *claims* section defines the claims or theorems that will be proved, and finally, the *proof* section proves the claims using the assumptions. The gamma and claims sections may not use the instructions for axioms or inference rules. The proof section may use any instruction.

```
syntax Pgm ::= SectionGamma SectionClaims SectionProof
syntax SectionGamma ::= "section" "gamma" Instructions
syntax SectionClaims ::= "section" "claims" Instructions
```

```
syntax SectionProof ::= "section" "proof" Instructions
syntax SectionName ::= "gamma" | "claims" | "proof"
syntax Instructions ::= List{Instruction, ""}
```

7.3.1 Data Structures

The proof RPN described earlier may be thought of as an expression for building matching logic patterns and proofs as a abstract data type. Since, for proof terms, we only care about the final conclusion of the proof and not the internal structure of the proof, we may only keep track of this conclusion.

```
syntax Term ::= Pattern | Proved(Pattern)
```

We must keep track of the constraints over meta-variables for when they are instantiated, and make sure that they remain consistent accross multiple occurances [1]. This is done my wrapping each meta-pattern's AST in a context that with constraints for each meta-variable in the pattern.

```
syntax Pattern ::= "<" ast: PatternAST "," constrCtx: ConstrCtx ">"
```

The AST follows the standard matching logic grammar, enriched with the various meta-constructs with the exception of `Instantiate`:

```
syntax PatternAST ::=
      Bot() | Implies(PatternAST, PatternAST)
    | Symbol(Int)    | App(PatternAST, PatternAST)
    | EVar(Int)      | Exists(Int, PatternAST)
    | SVar(Int)      | Mu(Int, PatternAST)

    | MetaVar(Int)
    | ESubst(PatternAST, var: Int, plug: PatternAST)
    | SSubst(PatternAST, var: Int, plug: PatternAST)
```

For now, we eagerly evaluate instantiations, treating it as a function over ASTs. However, for performance, it is likely better evaluate it lazyily. This is of special importance in the case of instantiation of and matching over notation—a frequently peformed operation, and essential to keeping proof sizes manageable. This can be done by making the following `Instantiate`

[1] It is not strictly necessary that we make sure that the set of constraints is consistent. For example, a meta-pattern in which an element must both be fresh and appear as an application context hole cannot be instantiated to a concrete pattern. However, disallowing this case simplifies the implementation by allowing us to only keep track of the strictest requirement for each variable-meta-variable pair—one always exists.

construct a constructor, and having operations such as matching commute it with the other operations on an as-needed basis—these would be restatements of the equational rules below. This will come at the cost of more expensive equality checking. See Section 7.6 for a more detailed discussion.

```
syntax PatternAST ::= Instantiate(PatternAST, IntList, PatternList) [function]
rule Instantiate(Bot(), _, _) => Bot()
...
```

The second component of a `Pattern` is the constraint context. A `ConstrCtx` is a map from meta-variable IDs to the constraints on its instantiations. The `MVConstr` contains the constraints for a single meta-variable, mapping element variables and set variables to constraints for their instantiation. These constraints include freshness of element and set variables, positivity and negativity of occurances of set variables (for constructing well-formed fixpoint patterns), and one for checking that an element variable occurs as a hole in a application context.

```
syntax ConstrCtx ::= Map // from Int to MVConstr
syntax MVConstr ::= "eConstr:" Map "," "sConstr:" Map
```

7.3.2 Checker State

During verification, the checker interprets the the proof stored in the `<k>` cell, initialized by the `$PGM` variable. It uses `<stack>` cell as working space for building patterns and the conclusions of proofs. The `<memory>` cell is used to store sub-patterns and proofs for reuse later (e.g. notation or lemmas). Axioms admitted in the `gamma` section are also introduced here when interpreting the `proof` section.

```
configuration
  <k> $PGM:Pgm  </k>
  <stack> .TermList </stack>
  <memory> .List </memory>
  <claims> .PatternList </claims>
  <currSection> .MaybeSectionName </currSection>
  <exit-code exit=""> 1 </exit-code>

syntax MaybeSectionName ::= SectionName | ".MaybeSectionName"
syntax MaybeMetaPattern ::= Pattern     | ".MaybeMetaPattern"
```

7.3.3 Push & Pop

Some instructions have a two-stage execution. For example, they may need to check meta-constraints, before manipulating the stack. `#push[]` and `#pushProved[]` allow us to simplify this two stage execution using strictness.

7.3.4 Pattern Construction

The `Bot` instruction is the simplest: it pushes the pattern with AST `Bot()` and the empty context to the stack.

```
rule [bot]: <k> Bot => #push[Bot(), .Map] ... </k>
```

Besides being parametric in the identifier, `Symbol`, `EVar`, and `SVar` are identical.

```
rule [symbol]: <k> Symbol I => #push[Symbol(I), .Map] ... </k>
rule   [evar]: <k> EVar I   => #push[EVar(I),   .Map] ... </k>
rule   [svar]: <k> SVar I   => #push[SVar(I),   .Map] ... </k>
```

The `Implies` instruction removes two patterns from the stack and constructs implication from the ASTs. It merges their contexts before pushing the built pattern to the stack. The `App` instruction is analogous.

```
rule [implies]:
    <k> Implies => #push[Implies(Left, Right),  #mergeCtxs[LCtx, RCtx]] ... </k>
    <stack> <Right, RCtx>, <Left, LCtx>, Stack => Stack </stack>
rule [app]:
    <k> App     => #push[App(Left, Right),      #mergeCtxs[LCtx, RCtx]] ... </k>
    <stack> <Right, RCtx>, <Left, LCtx>, Stack => Stack </stack>
```

`Exist` is quite simple, constructing the pattern while inheriting the constraints of the sub-pattern.

```
rule [exists]:
    <k> Exists I => #push[Exists(I, P), Ctx] ... </k>
    <stack> <P, Ctx>, Stack => Stack </stack>
```

When constructing a fixpoint binder, we need to assert that the bound variable only occurs positively.

```
rule [mu]:
     <k> Mu I => #check[positive, I, P] ~> #push[Mu(I, Pat), Ctx]... </k>
     <stack> (<Pat, Ctx> #as P), Stack => Stack </stack>
```

Before constructing a meta-variable, the constraint context must be constructed from the constraints listed as part of the `Instruction`.

```
rule [metavar]:
    <k> MetaVar I Constrs
     => #push[MetaVar(I), #makeCtx[I, Constrs, eConstr: .Map, sConstr: .Map]] ...
    </k>
```

