

Research Proposal: Blockchain of Truth (BoT)

1 Proposal Summary

Overview

This project proposes the *blockchain of truth* (BoT) as a means to make, store, certify, and cite *correctness claims*, in a decentralized, collaborative, and public manner. Correctness claims can be proved using rigorous, formal, and machine-checkable mathematical proofs. A fast and trustworthy *proof checker*, which has only 200 lines of code, will underlie BoT’s validators, guaranteeing that all BoT correctness claims are mathematically proved facts and thus are decentralized truths.

BoT is a major step towards the establishment of absolute correctness in programming language design, implementation, deployment, and analysis. BoT builds upon a solid mathematical foundation, matching logic, and a formal language framework, \mathbb{K} . Programming language designers only need to define the formal semantics of their languages. All language tools are autogenerated, correct-by-construction. Language definitions become matching logic theories. The correctness of the autogenerated language tools is established using matching logic proofs, which are encoded as formal proof objects that can be proof-checked by a fast and trustworthy checker.

BoT stores language definitions as logical theories, correctness claims about languages or programs as logical formulas, and matching logic proof objects as hashes. BoT has various stakeholders. Programming language designers or language standard committees can post the formal definitions of their languages to BoT. Program developers and customers can post formal properties about their programs to BoT as correctness claims. Formal reasoning service providers can work out the detailed mathematical proofs of the on-chain correctness claims. Mathematical proofs are automatically proof-checked by the matching logic proof checker, which is the only program running on BoT. If the check passes, the hash of the mathematical proof object is published to BoT.

Intellectual Merit

BoT aims at *absolute correctness* while staying *practically feasible*. To achieve these objectives, it builds upon a small and trusted logical foundation for all programming languages and all formal analyses. BoT uses \mathbb{K} , a language framework that has achieved much success in defining the formal semantics of real-world languages such as C, Java, JavaScript, Python, Ethereum virtual machine bytecode (EVM), and x86-64. \mathbb{K} is likely the most advanced language framework to date, its implementation consisting of more than 500,000 lines of code written in 4 different languages, but it has a minimal logical foundation, called matching logic. Using matching logic, the risk that the complexity of \mathbb{K} can result in errors is eliminated by associating logical theories to programming languages and logical formulas to language tasks (program execution, deductive verification, model checking, etc.). The correctness of each language task is formulated as a proof goal, and each proof goal is validated by an efficiently checkable proof object. The major challenge is to automatically generate proof objects for proof goals.

2 Motivation for the Blockchain of Truth (BoT)

An eternal topic in computer science is the *correctness of computing systems*. People make—sometimes “prove”—correctness claims about computer programs, software systems, and programming languages, using

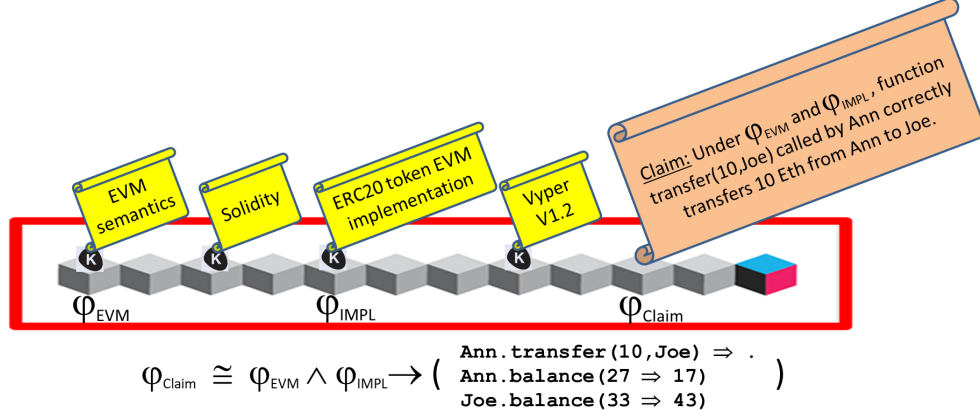


Figure 1: Blockchain of truth (BoT) allows users to upload formal language semantics and correctness claims as logical formulas, whose correctness is established by rigorous mathematical proofs.

one or more of the following formal and/or informal methods:

- formal verification “proofs”;
- state space exploration techniques such as model checking;
- static analysis tool reports;
- tests with “good coverage”;
- human audits (of correctness, security, safety, etc.).

However, the correctness claims that are made using the above methods cannot and should not be blindly trusted. There are many places where correctness is compromised. For example, formal verifiers and model checkers are buggy, so their proofs may be wrong. Static analysis tools miss errors and can report bogus ones. Tests only show bugs, not do not serve as proof of correctness. Human auditors are superficial, informal and biased. Therefore, whenever a correctness claim is made and/or proved, we must be aware of its underlying assumptions, or its *trust base*.

We believe that we need a trustworthy, systematic, and efficient infrastructure to make, store, certify, and cite *correctness claims* and/or their *correctness proofs*. Our vision is shown in Figure 1. Correctness claims are made by various stakeholders, including programming language designers (or language standards committees), program developers, customers, and investors. Hence the correctness claims are highly valuable and must be trustworthy. There is only one known way that can be used to assert—with *absolute certainty*—correctness, and that is the *mathematical proof*.

Blockchain of truth, abbreviated BoT, is our proposal to implement the vision in Figure 1. Its goal is to make, store, certify, and cite mathematical facts, in a decentralized, collaborative, and public manner. The various correctness claims become are mathematical facts, which can be established by rigorous, formal, machine-checkable mathematical proofs. The entire trust base of BoT is a simple 250-line program known as the *proof checker*. The main BoT components are:

1. an *ideal language framework* where programming language designers only need to define the formal syntax and semantics of their languages, and all language tools such as parsers, interpreters, debuggers, deductive verifiers, model checkers, etc., are autogenerated.
2. a uniform *logical foundation* that has an expressive formal language to specify the correctness claims of programming languages, programs, and language tools as logical formulas, and has an efficient proof system to reason about and prove the correctness claims;

3. a trustworthy, simple, and fast *proof checker* to check the above-mentioned formal proofs;
4. a *business model* that incentivizes users to post correctness claims and proofs on BoT.

