Proof of Proof:
A Universal Verifiable Computing Framework
Version 1.0

Pi Squared Inc.

February 2025

It is suggested that the reader first read "The Pi Squared Whitepaper" [31].

Abstract

This paper gives an overview of the Proof of Proof approach to universal and correct-by-construction verifiable computing proposed by the Pi Squared team. The idea of Proof of Proof is to separate the three underlying concerns: *computation*, *verification*, and *cryptography*.

First, recent developments in executable formal semantics allow us to efficiently and completely automatically reduce computation to mathematical proof. The universality of Proof of Proof comes from the fact that there is only one language for mathematical proofs, which works with math proofs corresponding to any computations done with any programs in any programming languages (PLs) or virtual machines (VMs).

Second, the generated math proofs are verified, not trusted, with a disarmingly simple and small proof checker of only a few hundred lines of code. The correctness of Proof of Proof comes from the fact that no (usually complex and error-prone) compilers, interpreters, or even formal provers or language frameworks need to be trusted or formally verified: all these become only instruments to assist the generation of math proofs; the math proofs, and not the tools that produced them, are the ultimate correctness arguments for the computations from which they were derived.

Finally, recent developments in cryptography, e.g., SNARKS, STARKS and zero-knowledge (ZK), allow us to implement the math proof checker as a cryptographic circuit, which effectively allows us to produce ZK proofs for the integrity of the math proofs, that is, (ZK) Proofs of (math) Proofs.

This paper does not discuss semantics-based execution in-depth, nor recent developments in the context of the $\mathbb{K}$ framework that make semantics-based execution comparable in performance with manual, adhoc language implementations. If the reader is interested in how a formal reasoning engine like $\mathbb{K}$ can execute programs as fast as or faster than dedicated interpreters, e.g., EVM programs as fast as or faster than Geth, they should refer to our white paper dedicated to that topic: *"Semantics-based execution and the LLVM backend of the $\mathbb{K}$ Framework"*. This paper only focuses on how to extract the math proofs from $\mathbb{K}$, verify them, and generate ZK proofs from them.

Table of Contents

1 Introduction

Proof of Proof is a universal, verifiable computing framework for all programming languages, virtual machines (VMs), and instruction set architectures (ISAs). Its universality comes from a semantics-based approach. In this approach, any programming language has a formal semantics, which is a rigorous, complete, and executable mathematical definition that specifies all behaviors of all programs in that language. This formal semantics is passed as an input to the Proof of Proof framework for two purposes: to execute programs in that language and to generate verifiable proofs for program execution. Because Proof of Proof is directly based on the formal semantics of programming languages, it is universal and language-agnostic (also known as language-independent and language-parametric).

Figure 1 shows the Proof of Proof workflow. To obtain a proof for the execution of a program P in a programming language L within certain execution environment, one should first prepare a formal semantics of L. This formal semantics, denoted by Γ^L, is written in an open-source universal language framework called the $\mathbb{K}$ framework. The formal semantics Γ^L then serves two purposes. Firstly, it is used by $\mathbb{K}$ to automatically derive an interpreter for L. Secondly, it is used as a basis for formal reasoning, for constructing formal proofs for the execution of programs in L.

Proof of Proof reduces arbitrary computation of any program in any programming language into a unifying and universal domain: mathematics. More specifically, mathematical logic. Proof of Proof is based on matching logic as its underlying logical foundation. A formal semantics Γ^L of a programming language L is a logical theory in matching logic that consists of mathematical axioms that specify the behaviors of all programs in the language. A concrete execution trace of a program P within a certain execution environment E is represented as a logical claim to be proved. Intuitively, the claim has the form $\gamma_{init} \Rightarrow \gamma_{final}$, where γ_{init} and γ_{final} represent the initial and final configurations at the beginning and at the end of the execution of P, respectively. Then, Proof of Proof generates a mathematical proof Π for the claim. We denote it as

$$\Pi : \Gamma^L \vdash \gamma_{init} \Rightarrow \gamma_{final} \tag{1}$$

The mathematical proof Π shows how to construct the claim $\gamma_{init} \Rightarrow \gamma_{final}$ from the axioms in Γ^L using the proof system of matching logic. The proof system of matching logic consists of a fixed number of proof rules. The mathematical proof Π is effectively a transcript that tells how to apply the proof rules in the proof system to construct the claim from the axioms. As a result, Π is often a much larger artifact than P or the configurations γ_{init} and γ_{final}. The process of generating Π from P and its initial and final configurations is called Math Proof Generation, abbreviated as MPG.