With element and set substitutions we must first merge their constraints before constructing the pattern.

```
rule [esubst]:
    <k> ESubst N => #push[ESubst(Pattern, N, Plug), #mergeCtxs[PlugCtx, PatternCtx]] .
    <stack> (<Pattern, PatternCtx>, <Plug, PlugCtx>, Stack) => Stack </stack>
rule [ssubst]:
     <k> SSubst N => #push[SSubst(Pattern, N, Plug), #mergeCtxs[PlugCtx, PatternCtx]]
     <stack> <Pattern, PatternCtx>, <Plug, PlugCtx>, Stack => Stack </stack>
```

7.3.5 Instantiation

Instantiation is the only construct listed here that does not have an analog in the Metamath formalization of AML. This is because it is built into Metamath and so does not need to be formalized explicitly. This has a somewhat complex semantics because it is variadic—instantiation corresponds to simultanuously substituting a set of metavariables for other (possibly meta-)patterns. We also ensure that the meta contexts of all these patterns are consistent.

```
rule <k> Instantiate IndexList
      => ( #gatherPlugs[IndexList; getCtx(P); .PatternList; .ConstrCtxList]
        ~> #buildInstantiate P IndexList
         )
         ...
     </k>
     <stack> P, Stack => Stack </stack>
```

Here, `#gatherPlugs` collects the values to be substituted for each of the meta-variables in the meta-pattern, merging their contexts; `#buildInstantiate` then uses these to pass to `Instantiate` along with the merged context.

7.3.6 Proof Rules

7.3.6.1 Propositional

The three propositional axioms are quite similar to the simpler instructions—they simply push something onto the stack. Although here, they push a `Proved` rather than a pattern. They must also initialize the constraint context.

```
rule <k> Prop1
      => #pushProved[ Implies(MetaVar(0),
                              Implies(MetaVar(1), MetaVar(0)))
                    , 0 |-> (eConstr: .Map, sConstr: .Map)
                      1 |-> (eConstr: .Map, sConstr: .Map)
                    ]
         ...
     </k>
```

Inference rules, such as modus ponens are, however quite distinct, deconstructing conclusions to construct a new one. We must also be sure to merge the contexts of the supplied hypothosis.

```
rule <k> ModusPonens => #pushProved[Psi, #mergeCtxs[MajCtx, MinCtx]] ... </k>
     <stack> Proved(<Phi, MinCtx>), Proved(<Implies(Phi, Psi), MajCtx>), Stack => Stac
```

7.3.7 First-Order

```
rule <k> Generalization Binder
      => #check[eFresh, Binder, <Psi, Ctx>]
      ~> #pushProved[Implies(Exists(Binder, Phi), Psi), Ctx]
         ...
     </k>
     <stack> Proved(<Implies(Phi, Psi), Ctx>), Stack => Stack </stack>
```

7.3.8 Propagation and Framing

```
rule [framing]:
     <k> Framing I
      => #push[ Implies(ESubst(MetaVar(0), I, Phi), ESubst(MetaVar(0), I, Psi))
              , #mergeCtxs[Ctx, 0 |-> eConstr: (I |-> app_ctx_hole), sConstr: .Map]
              ] ...
     </k>
     <stack> Proved(<Implies(Phi, Psi), Ctx>), _Stack </stack>
```

7.3.9 Fixpoint Reasoning

```
rule <k> PreFixpoint Binder
      => #pushProved[ Implies( SSubst(MetaVar(0), Binder, Mu(Binder, MetaVar(0)))
                             , Mu(Binder, MetaVar(0))
                             )
                    , 0 |-> ( eConstr: .Map, sConstr: (Binder |-> positive) )
                    ]
         ...
     </k>

rule <k> KnasterTarski Binder
      => #check[positive, Binder, C]
      ~> #pushProved[Implies(Mu(Binder, CPat), R), #mergeCtxs[PCtx, CCtx]]
         ...
     </k>
     <stack> Proved(<Implies(L, R), PCtx>), (<CPat, CCtx> #as C), Stack => Stack </sta
  requires applySSubst(CPat, Binder, R) ==K L
```

7.3.9.1 Sub-Pattern, and Sub-Proof Sharing

The `Save` and `Load` instructions allow the RPN to represent a DAG instead of just a tree. They allow us to save already constructed `Pattern`s and `Proved`s to memory and retrieve them when they need to be re-used. This may be used to implement both notation and lemmas.

```
rule <k> Save ⇒ .K ... </k>
     <stack> T:Term, _Stack </stack>
     <memory> ... .List ⇒ ListItem(T) </memory>
rule <k> Load N ⇒ .K ... </k>
     <stack> Stack ⇒ {Memory[N]}:>Term, Stack </stack>
     <memory> Memory:List </memory>
  requires N >=Int 0 andBool N  <Int size(Memory)
```

Together, `Load`, `Save`, and `Instantiate` allow us to implement notation as well as lemmas—`Load` and `Save` give us re-use, while instantiation allows us to extend this to schemas.

7.3.9.2 Axiom and Claim Management

The `Publish` instruction behaves differently in different sections. In the gamma section, it pushes the top of the stack to the list of axioms.

```
    rule <k> Publish => Save ... </k>
```

```
      <currSection> gamma </currSection>
      <stack> (Axiom => Proved(Axiom)), _Stack </stack>
```

Similarly, in the claims section, it pushes the top of the stack to the list of axioms.

```
rule <k> Publish => .K ... </k>
     <currSection> claims </currSection>
     <stack> Claim, _Stack </stack>
     <claims> Claims => Claim, Claims </claims>
```

Finally, in the proof section, if the final claim has been proved, it removes it from the claims list.

```
rule <k> Publish => .K ... </k>
     <currSection> proof </currSection>
     <stack> Proved(Claim), Stack => Stack </stack>
     <claims> Claim, Claims => Claims </claims>
```

7.4 BINARY REPRESENTATION

In BAML, the binary representation of this format, each token as well as identifiers for variables are mapped to a 8-bit numbers, while memory addresses are mapped to 16-bit numbers. The lists for meta-variable constraints are prefixed by their length, and must all be present in the following order `e_fresh`, `s_fresh`, `positive`, `negative`, and `app_ctx_holes`. Each proof section is in a separate file. This simple encoding allows us to easily interpret the proof with negligible overhead for parsing.

7.5 EVALUATION AND IMPLEMENTATIONS

This $\mathbb{K}$ specification serves as an executable specification of the TAML format. In addition, we have have implemented a checker for the BAML format in Rust. In this section, we evaluate proof validation in these formats against equivalent Metamath proofs, and where available against Metamath Zero proofs. Note that the comparison against Metamath isn't quite a one-for-one comparison While the claims for the proofs generated are identical, their proofs are produced through quite different implementations.