In other words, BoT can be regarded as a proof-carrying blockchain. The *main advantages* of BoT can be summarized as follows:

1. BoT is completely public and decentralized;
2. BoT has a clear and minimal trust base that is the proof checker;
3. Correctness claims posted on BoT can be trusted.

3 Overview of the BoT Approach

Our BoT approach is based on the following components. Firstly, it builds upon the vision of an ideal language framework, as shown in Figure 2, where programming language designers only need to define the formal syntax and semantics of their languages, and all language tools are autogenerated by the framework. Secondly, it requires a solid logical foundation for the language framework, where language definitions become logical theories, and program properties become logical formulas, which are provable within the theories. Here, we use the term “program properties” in a broader sense, including not only formal verification properties, but also (concrete and symbolic) program executions, model checking properties, safety and liveness properties, etc.

Thirdly, our BoT approach requires the generation of proof objects and a trustworthy proof checker. Thanks to the logical foundation, any program property can be expressed as a logical formula φ under the language definition (i.e., logical theory) Γ^L for a programming language L . The correctness of φ is then witnessed by a formal proof $\Gamma^L \vdash \varphi$. A proof object is a piece of data that encodes the detailed proof steps of $\Gamma^L \vdash \varphi$, which can be mechanically checked by a fast and trustworthy proof checker. Finally, BoT is a blockchain of language formal definitions Γ^L , program properties φ , and their proof objects $\Gamma^L \vdash \varphi$, where the validators are the proof checkers

We explain these components in detail in the following.

An Ideal Language Framework Vision and the \mathbb{K} Framework. As illustrated in Figure 2, an ideal language framework is based on the formal syntax and semantics definitions of programming languages. Language designers only need to define the formal language definitions, from which all language tools are autogenerated by the framework in a correct-by-construction manner. The *correctness* of these language tools is established by generating complete mathematical proofs as certificates that can be automatically machine-checked by a trustworthy proof checker.

The \mathbb{K} language framework (<http://kframework.org>) [29] is in pursuit of the above ideal vision. It provides a simple and intuitive front end language (i.e., a meta-language) for language designers to define the formal syntax and semantics of other programming languages. From such a formal language definition, the framework automatically generates a set of language tools, including a parser, an interpreter, a deductive verifier, a program equivalence checker, among many others [10, 28]. \mathbb{K} has obtained much success in practice, and has been used to define the complete executable formal semantics of many real-world languages, such as C [14], Java [2], JavaScript [25], Python [15], Ethereum virtual machines byte code [18], and x86-64 [12], from which their implementations and formal analysis tools are automatically generated. Some commercial products [16, 21] are powered by these autogenerated implementations and/or tools.

Matching Logic. A uniform logical foundation gives us a formal language to specify the formal semantics of any programming languages as logical axioms and/or theories, and to express program properties as logical formulas. For \mathbb{K} , the logical foundation is *matching logic* [26, 7, 8], which is proposed as a uniform logic designed to specify and reason about programming languages and program properties in a modular and compact way.

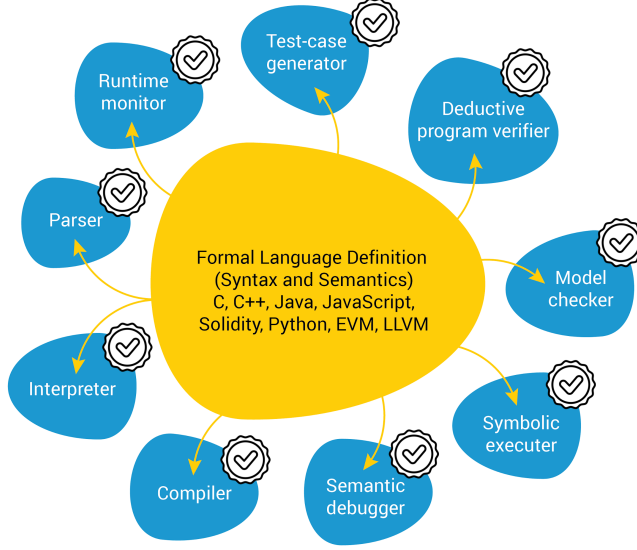



Figure 2: An ideal language framework vision; language tools are autogenerated, with machine-checkable mathematical proofs  as correctness certificates.

Matching logic is the logical foundation of \mathbb{K} in the following sense:

1. The language definition (in Figure 2) of a programming language L corresponds to a *matching logic theory* Γ^L , which, roughly speaking, consists of a set of logical symbols that represents the formal syntax of L , and a set of logical axioms that specify the formal semantics.
2. All language tools in Figure 2 and all language tasks that the framework conducts are formally specified by matching logic formulas. For example, *program execution* is specified (in our approach) by the following matching logic formula:

$$\varphi_{\text{init}} \Rightarrow \varphi_{\text{final}} \quad (1)$$

where φ_{init} is the formula that specifies the initial state of the execution, φ_{final} specifies the final state, and “ \Rightarrow ” states the rewriting/reachability relation between states.

3. There exists a matching logic *proof system* that defines the provability relation \vdash between theories and formulas. For example, the correctness of the above execution from φ_{init} to φ_{final} is witnessed by the formal proof:

$$\Gamma^L \vdash \varphi_{\text{init}} \Rightarrow \varphi_{\text{final}} \quad (2)$$

Therefore, matching logic is the logical foundation of our chosen language framework, whose *correctness* on conducting a language task is reduced to the *existence of a formal proof* in matching logic. Such formal proofs are encoded as proof objects, discussed below.

Proof Objects. In our approach, a proof object is an encoding of a matching logic formal proof, such as Equation (2). Proof objects are automatically generated from the proof parameters provided by \mathbb{K} . At a high level, a proof object for program execution consists of:

1. the formalization of matching logic and its provability relation \vdash ;
2. the formalization of the formal semantics Γ^L as a logical theory, which includes axioms that specify the rewrite/semantic rules $\varphi_{\text{lhs}} \Rightarrow \varphi_{\text{rhs}}$;

3. the formal proofs of all one-step executions, i.e., $\Gamma^L \vdash \varphi_i \Rightarrow \varphi_{i+1}$ for all i ;
4. the formal proof of the final proof goal $\Gamma^L \vdash \varphi_{\text{init}} \Rightarrow \varphi_{\text{final}}$.