Since the mathematical proof Π is large and its usage could be unpractical, Proof of Proof further uses Zero-Knowledge (ZK) Proof technology to reduce its size. While the mathematical proof demonstrates the correctness of a program's execution trace, the corresponding ZK proof π only shows that such a

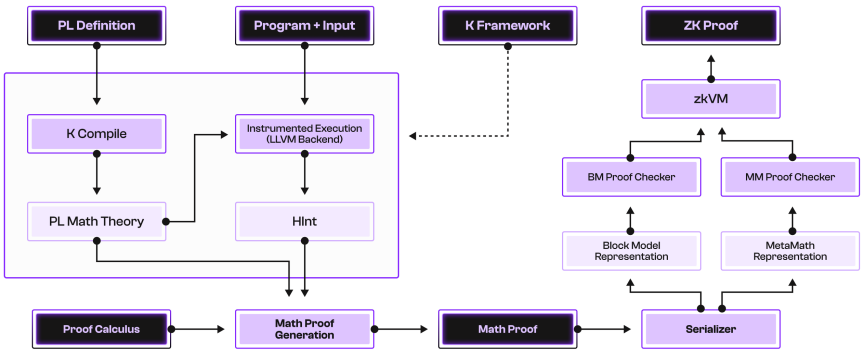

Figure 1: Proof of Proof Workflow

mathematical proof exists. Still, from the user's point of view, the ZK proof is enough. Proof of Proof is compatible with many ZK technologies. In particular, any existing zkVM can be used to generate ZK proofs from mathematical proofs.

Following this introduction, Section 2 provides the prerequisites for understanding Proof of Proof such as the $\mathbb{K}$ framework, matching logic, and zero-knowledge cryptography basics. Section 3 discusses how machine-verifiable mathematical proofs can be generated from a given program execution using its programming language's formal semantics. Section 4 elaborates on a series of experiments conducted to verify different mathematical proofs of existing zkVMs and the lessons we learned. Section 5 outlines our novel block model, an alternative to using zkVMs that allows us to represent and verify mathematical proofs directly and efficiently in ZK without translating to any VM or ISA.

2 Preliminaries

Here, we recall what formal semantics is, how the formal semantics framework $\mathbb{K}$ introduced in 2003 works, the logical foundation underlying $\mathbb{K}$ that allows us to universally reduce computation to mathematical proof, and finally, what zkVMs are and their role in our Proof of Proof approach.

2.1 Formal Semantics and $\mathbb{K}$ Framework

An easy way to understand $\mathbb{K}$ is to look at it as a meta-language, that can implement, or better say, define, other programming languages. In Figure 2, we show an example $\mathbb{K}$ language definition of an imperative language IMP. In the 40-line definition, we *completely* define the formal syntax and the (executable) formal semantics of IMP using a user-friendly front end language (part of $\mathbb{K}$).

We use IMP as an example to illustrate the main $\mathbb{K}$ features. There are two

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

```
20 module IMP
21   imports IMP-SYNTAX 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; => .K ...</k>
31     <state>... X |-> (_ => I) ...</state>
32   rule {} => .K
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 S1:Stmt S2:Stmt => S1 ~> S2
37   rule <k> int (X,Xs => Xs);_ </k>
38        <state> _ (.Map => X|->0) </state>
39   rule int .Ids; S => S
40 endmodule
```

Figure 2: Complete $\mathbb{K}$ Semantics of an Imperative Language

modules: `IMP-SYNTAX` defines the syntax, and `IMP` defines the semantics using rewrite rules. Syntax is defined as BNF grammar. The keyword `syntax` leads to production rules that can have attributes that specify the additional syntactic and/or semantic information. For example, the syntax of `if`-statements is defined in lines 10–11 and has the attribute `[strict(1)]`, meaning that the evaluation order is strict in the first argument, i.e., the condition of an `if`-statement.

