Certified Fixpoint Reasoning and Program Execution in Matching Logic

Thesis Proposal

Contents

1 Overview & Motivation

The semantics-first approach to programming language development aims to mitigate two major concerns with the current state of the art approach. First, it aims to ground the language's development in a formal, mathematical semantics, as opposed to a natural language document. Second, it allows the language designers to immerse themselves in the design of the language and leave the generation of the various tools.

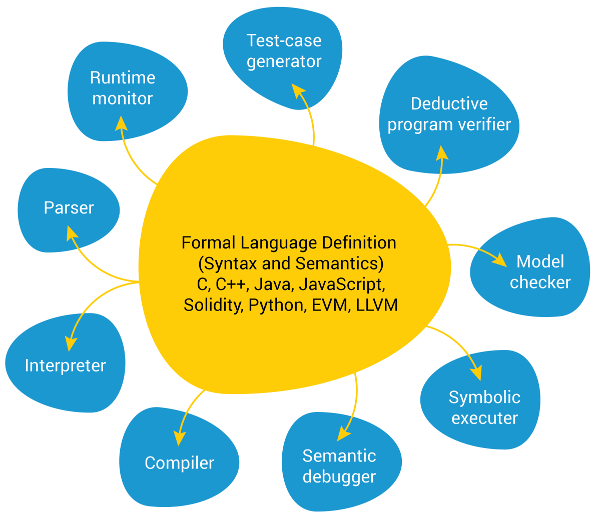

Test-case generator
Runtime monitor
Deductive program verifier
Parser
Formal Language Definition (Syntax and Semantics) C, C++, Java, JavaScript, Solidity, Python, EVM, LLVM
Model checker
Interpreter
Symbolic executer
Compiler
Semantic debugger

Figure 1: In an ideal language development framework, we first formally define the language's semantics. From this, each of the tools needed for the language are automatically derived.

As shown in fig. 1, the first step in this approach is to define a formal mathematical semantics of the language. From this semantics, we may automatically derive various programming language tools such as parsers, interpreters, compilers, and even formal deduction tools.

The 𝕂 Framework takes this approach to programming language development. Many languages, including C (Hathhorn, Ellison, and Roşu 2015), C++, JavaScript (Daejun Park, Stefănescu, and Roşu 2015), and EVM (Hildenbrandt et al. 2018) have their semantics defined for 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.

𝕂's success means that it has grown as a project, and so it has become 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. Much of the LLVM code is dynamically generated. This complexity forces us to seriously reckon with the correctness of $\mathbb{K}$. Full formal verification is tricky because the because of the size and complexity of the code base, and the fast pace of development—tens of commits a day.

Verified computing offers a solution to this. In this approach, the program returns the result along with a proof that the computation was carried out correctly. As $\mathbb{K}$'s mathematical foundation, matching logic is a natural choice to represent these proofs.

$\mathbb{K}$'s semantics are justified in terms of a translation to a formal matching logic theory. This foundation makes it possible to represent the formal semantics of a language as a logical theory. Programs and program state become terms in that logic, and tasks, such as execution, become formulae. 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 proven using matching logic's proof system.

Our long term goal is to instrument each of 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.

The immediate goal of this thesis is to produce such certificates for concrete program execution, and fixpoint reasoning.

2 Preliminaries

2.1 An Overview of Matching Logic

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

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

Definition 2.1. 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, (Chen and Roşu 2019) shows us that we can easily build up to more convenient abstractions. For the rest of this section 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 $\nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$.

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 $\mathrm{succ}(x)$ 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 (Chen and Roşu 2019, 2020). Formulae are embedded as patterns with little to no *representational distance*, quite often verbatim.

Patterns aren't 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.