7.6 FUTURE WORK

While our benchmarks already outperform Metamath there is much opportunity for improvement. These generally deal with two concerns—efficiency and usability. At their same time, their implementation is at odds with a vital concern of a proof checker—simplicity.

7.6.1 Efficient Notations

This specification implies that instantiation of patterns and notation must be performed eagerly, reducing them to their base matching logic constructs. This isn't necessary. Instantiation may instead be performed lazily, only (and perhaps) partially simplifying them when it is necessary to match or check equality between patterns.

Let us consider the notational sugar for logical equivalence, $\leftrightarrow$. As is typical in many a logic, this is defined as a conjunction of implications:

$$\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \psi)$$

If evaluating instantiations eagerly, we must recursively substitute φ and ψ through the conjunction and implications. This is a high cost for a function as common as instantiation. The trade-off is that checking equality becomes more expensive.

7.6.2 Alpha Renaming or Nameless Representations

In this proof language, we treat variables as concrete patterns. This introduces a fair amount of overhead when we needs several alpha-equivalent instances of the same variable. The Metamath formalization of matching logic handles this by treating variables as meta-variables—giving matching modulo alpha-equivalence for free.

We may take a similar approach to get these benefits—we may add an additional constraint to `MetaVar`s, requiring them to be instantiated only with variables. We would also need to extend the `Mu` and `Exists` instructions to take `MetaVar` parameters.

An alternate approach is to avoid the problem completely by using a nameless representation for bound variables—for example we may use a de Bruijn representation—where alpha-equivalence becomes representational equivalence, at least for closed patterns. Another option to consider is pointer equivalence—two variables (element, set, meta-) are considered equal, only if they are they were constructed using the same instructions. That is, the variable

was constructed once, `Save`d, and then retrieved using the `Load` command for other instances.

7.6.3 Determinization and caching of meta-constraints checks

One of the motivations of this proof format was reducing the verbosity of Metamath proofs. One of the ways this was done was by internalizing the various checks for meta-constraints, such as positivity, that are easily checkable via simple decision procedures. We believe we went too far with this, and the ideal balance is somewhere between the Metamath formalization does and this specification.

The problem is that many of these procedures involve backtracking. For example, checking `app_ctx_hole` requires the prover to guess whether the variable occurs under the left- or right-hand side of each application.

Another issue with not representing these checks explicitly is that we cannot cache them for re-use.

7.6.4 Proof Composition

Proof composition is vital for module generation of proofs. Language specific lemmas or even program specific ones may be pre-generated to improve the speed of proof generation by not needing to re-check previously proved lemmas. As described here, the proof language does not deal with modularity or proof composition. This is expected to be handled by a complementary tool. Each proof unit in this format may be thought of as proving the set of sequents: $\{\Gamma \vdash \varphi \mid \varphi \in \Phi\}$ where Γ is the set of patterns constructed in the gamma section, and Φ is the set constructed in the claims section. Let us abbreviate this as $\{\Gamma \vdash \Phi\}$.

The proof format above has an interesting property, that if we concatenate the instructiions in the axioms section with the claims sections (either as text in TAML or as raw bytes in BAML), we get valid instructions that may be placed in the gamma section of another proof unit. This allows makes it easy to compose proof-units using a weakened form of the cut rule from sequent calculus using the simple operation of string concatenation.

$$\frac{\{\Gamma \vdash \Delta\} \qquad \{\Gamma \cup \Delta \vdash \Phi\}}{\{\Gamma \vdash \Phi\}}$$

This form of composition is particularly useful in the Zero-Knowledge context, where execution after partial reading of inputs can be paused and resumed with multiple inputs.

Due to the `Save` and `Load` other forms of composition are more tricky and would need us to fully interpret the instructions. However, adding a separate section for defining notation in which `Save` and `Load` are allowed, `Publish` is not; and disallowing `Save` in the gamma and claims sections will allow additional modes of string-based composition, such as:

$$\frac{\{\Gamma \vdash \Delta\} \qquad \{\Delta \vdash \Phi\}}{\{\Gamma \vdash \Phi\}} \qquad \frac{\{\Gamma \vdash \Delta\} \qquad \{\Gamma \vdash \Phi\}}{\{\Gamma \vdash \Delta \cup \Phi\}}$$

7.6.5 Derived Inference Rules

In our initial implementatation of these languages, we focused on simplicity and decided not to allow derived inference rules. Experience gained since then has showed that these may be very helpful in the proof composition, and reduction in proof size, and may thus be worth the additional complexity in the proof-checker.

7.6.6 Meta Symbols

Another addition that may be worthwhile is the addition of "meta-symbols". While meta-variables allow us to talk about pattern-schema and proof-schema—reresenting sets of concrete patterns and concrete proofs over them—meta-symbols extend this idea to a matching logic theories as a whole—we may talk about meta-theories instead of a single concrete theory. For example, we may state that "for any pattern for which these axioms hold, we may prove these theorems".

The more pressing motivation for this is more mundane—it makes proof decomposition easier by allowing us to "forward declare" notation—state that a notation exists (and enumerate its properties) without defining it in full. This may be particularly helpful to decompose proofs for a long program executions.

CHAPTER 8: PROOF GENERATION

In this section, we describe the how proofs in this format are produced. We will present two evaluations—(1) proof generation for the equivalance of regular expressions and finite automata, via an instrumented python implementation of Brzozowski's method presented in this chapter; and (2) proof generation for concrete execution of programs through the instrumentation of $\mathbb{K}$'s concrete execution backend presented in the next chapter. Note that there is still more work to be done before we can call the second implementation complete—there are several "holes" in the proofs that are filled by assumptions.

In general, we expect that proofs generated through instrumented algorithms will consist of two parts. First, a set of manually proved lemmas. Each of these lemmas corresponds to particular steps or transitions that the algorithm can make. For example, normalizing a regular expression or taking its derivative correspond to a proof of equivalence between a pattern capturing the operation and the result of the said operation. Second, these lemmas are put together with the aid of the instrumentation. The glue that binds these two pieces together is a "logical" representation of the algorithm's state as pattern. The application of each pre-proved lemma takes the us from one logical state to the next. Each of these would be separated into two proof-units, and composed to get a complete proof.