Our proof objects have a *linear structure*, which implies a nice separation of concerns. Indeed, Item 1 is only about matching logic and is *not specific* to any programming languages/language tasks, so we only need to develop and proof-check it *once and for all*. Item 2 is specific to the language semantics Γ^L but is independent of the actual program executions, so it can be reused in the proof objects of various language executions for the same programming language L .

A Trustworthy Proof Checker. A proof checker is a small program that checks whether the formal proofs encoded in a proof object are correct. The proof checker is the main trust base of our work. In this paper, we use Metamath [23]—a third-party proof checking tool that is simple, fast, and trustworthy—to formalize matching logic and encode its formal proofs.

Blockchain of Truth (BoT). BoT is a blockchain where logical theories (as language definitions), logical formulas (as correctness claims), and formal proofs (as proof objects) are added and referred by their unique addresses. The underlying validator of BoT is a proof checker, that formally checks the proof object of a claim and changes its status from “claim” to “truth”. The underlying business model (Section 6.3) of BoT provides incentives for users to add proofs.

BoT is ruled by the proof checker, which is the only program executed on BoT. The proof checker, as discussed in Section 6.1, has a very small trust base with only 250 lines of code and is open source [19]. It is efficient, and proof steps can be checked in parallel. Any programming language and any program are patterns stored at some BoT addresses. Any claimed computation or property of any program in any programming language is a theorem, or truth, which must pass the proof checker to be deployed on BoT, giving us the ultimate decentralized trust.

4 Comparing BoT to the State of the Art

As a fully-decentralized store for the correctness claims and proofs about programming languages and program properties, BoT has many unique advantages compared to the state of the art.

BoT versus Databases. Databases are a classical technique to organize data that is stored and accessed electronically from a computer system. The difference between BoT and databases is that BoT inherently implements *decentralized trust*, which is what blockchains were created for. This decentralized trust makes the correctness claims and their proofs that are stored on BoT more trustworthy than those stored in a database. Indeed, to trust the proof of a claim in a database, we need to recheck the entire proof, including all the lemmas proved by the others and used in the proof, which is a highly expensive activity.

The decentralized trust offered by BoT also encourages collaboration between multiple parties. For example, two financial institutions may use each other’s claims (e.g., about an evolving blockchain language) and/or jointly prove (or pay for) the claims. BoT implements a business model (Section 6.3) that incentivizes third-party participants to generate proofs for existing on-chain claims and upload their proofs to BoT. This way, BoT becomes a history record of the contribution to the decentralized trust and thus enforces payment/award appropriately to all parties whenever a new truth is established and published on-chain.

BoT versus Theorem Provers. Theorem provers/assistants such as Coq [22], Agda [24], Isabelle [33], and Lean [13] are common in defining formal language definitions and proving program properties. Compared to BoT, these theorem provers often have a more complex logical foundation, and as a result, a more complex proof checker as their trust base. For example, the underlying logical foundation of Coq is the calculus of inductive constructors (CIC) [9], which is an extension of the calculus of constructors [36] with primitive

```

1  module IMP-SYNTAX
2    imports DOMAINS-SYNTAX
3    syntax Exp ::=
4      Int
5      | Id
6      | Exp "+" Exp    [left, strict]
7      | Exp "-" Exp    [left, strict]
8      | "(" Exp ")"    [bracket]
9    syntax Stmt ::=
10     Id "=" Exp ";" [strict(2)]
11     | "if" "(" Exp ")"
12       Stmt Stmt    [strict(1)]
13     | "while" "(" Exp ")" Stmt
14     | "{" Stmt "}"  [bracket]
15     | "{" "}"
16     > Stmt Stmt    [left, strict(1)]
17   syntax Pgm ::= "int" Ids ";" Stmt
18   syntax Ids ::= List{Id, ","}
19   endmodule

20 module IMP imports IMP-SYNTAX
21   imports DOMAINS
22   syntax KResult ::= Int
23   configuration
24     <T> <k> $PGM:Pgm </k>
25     <state> .Map </state> </T>
26   rule <k> X:Id => I ...</k>
27     <state>... X |-> I ...</state>
28   rule I1 + I2 => I1 +Int I2
29   rule I1 - I2 => I1 -Int I2
30   rule <k> X = I:Int => I ...</k>
31     <state>... X |-> ( _ => I ) ...</state>
32   rule {} S:Stmt => S
33   rule if(I) S _ => S requires I /=Int 0
34   rule if(0) _ S => S
35   rule while(B) S => if(B) {S while(B) S} {}
36   rule <k> int (X, Xs => Xs) ; S </k>
37     <state>... ( . => X |-> 0 ) </state>
38   rule int .Ids ; S => S
39   endmodule

```

Figure 3: The complete \mathbb{K} formal definition of an imperative language IMP.

support for defining inductive datatypes. As a result, the trust base of Coq (i.e., the internal Coq code that does type checking in CIC) has over 30,000 lines of OCaml [34].

In contrast, BoT has *matching logic* as its logical foundation, which is much simpler than the logical foundations of the above-mentioned theorem provers. As we discuss in Section 6.1, the entire syntax and proof system of matching logic can be completely formalized using only 250 lines of code in Metamath [23], which is a tiny formal language to define abstract mathematics and formal systems, whose proof checkers have only 350 lines of Python, or 400 lines of Haskell, or 74 lines of Mathematica [35]. Thanks to matching logic, BoT has the minimal trust base among all the theorem provers.

It is worth mentioning that the simplicity of matching logic does not come at a loss of its expressiveness. Matching logic handles formal language definitions, computations (e.g., program executions), and property claims by logical axioms, formulas, and formal proofs. In [7, 8], it is shown that both FOL extended with least fixpoints and type systems (such as the logic underlying Coq) can be defined as theories in matching logic. No Curry-Howard isomorphism required, and thus there is no need to verify or trust “program generation from proofs”. The vision of an ideal language framework ensures that there is no modeling gap of languages or of programs, because language implementations and tools are generated directly from the formal language definitions, which are matching logic theories. We put it in one slogan: “run what you prove, and prove what you run.” Although it is true that we can use theorem provers to *carry out* matching logic proofs and yield matching logic proof objects, but our proof objects based on \mathbb{K} are more suitable for BoT.