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

Proof System The third component to matching logic is its proof system, shown in fig. 2. It defines the provability relation, written $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system using the theory Γ as additional axioms. These proof rules fall into four categories. First, the FOL rules provide complete FOL and propositional reasoning. The (PROPAGATION) rules allow applications to commute through constructs with a "union" semantics, such as disjunction and existentials. The proof rule (KNASTER-TARSKI) is an embodiement of the Knaster-Tarski fixpoint theorem (Tarski et al. 1955), and together with (PREFIXEDPOINT) correspond to the Park induction rules of modal logic (David Park 1969; Kozen 1983). Finally, (EXISTENCE), (SINGLETON), and (SUBST) are technical rules, needed to work with variables.

2.2 The $\mathbb{K}$ Framework

The $\mathbb{K}$ Framework takes the semantics-first approach to programming language design, and attempts to implement the "ideal language framework" as shown in fig. 1. The small-step semantics of a language is defined formally in $\mathbb{K}$ as a set or rewriting rules. From this we may derive various language tools such an as parsers, interpreters, deductive verifiers, and even a compiler.

$\mathbb{K}$'s compiler is called the KSummarizer, it works by executing a program with symbolic inputs and reasoning in reachability logic, to produce a new $\mathbb{K}$ semantics, specific to that program. Producing proofs for this task would involve both fixpoint reasoning and formalizing symbolic execution.

2.3 Metamath

Metamath is a simple and flexible computer-processable language that supports rigorously verifying, archiving, and presenting mathematical proofs. Metamath is human-readable, and logic-agnostic—it allows defining the schemas and inference rules of the logic.

2.4 Metamath Zero

Metamath Zero is a language inspired by Metamath and Lean, designed to have the simplicity of Metamath, while avoiding some of its issues. This includes soundness problems with inherint in Metamath's design. For example, formulae in Metamath are represented as strings of symbols, rather than syntax trees. This means that it is easy for a user to accidentally define an ambiguous grammar, and allow proving inconsistencies. A second issue is that notations are defined as axioms in Metamath, again making it easy for innocent seeming notational defintions to cause unsoundness. While there are tools in the Metamath ecosystem to check that definitions are conservative, these must be run separately from the main verifier. At the same time, Metamath Zero offers a much lighter axiomatic framework than Lean, allowing us to avoid its complexity, and monolithic nature.

3 My Contribution

The work in my thesis will focus on formal program reasoning and proof generation within matching logic's Hilbert-style proof system. In particular we are interested in producing formal proofs of correctness for operations performed by the $\mathbb{K}$ framework, such as execution of programs, and deductive reasoning.

Our long term goal is to bring verified computing to the $\mathbb{K}$ ecosystem. To break this up into smaller tasks, my work will focus on three stepping stones:

1. **Practicality of the semantics-first approach.** The semantics-first approach adds an extra layer of indirection in implementing programming languages. This leads to questions regarding the scalability and performance of this approach. In joint work with Everett Hildenbrandt, we defined the the Ethereum Virtual Machine as a $\mathbb{K}$ semantics, a real-world programming language. The interpreter generated by $\mathbb{K}$ was faster than a hand-written Python interpreter and within an order of magnitude of one handwritten in C++.

   I also defined a formal semantics for the Boogie IVR in $\mathbb{K}$ to demonstrate the feasability of the semantics-first approach for modular verification of invariants.

2. **Detailed fixpoint reasoning in matching logic's proof system.** Various tools used by $\mathbb{K}$ depend on complex fixpoint reasoning. $\mathbb{K}$'s implementation glosses over them. We aim formalize this reasoning in matching logic. Matching logic's proof system gives us the basic tools we need for fixpoint reasoning via the (Knaster-Tarski) and (Pre-fixpoint) proof rules. These are akin to Park's induction rules for Modal Mu calculus. Again, we are hit by matching logics low-level nature—these rules have very limited application on their own. They are not applicable in many contexts. I begin to address these issues in this thesis.

3. **Producing concise yet detailed proofs of execution for real-world programming languages in matching logic.** Program execution is a complex task that involves many detailed steps when broken down to the level of matching logic. It is thus vital that we have a formal, yet concise and efficiently checkable format for representing these proofs.