In producing these proofs we have two seemingly conflicting goals: First, for the case of the manually produced proofs, we need a clean and convenient interface, with plenty of debugging information. Without this manually producing the non-trivual amount of high-level lemmas from the matching logics low-level proof-systen will quickly bog us down. Second, we need to produce these proofs efficiently so that we may obtain reasonable performance for long executions. This may involve turning off checking that the proof is correct at generation time—it will be verified later by the proof checker in any case. We also want to produce proofs on the fly, without keeping the entire proof term in memory.

To enable this, we define a domain specific lanuage (DSL) in Python that may be interpreted in different ways—perhaps with fast serialization in mind, perhaps with strict checking enabled, or perhaps with the goal of identifying repeated lemmas and memoizing them. This mechanism gives us a *polymorphism* of proofs—later the same mechanism may give us the ability to separate proofs into different proof-units depending on the level of abstraction we produce the proof at. It also gives us the ability to easily make changes to the underlying proof-format—or perhaps even generate a completely different one, such as Metamath as we did in an experiment.

8.1 MATCHING LOGIC DOMAIN SPECIFIC LANGUAGE

The core of this DSL is the `Interpreter` class. It is an abstract class defines one method for each instruction in the BAML format specified in the previous section.

```
class Interpreter:
    def bot(self): ...
    def evar(self, id): ...
    def implies(left, right): ...
    def mu(self, var, subpattern): ...
    ...

    def prop1(self): ...
    ...
    def modus_ponens(self, major, minor): ...
    def exists_quantifier(var_x, var_y): ...
    ...
    def prefixpoint(self, binder) ...:
    def kt(self, phi, binder, hypo):

    def instantiate(self, proved, theta): ...

    def pop(self, term): ...
    def save(self, id, term): ...
    def load(self, id, term): ...
    def publish_proof(self, term): ...
    def publish_axiom(self, term): ...
    def publish_claim(self, term: Pattern): ...
```

There are little to no constraints on how an instance of this abstract interface is defined. Let us explore some of the implementations we have written.

`VerifyingInterpreter` This interpreter verifies proofs as it interprets them. This may seem redundant since it duplicates the checking done by the BAML checker written in Rust. This, however, gives us the ability to emit developer friendly messages regarding exactly where in the proof the error has occured.

`TAMLSerializer` This interpreter serializes the proof to the textual representation of AML, allowing us define manually inspect the proof and use standard UNIX tool to eyeball changes to large and complex proofs for debugging purposes. We also use this interpreter to generate input to test the $\mathbb{K}$ specification of the TAML proof checker.

`BAMLSerializer` interprets proofs into the binary BAML format, to be checked by our small, trusted rust checker. This is the final, finished product of the proof-generation phase.

MemoizingIntperpreter This interpreter replaces patterns that have occured multiple times and inserts `Save` and `Load` instructions to allow their reuse. This works in conjunction with the `CountingInterpreter` that counts the number of times a pattern occurs. Unfortunately, this tool is a stepping stone for identifying possible lemmas—it breaks the streaming nature of proof generation, requiring two passes over the proof.

Since the interface the `Interpreter` gives us is pretty low-level, we use a class called `ProofExp` to give us a higher-level interface. This include versions of axioms that include instantiations, and convenience functions for constructing patterns. For example, the following method uses its interface to construct a proof for $\varphi \to \varphi$.

```
class Propositional(ProofExp):
  ...
  def imp_refl(self):
    phi_implies_phi = Implies(phi, phi)
    return self.modus_ponens(
              self.modus_ponens(
                  self.prop2_inst(phi, phi_implies_phi, phi),
                  self.prop1_inst(phi, phi_implies_phi)),
              self.prop1_inst(phi, phi))
```

8.2 INSTRUMENTING BRZOZOWSKI'S METHOD

In this section, we will describe how a Python implementation of Brzozowski's method for checking the equivalce of regular expressions may be instrumented to produce such proofs. Below, we see an implementation of Brzozowski's algorithm in Python. The lines highlighted in gray are instrumentation, and allow us to generate proofs.

```
def brz(exp: Regex, prev: list[Regex], instr) -> bool:
    if exp in prev:
        instr.leaf(prev.index(exp))
        return True
    if not has_ewp(exp, instr): return False
    prev.append(exp)
    instr.enter_node()
    left = brz(der(a, exp, instr), instr, prev=prev)
    right = brz(der(b, exp, instr), instr, prev=prev)
    prev.pop()
    instr.exit_node()
    return left and right
```

As described in Chapter 6, the proofs we generate here involve two parts—first for proving that the regular expressions is equivalent to an automata, and second that the this automata is valid. Both these depend on having a pattern $\mathfrak{pat}_{\mathcal{Q}}$ at hand. Since this pattern must also be generated through instrumenting the algorithm, the proof cannot be produced via streaming, and we require two passes of the algorithm.

The class `PatQInstr` below implements the instrumentation "hooks" called in the definition of `brz`. Thus, when passed in as the `instr` argument to the `brz` function, it builds $\mathfrak{pat}_{\mathcal{Q}}$ as the algorithm progresses.

```
class PatQInstr(BrzInstumentation):
    pat_n_stack: list[Pattern] = field(default_factory=lambda: [])
    def pat_n(self):
        return self.pat_n_stack[-1]

    def pat_q(self):
        assert len(self.pat_n_stack) == 1
        return self.pat_n()

    def leaf(self, index):
        self.pat_n_stack.append(SVar(index))

    def exit_node(self):
        right = self.pat_n_stack.pop()
        left = self.pat_n_stack.pop()
        index = len(self.prev)
        self.pat_n_stack.append(accepting_node(index)(left, right))
```

Similarly, the `ProveRegexImpliesPatQ` constructs a proof for $\Gamma_{\text{Word}} \vdash \alpha \to \mathfrak{pat}_{\mathcal{Q}}$. Note that this extends the `ProveNormalization` class which generates the sub-proofs involving derivates, and reasoning modulo associativity and commutativity.

```
class ProveRegexImpliesPatQ(PatQInstr, ProveNormalization):
    def proof(self):
        return self.proof_stack[0]

    def leaf(self, index):
        super().leaf(index)
        self.proof_stack.append(self.words.prop.imp_refl(self.prev[index].to_pattern()
```

```
    def exit_node(self):
        super().exit_node()
        pat_na, pat_nb = accepting_node(len(self.prev)).assert_matches(self.pat_n)
```