Matching Logic versus Other Logics. The proposed research idea is not limited to matching logic and can be used with any logic that is expressive enough to define language semantics and program properties and for which a proof checker is given. Compared to many existing logics (such as FOL, modal logic, and type systems), matching logic does have its unique advantages, which we discuss in Section 5.2. The main advantage is that matching logic has been specifically designed for specifying and reasoning about programming languages and program properties, and is implemented by \mathbb{K} , which has been used to define a plethora of real-world languages.

5 Prior Results

5.1 \mathbb{K} Framework

\mathbb{K} is an effort in realizing the ideal language framework vision in Figure 2. \mathbb{K} is the result of 20 years of continuous research and development, and has been successfully applied to define the formal semantics of many real-world languages (C [17], Java [3], JavaScript [25], Python [15], Ethereum virtual machine bytecode [18], x86-64 [12], etc.). Language tools for these languages, such as parsers, interpreters, debuggers, symbolic execution engines, deductive verifiers, model checkers, etc., have been automatically generated from their \mathbb{K} formal semantics, which has resulted in several commercial industrial products [16, 21].

Formal language definitions written in \mathbb{K} are human-readable, intuitive, modular, and compact. Figure 3 is an example \mathbb{K} definition of an imperative language, IMP. This complete language definition has only 39 lines, but it includes the entire formal definitions of the syntax, computation configurations, and (operational) semantics of IMP. From this 39-line language definition, \mathbb{K} can generate all language tools for IMP, including its parser, interpreter, program verifier, etc., all correct-by-construction (the program verifier is also relatively complete [10]).

The key components of a \mathbb{K} definition are the following:

1. *Formal programming language syntax* (Figure 3, lines 3–18).

\mathbb{K} uses the traditional BNF grammar to define the formal syntax of programming languages.

2. *Computation configurations* (Figure 3, lines 23–25).

\mathbb{K} uses (computation) configurations to organize the semantic information that is needed to execute programs. For example, a configuration of IMP consists of a piece of IMP code to be executed and a program state, i.e., a mapping from program variables to their values. Real-world languages have more complex configurations that maintain semantic information about heaps, threads, function stacks, exception stacks, etc.

3. *Operational semantics as rewrite rules* (Figure 3, lines 26–38).

\mathbb{K} uses rewrite rules, possibly with conditions, to define the formal operational semantics of the programming languages.

To sum up, \mathbb{K} is a meta-language that can be used to formally define other programming languages. The \mathbb{K} language definition (such as Figure 3) forms a complete and executable specification of a programming language, from which all language tools are automatically generated.

Applying \mathbb{K} on Blockchains. There is a series of recent work on using \mathbb{K} to formalize blockchain languages, with the explicit goal to generate language tools for the respective languages. \mathbb{K} has been applied to the development, deployment, and formal verification of *smart contracts* (i.e., computer programs running on the blockchains), as a reaction to the great demand for formal methods and rigorous program analysis tools in the field [11, 5, 4, 32, 1]. Complete formal semantics for the Ethereum virtual machine bytecode (EVM) has been defined in \mathbb{K} [18], from which an EVM interpreter is autogenerated and is tested on over 40,000 EVM programs in the official language test suite. The experiment reported that \mathbb{K} generated a *faster* EVM interpreter than many hand-written reference implementations of EVM, which suggests that it is realistic to generate virtual machines for blockchains from their formal semantics, because performance is no longer a main obstacle issue. The \mathbb{K} semantics for EVM powers a commercial toolkit called Firefly [30] that aims at improving the quality of Ethereum smart contracts.

A Simple \mathbb{K} Running Example. We introduce a running example of a tiny language (defined in \mathbb{K}) that mimics the behavior of a state machine that computes the sum from 1 to (any integer) input m , shown in Figure 4. The \mathbb{K} definition defines the language syntax of states that are simply integer pairs of the form $\langle m, n \rangle$. The (only) rewrite rule that defines the operational semantics of this tiny language is the following

```

module TWO-COUNTERS imports DOMAINS
  syntax State ::= "<" Int "," Int ">"
  rule <M, N> => <M -Int 1, N +Int M>
    requires M >Int 0
endmodule

```

Figure 4: A tiny language that implements a simple state machine that can be used to compute the sum from 1 to input n .

(also shown in Figure 4, lines 3–4):

$$\langle m, n \rangle \Rightarrow \langle m - 1, n + m \rangle \quad \text{if } m > 0 \quad (3)$$

Therefore, if the initial state is $\langle 100, 0 \rangle$, we will obtain the following program execution trace:

$$\langle 100, 0 \rangle \Rightarrow \langle 99, 100 \rangle \Rightarrow \langle 98, 199 \rangle \Rightarrow \dots \Rightarrow \langle 1, 5049 \rangle \Rightarrow \langle 0, 5050 \rangle \quad (4)$$

Although simple, this running example utilizes the same core \mathbb{K} features (i.e., defining syntax using BNF and semantics using rewrite rules) as the more complex real-world languages.

5.2 Matching Logic: The Logical Foundation of \mathbb{K}

Matching logic was proposed in [27] as a means to specify and reason about programs compactly and modularly, using a formalism that keeps and respects the syntax and the semantic structure of programs, without encodings or translations. The key concept is its formulas, called *patterns*, which are used to specify program syntax and semantics in a uniform way across various programming paradigms. In [26, 7, 8, 6], the authors show how matching logic captures FOL, FOL-lfp, separation logic, modal logic, temporal logics, Hoare logic, λ -calculus, type systems, etc.

The power of matching logic lies in its simplicity and expressiveness. In terms of simplicity, matching logic follows a minimalist design. For example, the syntax of matching logic patterns, shown below, has only 8 syntactic constructs that define the most basic concepts that are necessary to serve as the logical foundation of a language framework:

$\varphi ::= x$	// element variables
X	// set variables
σ	// symbols (from a user-provided signature)
$\varphi_1 \varphi_2$	// application
\perp	// bottom
$\varphi_1 \rightarrow \varphi_2$	// implication
$\exists x. \varphi$	// quantification
$\mu X. \varphi$	// least fixpoints

Every syntactic construct has a unique purpose. Element variables refer to individual elements in models, like FOL variables. Set variables refer to sets of elements in the models, like propositional variables in modal logic. Symbols are provided by the users and are used to represent constructors, functions, predicates, relations, and so on, whose actual semantics is determined by the axioms and theories. The application construct applies a symbol (or any pattern in general) to an argument pattern. Bottom and implication allow us to build logical constraints. Quantification allows us to create abstraction. Least fixpoints allow us to define induction and recursion.