In the module `IMP`, we define the *configurations* of IMP and its formal semantics. A configuration (lines 23-25) is a constructor term that has all semantic information needed to execute programs. IMP configurations are simple, consisting of the IMP code and a program state that maps variables to values. We organize configurations using *(semantic) cells*: `</k>` is the cell of IMP code and `</state>` is the cell of program states. In the initial configuration (lines 24-25), `</state>` is empty and `</k>` contains the IMP program that we pass to $\mathbb{K}$ for execution (represented by the special $\mathbb{K}$ variable `$PGM`).

We define formal semantics using *rewrite rules*. In lines 26–27, we define the semantics of variable lookup, where we match on a variable `X` in the `</k>` cell and look up its value `I` in the `</state>` cell by matching on the binding $\texttt{X} \mapsto \texttt{I}$. Then, we rewrite `X` to `I`, denoted by $\texttt{X} \Rightarrow \texttt{I}$ in the `</k>` cell in line 26. Rewrite rules in $\mathbb{K}$ generalize those in other rewrite engines, such as Maude [8], in the sense that they also mention the partial context in which the rewrites happen. That is, they are rewrites within a term called *local rewrites* in $\mathbb{K}$, and there can be more than one in the term – see, for example, the rewrite rule giving the semantics of assignment in lines 30–31.

2.2 Matching Logic

Matching logic [33, 6, 5] provides a unifying framework for defining and reasoning about the semantics of programming languages. A programming language is defined in matching logic as a *logical theory*, i.e., a set of axioms.

Thanks to the $\mathbb{K}$ framework, many real-world programming languages have been *completely defined* as matching logic theories: C [14], Java [3], JavaScript [30], Python [12], Rust [19, 39], Ethereum Virtual Machine (EVM) opcodes [15], x86-64 [11], and LLVM [20], among others. $\mathbb{K}$ provides a suite of tools to generate implementations and formal analysis tools for any programming language from its formal semantics. These implementations and tools include parsers, interpreters, model checkers, symbolic execution engines, and even deductive and inductive program verifiers [34, 9]. Some language tools, such as RV-Match, RV-Monitor, and RV-Predict based on the C semantics in matching logic, have commercialized applications.

On the other hand, matching logic was purposely crafted not only to be expressive enough to support all programming languages and computational models and paradigms, but also to admit the *smallest proof checker* known: it has only 200 lines of code [4]. That is, 200 lines of code that can verify the integrity of any execution of any program in any programming language. It can in fact do significantly more than that, but that is enough for our purpose here.

The syntax of matching logic is quite compact:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi \mid \mu X.\, \varphi \text{ if } \varphi \text{ is positive in } X \quad (2)$$

These 8 syntax constructs[1] build matching logic formulas, called *patterns*, which, semantically speaking, can be *matched* by a set of elements. Patterns can match structures that are of certain shapes, satisfy certain dynamic properties, or meet certain logical constraints, usually all of these together.

Element variables x are FOL-style variables that are necessary for ranging over individual elements, which can then be quantified (i.e., "abstracted") by the $\exists$ binder. *Set variables* X are like propositional variables in modal logic that are necessary for ranging over sets (i.e., predicates), which can then be quantified by μ to create least fixpoints. *Constant symbols* σ are used to represent functions, predicates, constructors, and modal operators in a uniform way. Together with *application*, constant symbols build complex patterns from simpler ones (i.e., $\sigma\,\varphi_1 \ldots \varphi_n$), which can represent terms (e.g., $\sigma \equiv f$ for function f), FOL-style formulas (e.g., $\sigma \equiv p$ for predicate p), program configurations (e.g., σ being the `<\K/>` cell), and modal formulas such as temporal and reachability properties (e.g., σ being the "next" operator $\circ$ in LTL [6]).

2.2.1 Notations

The expressivity of matching logic ($\mathbb{ML}$) can be extended on two dimensions:

1. Many logical frameworks can be subsumed to $\mathbb{ML}$ as theories [7]

[1] Actually there are only 7 core constructs, since $\bot$ can be defined as a notation for $\mu X.\, X$.