Publications and Timeline

- Current, published results:
  1. The following results show the practicality of the K framework in capturing complex, real-world languages.
     a. In (Hildenbrandt et al. 2018), we demonstrate that K and the semantics-first paradigm are competitive with traditional methods of language development. (Joint work with Everett Hildenbrandt).
     b. In the technical report (Rodrigues, Poulsen, and Roşu 2021), we show that K is as capable as other techniques used for program verification. In particular, we capture the core features of the Boogie IVR, including quantified invariants and reachability analysis, as a K semantics.
  2. The next results demonstrate the capabilities of matching logic with respect to fixpoint reasoning.
     a. (Chen et al. 2020) establishes that matching logic may be used as a unified framework for fixpoint reasoning. We show that fixpoint reasoning in separation logic, LTL, and Reachability may be reduced to matching logic.
     b. (Rodrigues et al. 2024) builds on the previous results to show that even more complex fixpoint reasoning may be captured. In particular, we capture regular expressions and finite automata, and reason about their equivalence.
- Planned work

1. **A concise proof-language for matching logic proofs:** We develop a concise yet formal language for representing matching logic proofs. It aims to be efficiently machine-checkable, but concise and formal enough to allow proving correctness.

   In addition, we will develop a proof-checker for this language. and port existing proof-generation projects, such as those in (Rodrigues et al. 2024) and (Chen et al. 2020), to target this language as a test of its applicability.

   Target completion: June 2024

2. Employ the above proof language to capture proofs of execution of real-world languages, defined in K. In particular, we aim for EVM formalized in K.

   Target completion: October 2024

4 Completed Work

4.1 Practicality of the semantics first approach.

In these two works we establish the practicality of the semantics-first approach.

KEVM: A complete semantics of the Ethereum virtual machine (Hildenbrandt et al. 2018)

Published: CSF 2018; Joint work with Everett Hildenbrandt

Here we define the semantics of the Ethereum Virtual Machine using the K Framework. The performance of the generated interpreter beats that of hand-written interpreters in interpreted languages (e.g. Python), and is competitive with one written in C++, a compiled language. This work is in commercial use at Runtime Verification, Inc.

KBoogie: A semantics of the Boogie IVR (Rodrigues, Poulsen, and Roşu 2021)

Technical Report

We define a semantics for the Boogie Intermediate Verification Language. This semantics includes the features of Boogie that are key to its application in programming verification. This includes reachability and invariant verification, where invariants may include quantifiers. Some features such as invariant inference are omitted as, architecturally, they are better suited to being implemented as part of K rather than as a feature of a semantics.

4.2 A Unified Framework for Automated Fixpoint Reasoning

Published: OOPSLA 2020 (Chen et al. 2020)

Joint work with Xiaohong Chen

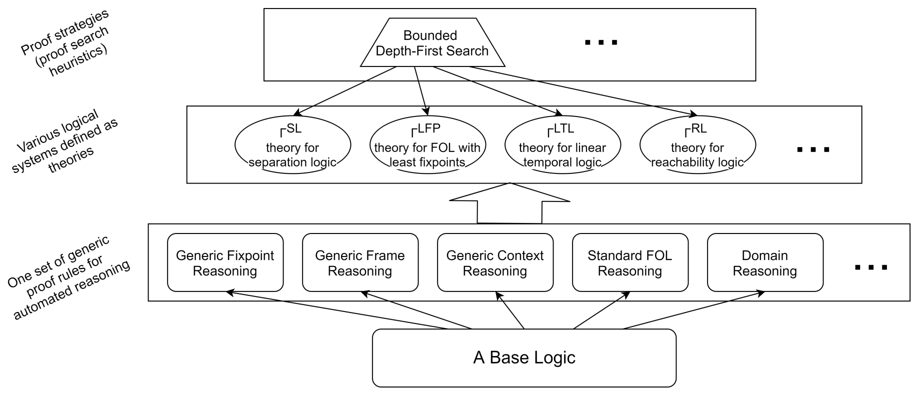

Proof strategies (proof search heuristics)
Bounded Depth-First Search
...
Various logical systems defined as theories
Γ^SL theory for separation logic
Γ^LFP theory for FOL with least fixpoints
Γ^LTL theory for linear temporal logic
Γ^RL theory for reachability logic
...
One set of generic proof rules for automated reasoning
Generic Fixpoint Reasoning
Generic Frame Reasoning
Generic Context Reasoning
Standard FOL Reasoning
Domain Reasoning
...
A Base Logic

Figure 3: An ideal logic for program reasoning.