Besides the syntax, matching logic has a Hilbert-style proof system (Figure 6) that derives proof judgments $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system from the axioms in Γ . We call Γ a *matching logic theory*. The matching logic proof system is critical to proof objects.

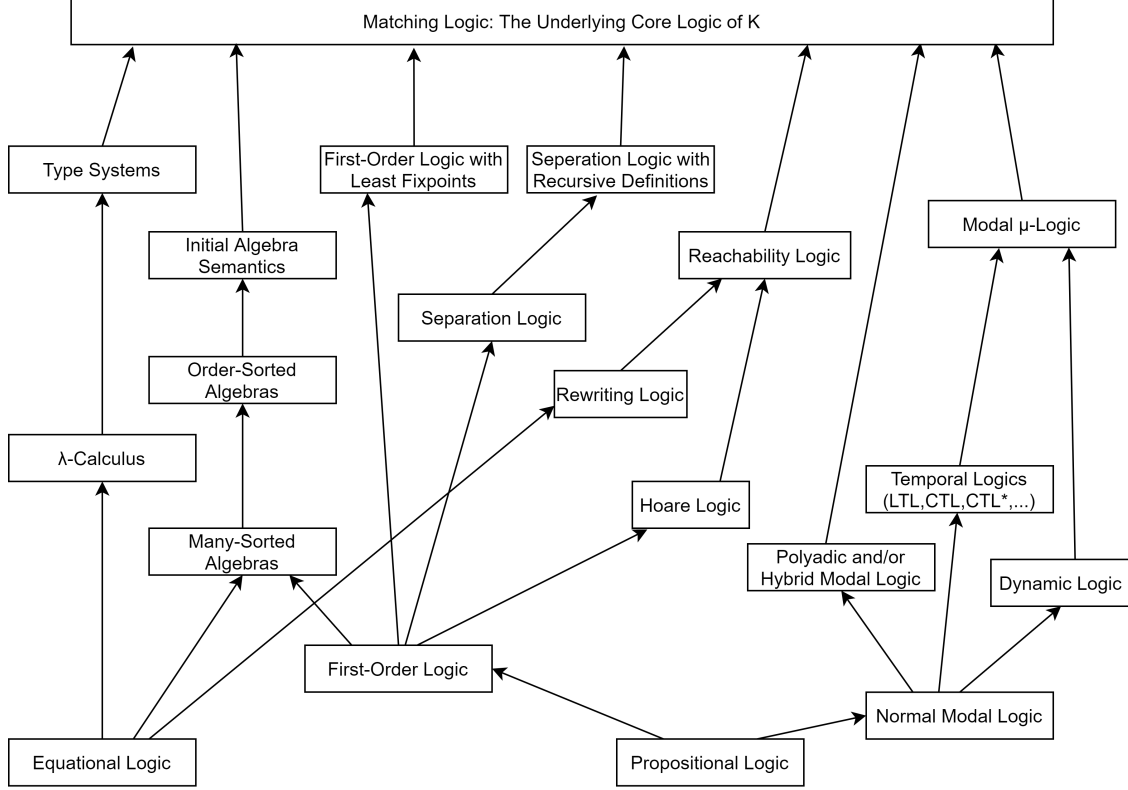


Figure 5: Matching logic is the logical foundation of \mathbb{K} . Its logical theories can capture various logics, calculi, and semantic approaches used in programming language design and analysis [26, 7, 6, 8].

Matching logic proof rules are sound [7] and can be divided into 4 categories: FOL reasoning, frame reasoning, fixpoint reasoning, and some technical rules. The FOL reasoning rules provide (complete) FOL reasoning (see, e.g., [31]). The frame reasoning rules state that application contexts are commutative with disjunctive connectives such as \vee and \exists . The fixpoint reasoning rules support the standard fixpoint reasoning as in modal μ -calculus [20]. The technical proof rules are needed for some completeness results (see [7] for details).

6 Proposed Work

6.1 Matching Logic Proof Checker

Matching logic is the formal language and logical foundation underlying BoT. Correctness claims are expressed as matching logic formulas, called *patterns* (see Section 5.2), and their proofs are encoded as matching logic proof objects. Therefore, the first component in the proposed work is a trustworthy matching logic *proof checker*.

We propose to use *Metamath* to formalize the syntax and proof system of matching logic and to implement the matching logic proof checker. Metamath [23] is a tiny language to state abstract mathematics and their proofs in a machine-checkable style. In our work, we use Metamath to formalize matching logic and to encode our proof objects. We choose Metamath for its simplicity and fast proof checking: Metamath proof checkers are often hundreds lines of code and can proof-check thousands of theorems in a second.

The matching logic proof checker is essentially a formalization of the syntax and proof rules of matching logic in Metamath. Figure 7 shows an extract of the 250-LOC Metamath formalization of matching logic.

FOL Rules	{	(PROPOSITIONAL 1)	$\varphi \rightarrow (\psi \rightarrow \varphi)$
		(PROPOSITIONAL 2)	$(\varphi \rightarrow (\psi \rightarrow \theta)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \theta))$
		(PROPOSITIONAL 3)	$((\varphi \rightarrow \perp) \rightarrow \perp) \rightarrow \varphi$
		(MODUS PONENS)	$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$
		(\exists -QUANTIFIER)	$\varphi[y/x] \rightarrow \exists x . \varphi$
		(\exists -GENERALIZATION)	$\frac{\varphi \rightarrow \psi}{(\exists x . \varphi) \rightarrow \psi} \quad x \notin FV(\psi)$
<hr/>			
Frame Rules	{	(PROPAGATION $_{\perp}$)	$C[\perp] \rightarrow \perp$
		(PROPAGATION $_{\vee}$)	$C[\varphi \vee \psi] \rightarrow C[\varphi] \vee C[\psi]$
		(PROPAGATION $_{\exists}$)	$C[\exists x . \varphi] \rightarrow \exists x . C[\varphi]$ with $x \notin FV(C)$
		(FRAMING)	$\frac{\varphi \rightarrow \psi}{C[\varphi] \rightarrow C[\psi]}$
<hr/>			
Fixpoint Rules	{	(SUBSTITUTION)	$\frac{\varphi}{\varphi[\psi/X]}$
		(PREFIXPOINT)	$\varphi[(\mu X . \varphi)/X] \rightarrow \mu X . \varphi$
		(KNASTER-TARSKI)	$\frac{\varphi[\psi/X] \rightarrow \psi}{(\mu X . \varphi) \rightarrow \psi}$
<hr/>			
Technical Rules	{	(EXISTENCE)	$\exists x . x$
		(SINGLETON)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg \varphi])$

Figure 6: Matching logic proof system (where C, C_1, C_2 are application contexts; that is, the path from the root of $C[\square]$ to the placeholder variable \square has only applications, and not other logical connectives).