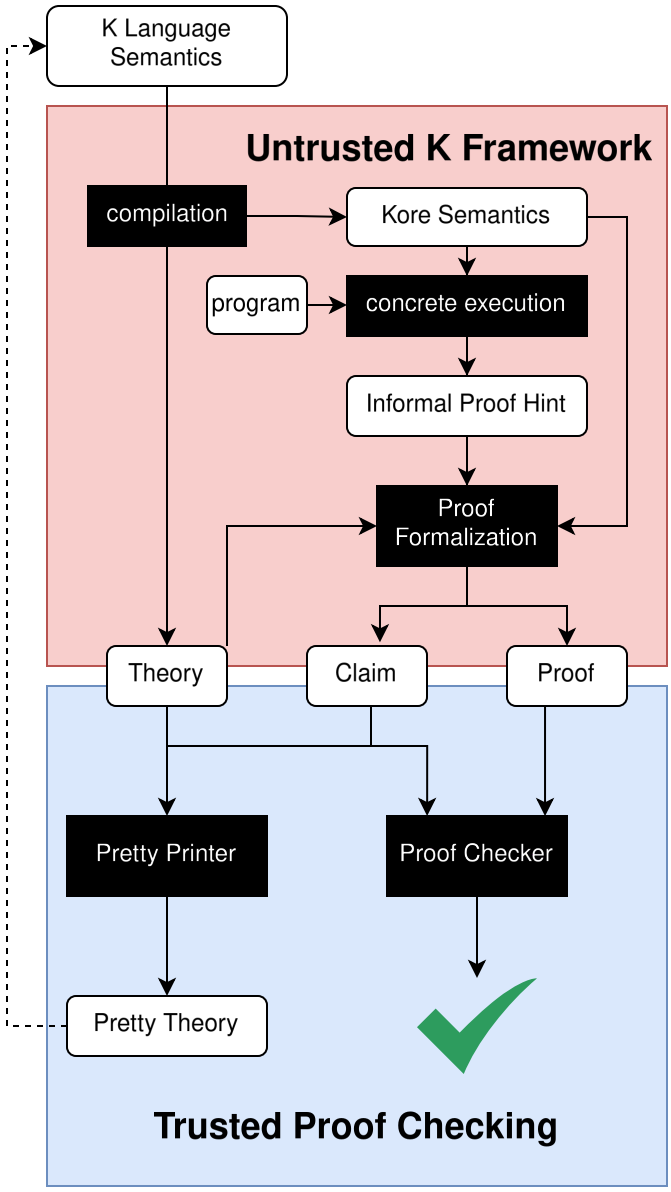
```
...
binder = len(self.prev)
self.proof_stack.append(
    self.words.pat_q_implies_regex_recursive(
        binder, ... alpha_has_ewp,
        alpha_der_a_simpl, pat_na_implies_der_a,
        alpha_der_b_simpl, pat_nb_implies_der_b,
    )
)
```

Here `self.words.prop.imp_refl`, corresponds to the base case of the Lemma 6.1, while `self.words.pat_q_implies_regex_recursive` refers to the recursive case. To stress on something mentioned previously, the simplicity of this implementation is only possible because of the structural similarity between the computational aspect of the algorithm and it's logical representation.

8.3 PROOFS OF $\mathbb{K}$'S CONCRETE EXECUTION

In this section, we describe an on-going joint effort with Pi Squared **?**. We will focus on the technical developments since the work of Chen et al in [5] rather than the theoretical work of defining $\mathbb{K}$ specifications in matching logic, already covered there. There are two goals of our implementation over the previous implementation is first, to use a streaming-based approach and second, to enable use of the more performant proof language and so enable zero-knowledge proofs. This work is in progress, and so we will not go into too much detail.

For the most part, the implementation of proof generation for $\mathbb{K}$ execution is a scaled up and more complex version of the proof generation we saw in the previous section, with a few key differences. First, rather than working over a single concrete theory Γ_{Word} we need to generate a theory from the K lanauge semantics. This is because the theory used is specific to the language. To do this we will use the informal "kore definition" to produce a formal matching logic theory emitted by the $\mathbb{K}$ compilation process. The formal theory will be used as part of the trust-base, while the kore definition is an *untrusted* internal representation of the semantics used internally by $\mathbb{K}$ tooling. Note that this is a temporary stepping stone—eventually we will want the $\mathbb{K}$ compilation process to direclty produce the formal matching logic theory.



Next, $\mathbb{K}$'s `krun` tool reads the kore semantics, along with an input program and executes it. As a side effect, it produces a *proof hint*. This is serialization of the instrumentation—an informal artifact that contains all the information needed to produce a proof. For example, it includes the rules and simplifications applied at each step, as well as the substitutions and contexts in which they were applied.

We decided to put in this layer of indirection in order to keep proof generation code separate from concrete rewriting engine. This allows us to keep a high performance standard for the execution engine (e.g. not breaking tail-call optimization) while trading-off performance for having a working product and keeping in-line with the general principle of avoiding pre-mature optimization. To allow this, the proof-generator does not consume instrumentation directly

but instead produces an artifact called *proof-hints* serializing this instrumentation. Once we have a robust implementation we may think of embedding the proof generation directly into the backend, perhaps via a plugin architecture.

Finally, this proof hint is formalized into a proof using a separate proof-formalization tool. This consumes the proof hint, and produces a proof, similar to the `ProveRegexImpliesPatQ` in the previous section.

With these two components, the remaining gap is the untrusted compilation from the $\mathbb{K}$ language semantics to the matching logic theory. How can ensure that the original semantics and the emitted theory talk about the same language? The *pretty printer* aims to close this gap. Through the use of notation, we want to produce pretty printed theory, with minimal representational distance with the original language semantics. Eventually, we aim to make this distance zero by defining a formal semantics of $\mathbb{K}$ in $\mathbb{K}$, simplifying high-level constructs into lower-level ones, and eventually into matching logic.

8.4 EVALUATION

In this section, we will evaluate the new proof generation architecture from the point of view of both generation and checking, comparing it against the previous Metamath Zero implementation in Chapter 6. In addition, we will compare a few proofs against the Metamath formalization of matching logic developed in [5].

Bibliography

[1] GCC Team, "Standards (using the GNU compiler collection (GCC))."

[2] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of C," in *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation*, 2015, pp. 336–345.

[3] D. Park, A. Stefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation*, 2015, pp. 346–356.

[4] E. Hildenbrandt *et al.*, "KEVM: A complete semantics of the Ethereum virtual machine," in *Proceedings of the 2018 IEEE computer security foundations symposium (CSF'18)*, IEEE, 2018.

[5] X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu, "Towards a trustworthy semantics-based language framework via proof generation," in *Computer aided verification: 33rd international conference, CAV 2021, virtual event, july 20–23, 2021, proceedings, part II 33*, Springer, 2021, pp. 477–499.