Since $\mathbb{K}$ performs reasoning in using a variety of techniques established by in foundational work in the field, it is important that we can capture these forms of reasoning in matching logic, rather than re-inventing the wheel.

In this work, we establish that matching logic provides a uniform basis for fixpoint reasoning. We use it as the basis of an automated prover for separation logic with inductive definitions (targeting the SL-COMP benchmark), reachability logic, and use it to prove a complete axiomatization of LTL. The automated prover uses a small number of manually proven theorems that are then used across these domains.

$$(\textsc{Frame})\frac{\vdash \varphi \to \psi}{\vdash C[\varphi] \to C[\psi]} \qquad\qquad \vdash C[\varphi] \to \psi \quad \underset{(\textsc{wrap})}{\overset{(\textsc{unwrap})}{\rightleftharpoons}} \quad \vdash \varphi \to (C \multimap \psi)$$

Figure 4: Matching logic rules for frame reasoning, and "wrapping" and "unwrapping" fixpoints. These are key to fixpoint reasoning in contexts.

4.3 Reasoning about Finite Automata in Matching Logic

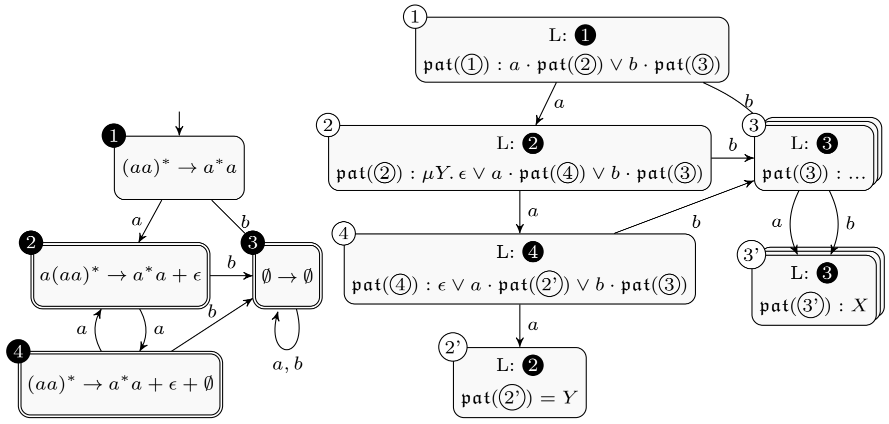
Published: TACAS 2024 (Rodrigues et al. 2024)

This paper demonstrates more complex fixpoint reasoning in matching logic, by demonstrating that we can prove equivalence between arbitrary regular expressions and finite automata.

We present a sound and complete axiomatization of finite words using matching logic. A unique feature of our axiomatization is that it gives a *shallow embedding* of regular expressions into matching logic, and a *logical* representation of finite automata. The semantics of both expressions and automata are precisely captured as matching logic formulae that evaluate to the corresponding language. A regular expressions is matching logic formula *as is*, while the embedding of an automaton $\mathcal{Q}$, is captured by a pattern $\mathfrak{pat}_{\mathcal{Q}}$ that is a *structural analog* of the automata. That is, computational aspects of the automaton is captured as syntactic features of $\mathfrak{pat}_{\mathcal{Q}}$.

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

This property of $\mathfrak{pat}_{\mathcal{Q}}$, allows us to capture computational manipulations of automata as logical manipulations of the pattern $\mathfrak{pat}_{\mathcal{Q}}$. This in turn allows us to capture algorithms that involve these manipulations as logical proofs in matching logic. In particular, this paper demonstrates this with Brzozowski's method for checking equivalence of regular expressions. The proofs produced can be efficiently checked by the Metamath Zero verifier.