We declare five constants in a row in line 1, where `\imp`, `(`, and `)` build the syntax, `#Pattern` is the type of patterns, and `|-` is the provability relation. We declare three metavariables of patterns in lines 3-6, and the syntax of implication $\varphi_1 \rightarrow \varphi_2$ as `(\imp ph1 ph2)` in line 7. Then, we define matching logic proof rules as Metamath axioms. For example, lines 18-22 define the rule (MODUS PONENS). Line 23 shows the (meta-)theorem that states that $\vdash \varphi \rightarrow \varphi$ holds, accompanied by its machine-checkable proof (lines 25–43).

Our implementation of the matching logic proof checker follows closely the definitions of matching logic syntax and proof rules. The resulting formalization has only 250 lines of code [19], which is therefore the entire trust base of BoT.

6.2 Proof Object Generation for \mathbb{K}

6.2.1 Kore: The Intermediate Between \mathbb{K} and Matching Logic.

The \mathbb{K} compilation tool `kompile` (explained shortly) is what compiles a \mathbb{K} language definition into the matching logic theory Γ^L , written in a formal language called Kore. For legacy reasons, the Kore language is not the same as the syntax of matching logic but is an axiomatic extension with equality, sorts, sorted functions, and rewriting. Therefore, to formalize Γ^L in our proof objects, we need to (1) formalize the basic matching logic theories that we showed above; and (2) automatically translate Kore definitions into the corresponding matching logic theories.

Phase 1: From \mathbb{K} to Kore. To compile a \mathbb{K} definition such as `two-counters.k` in Figure 4, we pass it to the \mathbb{K} compilation tool `kompile` as follows:

```
$ kompile two-counters.k
```

The result is a compiled Kore definition `two-counters.kore`. We show the autogenerated Kore axiom in

```

1  $c \imp ( ) #Pattern |- $.
2
3  $v ph1 ph2 ph3 $.
4  ph1-is-pattern $f #Pattern ph1 $.
5  ph2-is-pattern $f #Pattern ph2 $.
6  ph3-is-pattern $f #Pattern ph3 $.
7  imp-is-pattern
8    $a #Pattern ( \imp ph1 ph2 ) $.
9
10 axiom-1
11   $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.
12
13 axiom-2
14   $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
15             ( \imp ( \imp ph1 ph2 )
16                   ( \imp ph1 ph3 ) ) ) $.
17
18 ${
19   rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
20   rule-mp.1 $e |- ph1 $.
21   rule-mp   $a |- ph2 $.
22 }$

```

```

23 imp-refl $p |- ( \imp ph1 ph1 )
24 $=
25   ph1-is-pattern ph1-is-pattern
26   ph1-is-pattern imp-is-pattern
27   imp-is-pattern ph1-is-pattern
28   ph1-is-pattern imp-is-pattern
29   ph1-is-pattern ph1-is-pattern
30   ph1-is-pattern imp-is-pattern
31   ph1-is-pattern imp-is-pattern
32   imp-is-pattern ph1-is-pattern
33   ph1-is-pattern ph1-is-pattern
34   imp-is-pattern imp-is-pattern
35   ph1-is-pattern ph1-is-pattern
36   imp-is-pattern imp-is-pattern
37   ph1-is-pattern ph1-is-pattern
38   ph1-is-pattern imp-is-pattern
39   ph1-is-pattern axiom-2
40   ph1-is-pattern ph1-is-pattern
41   ph1-is-pattern imp-is-pattern
42   axiom-1 rule-mp ph1-is-pattern
43   ph1-is-pattern axiom-1 rule-mp
44   $.

```

Figure 7: An extract of the Metamath formalization of matching logic (i.e., the proof checker).

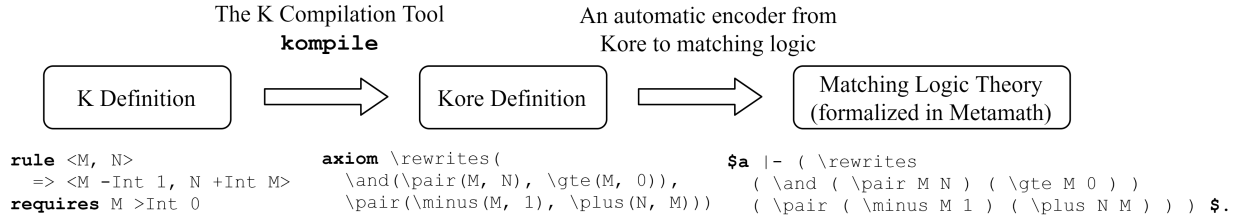


Figure 8: Automatic translation from \mathbb{K} to matching logic, via Kore

Figure 8 that corresponds to the rewrite rule in Equation (3). As we can see, Kore is a much lower-level language than \mathbb{K} , where the programming language concrete syntax and \mathbb{K} 's front end syntax are parsed and replaced by the abstract syntax trees, represented by the constructor terms.

Phase 2: From Kore to Matching Logic. We develop an automatic encoder that translates Kore syntax into matching logic patterns. Since Kore is essentially the theory of equality, sorts, and rewriting, we can define the syntactic constructs of the Kore language as *notations*.

6.2.2 Proof Generation for Program Executions.

Consider the following \mathbb{K} definition that consists of K (conditional) rewrite rules:

$$S = \{t_k \wedge p_k \Rightarrow s_k \mid k = 1, 2, \dots, K\}$$

where t_k and s_k are the left- and right-hand sides of the rewrite rule, respectively, and p_k is the rewriting condition. Consider the following execution trace:

$$\varphi_0, \varphi_1, \dots, \varphi_n \tag{5}$$

where $\varphi_0, \dots, \varphi_n$ are snapshots. We let \mathbb{K} generate the following proof parameter:

$$\Theta \equiv (k_0, \theta_0), \dots, (k_{n-1}, \theta_{n-1}) \tag{6}$$

where for each $0 \leq i < n$, k_i denotes the rewrite rule that is applied on φ_i ($1 \leq k_i \leq K$) and θ_i denotes the corresponding substitution such that $t_{k_i}\theta_i = \varphi_i$.