[6] X. Chen and G. Roşu, "Matching μ-logic," University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/102281, Jan. 2019.

[7] X. Chen and G. Roşu, "A general approach to define binders using matching logic," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–32, 2020.

[8] A. Tarski *et al.*, "A lattice-theoretical fixpoint theorem and its applications." *Pacific journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[9] D. Park, "Fixpoint induction and proofs of program properties," *Machine intelligence*, vol. 5, 1969.

[10] D. Kozen, "Results on the propositional μ-calculus," *Theoretical computer science*, vol. 27, no. 3, pp. 333–354, 1983.

[11] X. Chen and G. Roşu, "Matching μ-logic," in *2019 34th annual ACM/IEEE symposium on logic in computer science (LICS)*, IEEE, 2019, pp. 1–13.

[12] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th annual IEEE symposium on logic in computer science*, IEEE, 2002, pp. 55–74.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, no. 10, pp. 576–580, Oct. 1969, doi: 10.1145/363235.363259.

[14] P. W. O'Hearn and D. J. Pym, "The logic of bunched implications," *Bulletin of Symbolic Logic*, vol. 5, no. 2, pp. 215–244, 1999.

[15] G. Roşu, "Matching Logic," *Logical Methods in Computer Science*, vol. Volume 13, Issue 4, Dec. 2017, doi: 10.23638/LMCS-13(4:28)2017.

[16] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014.

[17] V. Buterin and Ethereum Foundation, "Ethereum White Paper." Ethereum, 2013.

[18] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International symposium on formal methods for components and objects*, Springer, 2005, pp. 364–387.

[19] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*, Springer, 2010, pp. 348–370.

[20] E. Cohen *et al.*, "VCC: A practical system for verifying concurrent C," in *Proceedings of the 22nd international conference on theorem proving in higher order logics (TPHOLs'09)*, Springer, Aug. 2009, pp. 23–42. doi: 10.1007/978-3-642-03359-9.

[21] KEVM Team, "KEVM: Semantics of EVM in K." https://github.com/kframework/evm-semantics, 2017.

[22] X. Wan and A. Adams, "Just-in-time liquidity on the uniswap protocol," *Available at SSRN 4382303*, 2022.

[23] M. Bellare and G. Neven, "Identity-based multi-signatures from RSA," in *Topics in cryptology – CT-RSA 2007*, M. Abe, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 145–162.

[24] M. Scharnowski, S. Scharnowski, and L. Zimmermann, "Fan tokens: Sports and speculation on the blockchain," *Journal of International Financial Markets, Institutions and Money*, vol. 89, p. 101880, 2023.

[25] S. Lee *et al.*, "Cybercriminal minds: An investigative study of cryptocurrency abuses in the dark web," in *Proceedings 2019 network and distributed system security symposium*, Internet Society, 2019.

[26] M. White, "Web3 is going just great." 2024.

[27] P. Daian, "DAO attack." 2016.

[28] M. del Castillo, "Ethereum executes blockchain hard fork to return DAO funds," 2016.

[29] V. Buterin, "Thinking about smart contract security," 2016.

[30] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts." *IACR Cryptology ePrint Archive*, vol. 2016, p. 1007, 2016.

[31] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, "An in-depth look at the parity multisig bug," 2017.

[32] J. Steiner, "Security is a process: A postmortem on the parity multi-sig library self-destruct." Ethcore, 2017.

[33] The Ethereum Foundation, "ERC20 token standard." https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md, 2017.

[34] J. Solana, "$500K hack challenge backfires on blockchain lottery SmartBillions." CalvinAyre, 2017.

[35] J. Manning, "Ether.camp's HKG token has a bug and needs to be reissued," 2017.

[36] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010, doi: 10.1016/j.jlap.2010.03.012.

[37] Formal Systems Lab, UIUC, "The K framework." 2006.

[38] A. Quenston, "Ethereum's blockchain accidentally splits," 2016.

[39] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers." WSTC17, International Conference on Financial Cryptography and Data Security, 2017.

[40] KEVM and Runtime Verification, "Github: Verified Smart Contracts." https://github.com/runtimeverification/verified-smart-contracts, 2018.

[41] The Ethereum Foundation, "Ethereum tests." https://github.com/ethereum/tests, 2015.

[42] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming languages and systems: 22nd european symposium on programming, ESOP 2013, held as part of the european joint conferences on theory and practice of software, ETAPS 2013, rome, italy, march 16-24, 2013. proceedings*, M. Felleisen and P. Gardner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. doi: 10.1007/978-3-642-37036-6_8.

[43] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," in *ACM SIGPLAN notices*, ACM, 2014, pp. 175–188.

[44] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts. Technical report." 2018.

[45] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter." Cryptology ePrint Archive, Report 2016/633, 2016.

[46] T. of Bits, "Manticore: Symbolic execution for humans." https://github.com/trailofbits/manticore, 2017.

[47] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of security and trust*, L. Bauer and R. Küsters, Eds., Cham: Springer International Publishing, 2018, pp. 243–269.

[48] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[49] E. Cohen *et al.*, "VCC: A practical system for verifying concurrent c," in *International conference on theorem proving in higher order logics*, Springer, 2009, pp. 23–42.

[50] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić, "A reachability predicate for analyzing low-level software," in *International conference on tools and algorithms for the construction and analysis of systems*, Springer, 2007, pp. 19–33.

[51] C. Hawblitzel *et al.*, "IronFleet: Proving practical distributed systems correct," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 1–17.

[52] C. Hawblitzel *et al.*, "Ironclad apps: End-to-end security via automated full-system verification," in *11th {USENIX} symposium on operating systems design and implementation ({OSDI} 14)*, 2014, pp. 165–181.

[53] N. Polikarpova, C. A. Furia, and S. West, "To run what no one has run before: Executing an intermediate verification language," in *International conference on runtime verification*, Springer, 2013, pp. 251–268.

[54] D. Liew, C. Cadar, and A. F. Donaldson, "Symbooglix: A symbolic execution engine for boogie programs," in *2016 IEEE international conference on software testing, verification and validation (ICST)*, IEEE, 2016, pp. 45–56.

[55] A. Lal, S. Qadeer, and S. Lahiri, "Corral: A whole-program analyzer for boogie," in *These informal proceedings contain the papers presented at BOOGIE 2011, the first international workshop on intermediate verification languages held on 1 august 2011 in wroc law. There were 8 submissions. Each submission was reviewed by at least 3 pro-gram committee members. The committee decided to accept 7 papers, repre*, Citeseer, 2011, p. 78.