$(aa)^* \to a^*a$
$a(aa)^* \to a^*a + \epsilon$
$\emptyset \to \emptyset$
$(aa)^* \to a^*a + \epsilon + \emptyset$
a
b
a, b
L: ❶
$\mathfrak{pat}(①) : a \cdot \mathfrak{pat}(②) \vee b \cdot \mathfrak{pat}(③)$
L: ❷
$\mathfrak{pat}(②) : \mu Y.\, \epsilon \vee a \cdot \mathfrak{pat}(④) \vee b \cdot \mathfrak{pat}(③)$
L: ❸
$\mathfrak{pat}(③) : \ldots$
L: ❹
$\mathfrak{pat}(④) : \epsilon \vee a \cdot \mathfrak{pat}(2') \vee b \cdot \mathfrak{pat}(③)$
L: ❷
$\mathfrak{pat}(2') = Y$
L: ❸
$\mathfrak{pat}(3') : X$

Figure 5: The pattern, $\mathfrak{pat}_{\mathcal{Q}}$, for a $\mathcal{Q}$ is defined via it's unfolding tree. Here, a DFA for the language of $(aa)^* \to a^*a$, is shown on the left. It's unfolding tree is shown on the right. 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 $\mathfrak{pat}(①)$.

We propose this as a general methodology for producing machine-checkable formal proofs, enabled by capturing structural analogs of computational artifacts in logic.

5 Proposed Work

5.1 Proposed architecture for Proof Checking

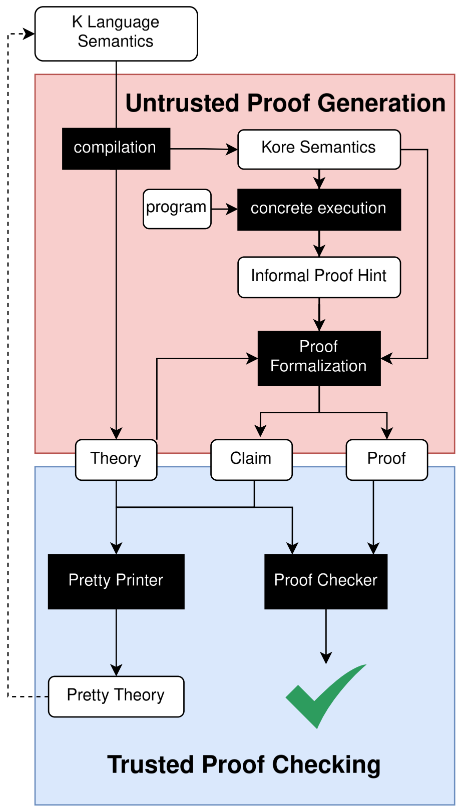


Figure 6: Proposed Architecture for certification of Program Execution. There are two major components—untrusted proof generation, and trusted proof checking and pretty printing.

To gain assuredness of the correctness of the 𝕂 framework, we propose an architecture with three important components—an untrusted proof generation, a trusted proof-checker, and a trusted pretty-printer. This is shown in fig. 6. Let us talk about each component in detail.

Let us start with the **proof checker**. This trusted component takes as input 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. Roughly, it may be thought of as a function:

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

The next component is the **proof generation framework**. This concerns itself with generating the inputs for the proof checker. In the case of $\mathbb{K}$ tools, this involves a number of steps. Let us walk through each of these for concrete execution.

First, as part of an initial compilation phase[1], the language semantics is transformed into a formal matching logic theory, as well as an informal "kore semantics". The formal theory will be used as part of the trust-base, while the kore semantics is an *untrusted* internal representation of the semantics used internally by $\mathbb{K}$ tooling.

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 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.

Finally, this proof hint is formalized into a proof using a separate proof-formalization tool. We do not directly emit fully formalized tools from `krun` to keep a separation of concerns.

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.

5.2 An Efficient Matching Logic Proof Checker

In progress: target completion June 2024

Since formal proofs of program execution need to have detailed reasoning about every aspect of the programming language, it is vital that there proofs have a concise representation that is efficiently machine checkable. At the same time, this representation needs to be close enough to the mathematical abstraction of matching logic's Hilbert-style proof system.