As an example, the rewrite rule of **TWO-COUNTERS**, restated below:

$$\langle m, n \rangle \Rightarrow \langle m - 1, n + m \rangle \quad \text{if } m > 0$$

has the left-hand side $t_k \equiv \langle m, n \rangle$, the right-hand side $s_k \equiv \langle m - 1, n + m \rangle$, and the condition $p_k \equiv m \geq 0$. Note that the right-hand side pattern s_k contains the arithmetic operations “+” and “−” that can be further evaluated to a value, if concrete instances of the variables m and n are given. Generally speaking, the right-hand side of a rewrite rule may include (built-in or user-defined) functions that are not constructors and thus can be further evaluated. We call such evaluation process a *simplification*.

In the following, we list all proof objects for one-step executions.

$$\begin{array}{ll} \Gamma^L \vdash \varphi_0 \Rightarrow s_{k_0} \theta_0 & // \text{ by applying } t_{k_0} \wedge p_{k_0} \Rightarrow s_{k_0} \text{ using } \theta_0 \\ \Gamma^L \vdash s_{k_0} \theta_0 = \varphi_1 & // \text{ by simplifying } s_{k_0} \theta_0 \\ \dots & \\ \Gamma^L \vdash \varphi_{n-1} \Rightarrow s_{k_{n-1}} \theta_{n-1} & // \text{ by applying } t_{k_{n-1}} \wedge p_{k_{n-1}} \Rightarrow s_{k_{n-1}} \text{ using } \theta_{n-1} \\ \Gamma^L \vdash s_{k_{n-1}} \theta_{n-1} = \varphi_n & // \text{ by simplifying } s_{k_{n-1}} \theta_{n-1} \end{array}$$

As we can see, there are two types of proof objects: one that proves the results of *applying rewrite rules* and one that *applies simplification*.

Rewriting. The main steps in proving $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i} \theta_i$ are (1) to *instantiate* the rewrite rule $t_{k_i} \wedge p_{k_i} \Rightarrow s_{k_i}$ using the substitution

$$\theta_i = [c_1/x_1, \dots, c_m/x_m]$$

given in the proof parameter, and (2) to show that the (instantiated) rewriting condition $p_{k_i} \theta_i$ holds. Here, x_1, \dots, x_m are the variables that occur in the rewrite rule and c_1, \dots, c_m are terms by which we instantiate the variables. For (1), we need to first prove the following lemma, called (FUNCTIONAL SUBSTITUTION) in [7], which states that \forall -quantification can be instantiated by functional patterns:

$$\frac{\forall \vec{x}. t_{k_i} \wedge p_{k_i} \Rightarrow s_{k_i} \quad \exists y_1. \varphi_1 = y_1 \quad \dots \quad \exists y_m. \varphi_m = y_m}{t_{k_i} \theta_i \wedge p_{k_i} \theta_i \Rightarrow s_{k_i} \theta_i} \quad y_1, \dots, y_m \text{ fresh}$$

Intuitively, the premise $\exists y_1. \varphi_1 = y_1$ states that φ_1 is a functional pattern (it equals an element y_1).

If Θ in Equation (6) is the correct proof parameter, θ_i is the correct substitution and thus $t_{k_i} \theta_i \equiv \varphi_i$. Therefore, to prove the original proof goal for one-step execution, i.e. $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i} \theta_i$, we only need to prove that $\Gamma^L \vdash p_{k_i} \theta_i$, i.e., the rewriting condition p_{k_i} holds under θ_i . This is done by *simplifying* $p_{k_i} \theta_i$ to \top , discussed together with the simplification process in the following.

Simplification. \mathbb{K} carries out simplification exhaustively before trying to apply a rewrite rule, and simplifications are done by applying equations. Generally speaking, let s be a functional pattern and $p \rightarrow t = t'$ be a (conditional) equation, we say that s can be *simplified* w.r.t. $p \rightarrow t = t'$, if there is a sub-pattern s_0 of s (written $s \equiv C[s_0]$ where C is a context) and a substitution θ such that $s_0 = t\theta$ and $p\theta$ holds. The resulting *simplified pattern* is denoted $C[t'\theta]$. Therefore, a proof object of the above simplification consists of two proofs: $\Gamma^L \vdash s = C[t'\theta]$ and $\Gamma^L \vdash p\theta$. The latter can be handled recursively, by simplifying $p\theta$ to \top , so we only need to consider the former.

6.3 A Business Model of BoT

As a decentralized system, BoT is equipped with a business model that provides incentives for its users to upload formal language definitions, correctness claims about program properties, and proof objects. The main principle of the business model is summarized as follows:

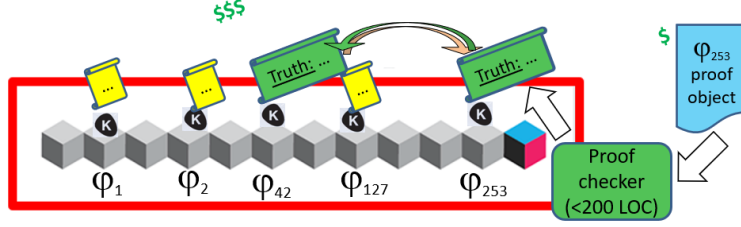
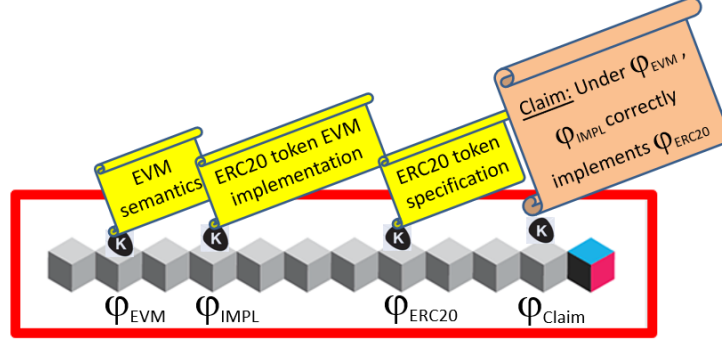


Figure 9: The business model provides incentives for users to upload proof objects for the pending claims.



$$\varphi_{\text{Claim}} \cong \varphi_{\text{EVM}} \rightarrow (\varphi_{\text{IMPL}} \rightarrow \varphi_{\text{ERC20}})$$

Figure 10: Formal verification as mathematical proof.