[56] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers (t)," in *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*, IEEE, 2015, pp. 166–177.

[57] K. R. M. Leino, "This is boogie 2," *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[58] T. K. team, *KLEE: Solver chain*. Available: https://klee.github.io/docs/solver-chain/#caching-solvers

[59] M. Ameri and C. A. Furia, "Why just boogie?" in *International conference on integrated formal methods*, Springer, 2016, pp. 79–95.

[60] X. Chen and G. Roşu, "Matching μ-logic," in *Proceedings of the 34th annual ACM/IEEE symposium on logic in computer science (LICS'19)*, Vancouver, Canada: ACM, 2019, pp. 1–13.

[61] Z. Ésik, "Completeness of Park induction," *Theoretical Computer Science*, no. 1, pp. 217–283, 1997, doi: 10.1016/S0304-3975(96)00240-X.

[62] D.-H. Chu, J. Jaffar, and M.-T. Trinh, "Automatic induction proofs of data-structures in imperative programs," in *Proceedings of the 36th annual ACM SIGPLAN conference on programming language design and implementation (PLDI'15)*, ACM, 2015, pp. 457–466. doi: 10.1145/2737924.2737984.

[63] M. Sighireanu *et al.*, "SL-COMP: Competition of solvers for separation logic," in *Tools and algorithms for the construction and analysis of systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds., Cham: Springer International Publishing, 2019, pp. 116–132.

[64] R. Goldblatt, *Logics of time and computation*, 2. ed. in CSLI lecture notes, no. 7. Stanford, CA: Center for the Study of Language; Information, 1992.

[65] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th annual IEEE symposium on logic in computer science (LICS'02)*, IEEE, Jul. 2002, pp. 55–74. doi: 10.1109/lics.2002.1029817.

[66] A. Pnueli, "The temporal logic of programs," in *Foundations of computer science, 1977., 18th annual symposium on*, IEEE, 1977, pp. 46–57.

[67] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *Proceedings of the 28th symposium on logic in computer science (LICS'13)*, New Orleans, USA: IEEE, Jun. 2013, pp. 358–367. doi: 10.1109/lics.2013.42.

[68] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, no. 5, pp. 279–295, 1997, doi: 10.1109/32.588521.

[69] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications (OOPSLA'16)*, ACM, Nov. 2016, pp. 74–91. doi: 10.1145/2983990.2984027.

[70] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems (TACAS'08)*, Springer, Apr. 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[71] C. Barrett *et al.*, "CVC4," in *Proceedings of the 23rd international conference on computer aided verification (CAV'11)*, Springer, 2011, pp. 171–177.

[72] W. W. Boone, "The word problem," *Proceedings of the National Academy of Sciences*, vol. 44, no. 10, pp. 1061–1065, 1958, doi: 10.1073/pnas.44.10.1061.

[73] C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar, "Compositional entailment checking for a fragment of separation logic," *Formal Methods in System Design*, vol. 51, no. 3, pp. 575–607, Dec. 2017, doi: 10.1007/s10703-017-0289-4.

[74] R. Iosif, A. Rogalewicz, and J. Simacek, "The tree width of separation logic with recursive definitions," in *Proceedings of the 24th international conference on automated deduction (CADE'13)*, Springer, Jun. 2013, pp. 21–38. doi: 10.1007/978-3-642-38574-2_2.

[75] J. Brotherston, N. Gorogiannis, and R. L. Petersen, "A generic cyclic theorem prover," in *Programming languages and systems*, R. Jhala and A. Igarashi, Eds., Springer, 2012, pp. 350–367.

[76] X. Chen, M.-T. Trinh, N. Rodrigues, L. Peña, and G. Roşu, "Towards a unified proof framework for automated fixpoint reasoning using matching logic," University of Illinois at Urbana-Champaign, 2020. Available: http://hdl.handle.net/2142/108369

[77] J. Brotherston and M. Kanovich, "Undecidability of propositional separation logic and its neighbours," *Journal of the ACM*, vol. 61, no. 2, Apr. 2014, doi: 10.1145/2542667.

[78] J. Berdine, C. Calcagno, and P. W. O'Hearn, "A decidable fragment of separation logic," in *Proceedings of the 24th international conference on foundations of software technology and theoretical computer science (FSTTCS'04)*, Springer, 2004, pp. 97–109. doi: 10.1007/978-3-540-30538-5_9.

[79] J. Brotherston, C. Fuhs, J. A. N. Pérez, and N. Gorogiannis, "A decision procedure for satisfiability in separation logic with inductive predicates," in *Proceedings of the joint meeting of the twenty-third EACSL annual conference on computer science logic (CSL) and the twenty-ninth annual ACM/IEEE symposium on logic in computer science (LICS)*, in CSL-LICS '14. New York, NY, USA: ACM, 2014, pp. 25:1–25:10. doi: 10.1145/2603088.2603091.

[80] J. Katelaan, C. Matheja, and F. Zuleger, "Effective entailment checking for separation logic with inductive definitions," in *Tools and algorithms for the construction and analysis of systems*, T. Vojnar and L. Zhang, Eds., Cham: Springer International Publishing, 2019, pp. 319–336.

[81] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *Proceedings of the 3rd asian conference on programming languages and systems (APLAS'05)*, Springer, Nov. 2005, pp. 52–68. doi: 10.1007/11575467_5.

[82] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties via user-defined predicates in separation logic," *Sci. Comput. Program.*, vol. 77, no. 9, pp. 1006–1036, Aug. 2012, doi: 10.1016/j.scico.2010.07.004.

[83] Z. Rakamarić, J. Bingham, and A. J. Hu, "An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures," in *Proceedings of the 8th international conference on verification, model checking, and abstract interpretation (VMCAI'07)*, Springer, Jan. 2007, pp. 106–121. doi: 10.1007/978-3-540-69738-1_8.

[84] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti, "Verifying heap-manipulating programs in an SMT framework," in *Proceedings of the 5th international symposium on automated technology for verification and analysis (ATVA'07)*, Springer, Oct. 2007, pp. 237–252. doi: 10.1007/978-3-540-75596-8_18.

[85] S. Lahiri and S. Qadeer, "Back to the future: Revisiting precise program verification using SMT solvers," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'08)*, ACM, Jan. 2008, pp. 171–182. doi: 10.1145/1328438.1328461.

[86] S. Ranise and C. Zarba, "A theory of singly-linked lists and its extensible decision procedure," in *Proceedings of the 4th IEEE international conference on software engineering and formal methods (SEFM'06)*, IEEE, Oct. 2006, pp. 206–215. doi: 10.1109/sefm.2006.7.