In (Rodrigues et al. 2024), I used the Metamath *Zero* to formalize matching logic's proof system, whereas (Chen et al. 2021) used Metamath. Both these

[1]note that compiles the K semantics, and not the input program

proof languages are intended to be used to represent hand-written and human-readable mathematical proofs. This makes efficiently generating large proofs of exection difficult. While Metamath Zero does include a binary format, it is complex, reads more as a embedded systems language than a formal one.

Further, they are logic-agnostic, and not specific to matching logic. This makes many seemingly mechanical proof steps (such as constructing well-formed patterns, working with substitutions, and instantiations) explicit and repetitive and causes a huge blowup in proof size even for moderately sized program executions—several megabytes and even gigabytes for larger proofs. This language-agnostic nature also makes it quite tricky to formally verify, since besides the implementation, the formalization of matching logic in the base logic must also be reasoned about.

In the case of Metamath, an even more worrying observation is that it is easy to accidentally introduce unsoundness through the definition of notation. To prevent this, we would need additional well-formedness checks in addition to Metamath's default verification process. At this point, we lose much of the advantages of Metamath as an out-of-the-box solution.

Another problem with using these logic-agnostic frameworks is that we must formalize matching logic within them, adding another layer of complexity to formal verification of the checker as a whole.

To remedy this, we plan to develop a proof-language that is:

- *Efficiently machine-checkable:* It is a well structured binary format, that does not need to be parsed ahead-of-time before checking the proof.
- *Formal:* This language may be viewed as a reverse-polish-notation for constructing expressions corresponding to matching logic patterns and proofs, with the addition of features for sharing sub-patterns and sub-proofs. This will make is easy to verify.
- *Economic in Complexity:* We keep the features of the proof-language as minimal as possible, while still remaining practical. The same sharing mechanism mentioned above is used in techniques for representing lemmas and notation. It may also be used to implement let-bindings.
- *Pretty-printable:* In order to trust a proof, it is vital that we may view the assumptions and claims in an unambiguous format. At the same time, this format must be at a high level of abstraction—it is not easy to read a complex program configuration when given directly in terms of matching logics primitives. As a trade-off for efficient checkablity we chose a binary proof format. At the same time, the format is easily printable into an unambigous S-expression based format, or a (possibly ambiguous) mixfix format for human consumption.

Research Goals

- Define a formal concise proof language for representing matching logic proofs. (Complete)

5.3 Efficient Proof Generation

Once we have this concise proof language our goal becomes to efficiently produce proofs in it. As mentioned previously our ultimate goal is to be able to produce proofs for the various $\mathbb{K}$ tasks such as symbolic execution and reachability analysis. Towards that, our medium-term goal is to *efficiently* produce proofs for concrete execution and fixpoint reasoning. These are the stepping-stones we plan on taking towards that:

1. Port proofs generated in (Rodrigues et al. 2024).
   (In Progress; Projected completion April 2024).
2. Port proofs generated in (Chen et al. 2021).
   (In progress; Joint work with Runtime Verification).
3. Generate proofs for a more complex, production programming language such as EVM.
   (Joint work with Runtime Verification).

Port proofs generated in (Rodrigues et al. 2024) and improve efficiency

In (Rodrigues et al. 2024), the focus of the paper was theory, rather than efficient proof generation. We produced a proof of concept implementing Brzozowski's method, instrumented to generate a proof. Now, we may focus on efficiency. I believe that it is possible to bring the effiency of proof generation to within 2-3x of uninstrumented proof generation, not including the cost of IO. This will likely involve being able to *blindly emit the proof*—that is, emit the binary representation of the proof as a raw stream of bytes during execution without any context about where in the proof we are (besides what the algorithm's implementation provides).

Port proofs generated in (Chen et al. 2021) and improve efficiency

We must similarly improve the effiency of the results here. Here, because of the complexity of the task involved I am not confident about being able to blindly emit the proof. Instead, our goal would be to generate the proof in a effient streaming fashion—we should be able to emit the proof as execution is performed, with a reasonable runtime overhead.