1. Formal language definitions are uploaded to BoT by language creators or committees.
2. Correctness claims can be uploaded to BoT with a fee locked for proof finders. This way, stakeholders interested in making claims, for example a business entity claiming that their smart contract is correct, can pay a price to whoever proves the claim in the future.
3. Proof providers get rewarded by working out the proof objects for open claims and upload them to BoT.

To encourage *proof reuse*, e.g., by firstly proving some important lemmas and then finish the proofs using the lemmas, the business model assigns part of the fees to those who prove the lemmas. In other words, it is encouraged to prove lemmas and use existing lemmas when creating proof objects (instead of inlining the proofs of the lemmas).

7 Use Cases of BoT

7.1 Trusted Formal Verification

In Figure 10, we show an example of *formal verification* that is stored, claimed, and proved on BoT. Logical formulas φ_{EVM} and φ_{IMPL} have the same meanings as above. The formula φ_{ERC20} formalizes the ERC20 token specification. The claim formula φ_{claim} states that the implementation conforms to the ERC20 specification w.r.t. the formal semantics of EVM.

7.2 Rating of Digital Assets

In BoT, language definitions, correctness claims, and proof objects are all digital assets, uniformly stored and maintained on-chain. Based on how a correctness claim is established on BoT, we can assign *trustworthiness rating* to the digital assets that appear on BoT. Depending on how a correctness claim is proved, BoT allows for various degrees of truth, backed by proof service providers:

- **A+**: a proof object for the claim provided and stored on BoT.
- **A**: formal modeling and validation of model.
- **A-**: formal verification of code functional correctness.
- **B**: specifications provided and human security audits.
- **C**: automated static and symbolic execution analysis tools.
- **D**: Passing tests provided with high (>90%) code coverage.
- **F**: No evidence provided.

References

- [1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, Berlin, Heidelberg, 2017. Springer.
- [2] Denis Bogdanas and Grigore Roşu. K-java: A complete semantics of java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 445–456, 2015.
- [3] Denis Bogdănaş and Grigore Roşu. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL’15)*, pages 445–456, Mumbai, India, 2015. ACM.
- [4] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. *Hacking Distributed*, 1(1):1–1, 2017. <http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/>.
- [5] Vitalik Buterin. Thinking about smart contract security. *Ethereum Blog*, 1(1):1–1, 2016. <https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/>.
- [6] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Initial algebra semantics in matching logic. Technical Report <http://hdl.handle.net/2142/107781>, University of Illinois at Urbana-Champaign, July 2020.
- [7] Xiaohong Chen and Grigore Roşu. Matching μ -logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS’19)*, pages 1–13, Vancouver, Canada, 2019. IEEE.
- [8] Xiaohong Chen and Grigore Roşu. A general approach to define binders using matching logic. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP’20)*, pages 1–32, New Jersey, USA, 2020. ACM.
- [9] Coq Team. Coq documents: calculus of inductive constructions. Online at <https://coq.inria.fr/refman/language/cic.html>, 2020.

- [10] Andrei Ștefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roșu. 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)*, pages 74–91. ACM, 2016.
- [11] Phil Daian. DAO attack, 2016. <http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>.
- [12] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roșu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, pages 1133–1148, Phoenix, Arizona, USA, 2019. ACM.
- [13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction Automated Deduction (CADE'15)*, pages 378–388, Cham, 2015. Springer International Publishing.
- [14] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.
- [15] Dwight Guth. A formal semantics of python 3.3. 2013.
- [16] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roșu. RV-Match: Practical semantics-based program analysis. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, volume 9779, pages 447–453. Springer, 2016.
- [17] Chris Hathhorn, Chucky Ellison, and Grigore Roșu. Defining the undefinedness of C. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345, Portland, OR, 2015. ACM.
- [18] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ștefănescu, and Grigore Roșu. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*, pages 204–217, Oxford, UK, 2018. IEEE. <http://jellopaper.org>.
- [19] K Team. Matching logic proof checker. Available at <https://github.com/kframework/matching-logic-proof-checker/blob/main/matching-logic-250-loc.mm>, 2021.
- [20] Dexter Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.
- [21] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O’Neil Meredith, Traian Florin Șerbănuță, and Grigore Roșu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Proceedings of the 5th International Conference on Runtime Verification (RV'14)*, pages 285–300, 2014.
- [22] Coq Team. *The Coq proof assistant*. LogiCal Project, 2020.
- [23] Norman Megill and David A. Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu.com, 2019.
- [24] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'09)*, pages 230–266, Heijten, The Netherlands, 2009. Springer.

- [25] Daejun Park, Andrei Ștefănescu, and Grigore Roșu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356, Portland, OR, 2015. ACM.
- [26] Grigore Roșu. Matching logic. *Logical Methods in Computer Science*, 13(4):1–61, 2017.
- [27] Grigore Roșu and Wolfram Schulte. Matching logic—extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.
- [28] Grigore Rosu. K—A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*. IOS Press, 2017.
- [29] Grigore Roșu and Traian Florin Șerbănuță. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
- [30] Runtime Verification, Inc. Firefly: Quality assurance for ethereum smart contract developers. Published at <https://fireflyblockchain.com/>.
- [31] Joseph R. Shoenfield. *Mathematical logic*. Addison-Wesley Pub. Co, 1967.
- [32] Jutta Steiner. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. <http://goo.gl/LBh1vR>.
- [33] The Isabelle development team. Isabelle, 2018. <https://isabelle.in.tum.de/>.
- [34] Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85–100, 2012. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10).
- [35] Wolfram Research, Inc. Mathematica, Version 12.2. Available at <https://www.wolfram.com/mathematica>, 2020. Champaign, IL.
- [36] Luo Zhaohui. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, 1990. Available at <http://hdl.handle.net/1842/12487>.