[87] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu, "A logic-based framework for reasoning about composite data structures," in *Proceedings of the 20th international conference on concurrency theory (CONCUR'09)*, Springer, Sep. 2009, pp. 178–195. doi: 10.1007/978-3-642-04081-8_13.

[88] N. Bjørner and J. Hendrix, "Linear functional fixed-points," in *Proceedings of the 21st international conference on computer aided verification (CAV'09)*, Springer, 2009, pp. 124–139. doi: 10.1007/978-3-642-02658-4_13.

[89] J. A. N. Pérez and A. Rybalchenko, "Separation logic + superposition calculus = heap theorem prover," in *Proceedings of the 32nd annual ACM SIGPLAN conference on programming language design and implementation (PLDI'11)*, ACM, Jun. 2011, pp. 556–566. doi: 10.1145/1993498.1993563.

[90] R. Piskac, T. Wies, and D. Zufferey, "Automating separation logic using SMT," in *Proceedings of the 25th international conference on computer aided verification (CAV'13)*, Springer, Jul. 2013, pp. 773–789. doi: 10.1007/978-3-642-39799-8_54.

[91] K. R. M. Leino and M. Moskal, "Co-induction simply," in *Proceedings of the 19th international symposium on formal methods (FM'14)*, Springer, May 2014, pp. 382–398. doi: 10.1007/978-3-319-06410-9.

[92] B. Jacobs, J. Smans, and F. Piessens, "A quick tour of the VeriFast program verifier," in *Proceedings of the 8th asian symposium of programming languages and systems (APLAS'10)*, Springer, Nov. 2010, pp. 304–311. doi: 10.1007/978-3-642-17164-2.

[93] *The Coq proof assistant reference manual*. LogiCal Project, 2004.

[94] The Isabelle development team, "Isabelle." 2018.

[95] H. Unno, S. Torii, and H. Sakamoto, "Automating induction for solving Horn clauses," in *Proceedings of the 29th international conference on computer aided verification (CAV'17)*, Springer, Jul. 2017, pp. 571–591. doi: 10.1007/978-3-319-63390-9_30.

[96] Q.-T. Ta, T. C. Le, S.-C. Khoo, and W.-N. Chin, "Automated mutual induction proof in separation logic," *Formal Aspects of Computing*, no. 2, pp. 207–230, Apr. 2019, doi: 10.1007/s00165-018-0471-5.

[97] C. Löding, M. Parthasarathy, and L. Peña, "Foundations for natural proofs and quantifier instantiation," *Proceedings of the ACM on Programming Languages (POPL'17)*, pp. 1–30, 2017, doi: 10.1145/3158098.

[98] J. Brotherston, D. Distefano, and R. L. Petersen, "Automated cyclic entailment proofs in separation logic," in *Proceedings of the 23rd international conference on automated deduction (CAV'11)*, Springer, 2011, pp. 131–146.

[99] X. Chen, M.-T. Trinh, N. Rodrigues, L. Peña, and G. Roşu, "Towards a unified proof framework for automated fixpoint reasoning using matching logic," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[100] N. Rodrigues, M. Sebe, X. Chen, and G. Roşu, "Technical report: A logical treatment of finite automata," https://hdl.handle.net/2142/121770, 2024.

[101] J. R. Buchi, "Weak second-order arithmetic and finite automata," *Mathematical Logic Quarterly*, vol. 6, no. 1–6, pp. 66–92, 1960, doi: https://doi.org/10.1002/malq.19600060105.

[102] C. C. Elgot, "Decision problems of finite automata design and related arithmetics," *Transactions of the American Mathematical Society*, vol. 98, no. 1, pp. 21–51, 1961.

[103] B. A. Trakhtenbrot, "Finite automata and the logic of one-place predicates," *Sibirskii Matematicheskii Zhurnal*, vol. 3, no. 1, pp. 103–131, 1962.

[104] W. Thomas, "Languages, automata, and logic," in *Handbook of formal languages: Volume 3 beyond words*, Springer, 1997, pp. 389–455.

[105] D. Traytel and T. Nipkow, "Verified decision procedures for MSO on words based on derivatives of regular expressions," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 3–12, 2013.

[106] A. Salomaa, "Two complete axiom systems for the algebra of regular events," *Journal of the ACM (JACM)*, vol. 13, no. 1, pp. 158–169, 1966.

[107] T. Coquand and V. Siles, "A decision procedure for regular expression equivalence in type theory," in *Certified programs and proofs: First international conference, CPP 2011, kenting, taiwan, december 7-9, 2011. Proceedings 1*, Springer, 2011, pp. 119–134.

[108] A. Krauss and T. Nipkow, "Proof pearl: Regular expression equivalence and relation algebra," *Journal of Automated Reasoning*, vol. 49, pp. 95–106, 2012.

[109] N. Rodrigues and M. Sebe, "Matching logic in MM0." Oct. 2023. Accessed: Apr. 14, 2023. [Online]. Available: https://github.com/formal-systems-laboratory/matching-logic-in-mm0

[110] N. Rodrigues and M. Sebe, "A logical treatment of finite automata (artifact)." Zenodo, Dec. 2023. doi: 10.5281/zenodo.10431211.

[111] T. Nipkow and D. Traytel, "Unified decision procedures for regular expression equivalence," in *Interactive theorem proving: 5th international conference, ITP 2014, held as part of the vienna summer of logic, VSL 2014, vienna, austria, july 14-17, 2014. Proceedings 5*, Springer, 2014, pp. 450–466.

[112] S. Fischer, F. Huch, and T. Wilke, "A play on regular expressions: Functional pearl," in *Proceedings of the 15th ACM SIGPLAN international conference on functional programming*, 2010, pp. 357–368.

[113] V. Antimirov, "Partial derivatives of regular expressions and finite automata constructions," in *STACS 95: 12th annual symposium on theoretical aspects of computer science munich, germany, march 2–4, 1995 proceedings*, Springer, 2005, pp. 455–466.

[114] I. Aad, "Zero-knowledge proof," in *Trends in data protection and encryption technologies*, V. Mulder, A. Mermoud, V. Lenders, and B. Tellenbach, Eds., Cham: Springer Nature Switzerland, 2023, pp. 25–30. doi: 10.1007/978-3-031-33386-6_6.

[115] Metamath Team, "Metamath - a computer language for mathematical proofs." https://us.metamath.org/downloads/metamath.pdf, 2019.