Proof Generation for larger and more complex languages.

Not started; Target Completion October 2024

Extend proof generation to more complex programming languages such as EVM via the KFramework and KEVM semantics. This will involve dealing with built-ins, maps, and other complex constructs.

Research problems:

- Optimize checker, and especially ZK components to handle larger proofs.
- Handle various constructs such as:
  - builtin reasoning for integers, etc.
  - assoc/comm maps, sets

6 Future Work

6.1 Zero-Knowledge implementation of proof checker

Even with the savings proofs for program execution may be quite large. Besides, many of the results we need to prove for the same language, but for different programs, or even different runs of the same program may be shared to further reduce proof generation and checking overhead. To do this, we must split up the proof into multiple units—language-agnostic lemmas, language-specific lemmas, and program-specific lemmas.

We plan experiments with *Zero Knowledge* certificates to allow modular, yet secure proof checking. The Zero Knowledge certificate methodology give us a mechanism for doing this, without any third-party trust. It produces a cryptographic certificate proving that the checker successfully ran over the proof. giving a probabilistic guarantee of the proofs correctness. This certificate can be checked much faster than re-checking the entire proof, allowing us to check dependencies between proof modules without re-checking generic lemmas again and again.

6.2 Proof generation for more $\mathbb{K}$ tools

Our long term goal is proof generation for all of $\mathbb{K}$'s tools. Once the work concrete execution, and fixpoint reasoning are complete, we may move on to more complex tools. The most advanced tools, including the deductive verifier and the symbolic execution engine, require the combination of symbolic execution and fixpoint reasoning. Those will therefore be then next target.

6.3 A $\mathbb{K}$ semantics for $\mathbb{K}$

Currently, a large gap in our trustbase is the compilation of a $\mathbb{K}$ language semantics into a matching logic theory. While the use of notations narrows this gap somewhat, it is important to have this translation formally defined. We plan to define this translation as a series of smaller steps simpliying the higher level language into matching logic.

Bibliography

Chen, Xiaohong, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. 2021. "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*, 477–99. Springer.

Chen, Xiaohong, and Grigore Roşu. 2019. "Matching μ-Logic." http://hdl.handle.net/2142/102281. University of Illinois at Urbana-Champaign.

———. 2020. "A General Approach to Define Binders Using Matching Logic." *Proceedings of the ACM on Programming Languages* 4 (ICFP): 1–32.

Chen, Xiaohong, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. 2020. "Towards a Unified Proof Framework for Automated Fixpoint Reasoning Using Matching Logic." *Proceedings of the ACM on Programming Languages* 4 (OOPSLA): 1–29.

Hathhorn, Chris, Chucky Ellison, and Grigore Roşu. 2015. "Defining the Undefinedness of C." In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 336–45.

Hildenbrandt, Everett, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, et al. 2018. "KEVM: A Complete Semantics of the Ethereum Virtual Machine." In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE.

Kozen, Dexter. 1983. "Results on the Propositional μ-Calculus." *Theoretical Computer Science* 27 (3): 333–54.

Park, Daejun, Andrei Stefănescu, and Grigore Roşu. 2015. "KJS: A Complete Formal Semantics of JavaScript." In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 346–56.

Park, David. 1969. "Fixpoint Induction and Proofs of Program Properties." *Machine Intelligence* 5.

Rodrigues, Nishant, Seth Poulsen, and Grigore Roşu. 2021. "KBoogie: Semantics of Boogie." https://hdl.handle.net/2142/121901. University of Illinois at Urbana-Champaign.

Rodrigues, Nishant, Mircea Sebe, Xiaohong Chen, and Grigore Roşu. 2024. "A Logical Treatment of Finite Automata." In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Proceedings, Part I.*

Tarski, Alfred et al. 1955. "A Lattice-Theoretical Fixpoint Theorem and Its Applications." *Pacific Journal of Mathematics* 5 (2): 285–309.