2. A domain-specific logic can be defined as an $\mathbb{ML}$ theory using a simple and powerful notation mechanism.

The *notation mechanism* can be expressed as a chain of theories. Unlike the classical approach, we represent a theory as a triple $(\Sigma, \Phi, \vdash)$, where Σ is the alphabet of the constant symbols, Φ a set of formulas (patterns), and $\vdash$ an entailment relation (more on the entailment/provability relation and the proof system of matching logic in Section 2.2.2). A notation-based specification is given as an inclusion theory morphism $(\Sigma, \Phi, \vdash) \to (\Sigma, \Phi', \vdash')$ such that each *new formula* $\varphi' \in \Phi' \setminus \Phi$ is a *notation* of a formula $\varphi \in \Phi$, written as $\varphi' :\leftrightarrow \varphi$ and expressed by two new axioms: $\vdash' \varphi' \to \varphi$ and $\vdash' \varphi \to \varphi'$. Φ' may also include other axioms that constrain the use of notations.

Derived operators as notations

1. **New formulas (patterns):**

$$\varphi ::= \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 \mid \forall x.\, \varphi \mid \nu X.\, \varphi$$

2. **New axioms:**

$$\begin{aligned}
\neg\varphi &:\leftrightarrow \varphi \to \bot && // \text{ negation}\\
\varphi_1 \vee \varphi_2 &:\leftrightarrow (\varphi_1 \to \bot) \to \varphi_2 && // \text{ disjunction}\\
\varphi_1 \wedge \varphi_2 &:\leftrightarrow \neg(\neg\varphi_1 \vee \neg\varphi_2) && // \text{ conjunction}\\
\varphi_1 \leftrightarrow \varphi_2 &:\leftrightarrow (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) && // \text{ equivalence}\\
\forall x.\, \varphi &:\leftrightarrow \neg\exists x.\, \neg\varphi && // \text{ universal quantification}\\
\nu X.\, \varphi &:\leftrightarrow \neg\, \mu X.\, \neg\varphi[\neg X/X] && // \text{ greatest fixpoint}
\end{aligned}$$

Note that the above grammar extends the one over which the notation is defined (here, the original grammar).

Equality and membership as notations

Although the syntax of patterns does not have equality, we can define it as a notation (see [5]). An equality of two patterns φ_1 and φ_2, written $\varphi_1 = \varphi_2$, is equivalent to $\top$ if the same elements match the two patterns. Otherwise, it is equivalent to $\bot$.

Assume that $(\Sigma, \Phi, \vdash)$ includes a symbol $def \in \Sigma$, called the *definedness* symbol, and define the following axiom:

$$\textsc{Definedness} \qquad \vdash \forall x.\, def\ x$$

Intuitively, Definedness states that each individual element is *defined* (i.e., not $\bot$). Thus, for any pattern ψ that is matched by some elements, $def\ \psi$ is $\top$.

1. **New formulas (patterns):**

$$\varphi ::= \lceil\varphi\rceil \mid \lfloor\varphi\rfloor \mid \varphi_1 = \varphi_2 \mid \varphi_1 \subseteq \varphi_2 \mid x \in \varphi$$

2. **New axioms:**

$$\begin{array}{rl} \lceil \varphi \rceil :\leftrightarrow (\mathit{def}\ x) & // \text{ definedness} \\ \lfloor \varphi \rfloor :\leftrightarrow \neg\lceil \neg\varphi \rceil & // \text{ totality} \\ \varphi_1 = \varphi_2 :\leftrightarrow \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor & // \text{ equality} \\ \varphi_1 \subseteq \varphi_2 :\leftrightarrow \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor & // \text{ set inclusion} \\ x \in \varphi :\leftrightarrow x \subseteq \varphi & // \text{ membership} \end{array}$$

Sorts as notations

A *sort* has a name and is associated with a set of its *inhabitants*. In matching logic, we use a symbol $s \in \Sigma$ to represent the sort name and use $(\mathsf{inh}\ s)$ to represent all its inhabitants, where $\mathit{inh} \in \Sigma$ is an ordinary symbol.

1. **New formulas (patterns)**:

$$\varphi ::= \top_s \mid \neg_s\varphi \mid \forall x{:}s.\,\varphi \mid \exists x{:}s.\,\varphi \mid \varphi{:}s \mid \forall x_1, \ldots, x_n{:}s.\,\varphi \mid \exists x_1, \ldots, x_n{:}s.\,\varphi$$

2. **New axioms**:

$$\begin{array}{rl} \top_s :\leftrightarrow \mathit{inh}\ s & // \text{ inhabitants of } s \\ \neg_s\varphi :\leftrightarrow (\neg\varphi) \wedge \top_s & // \text{ negation within sort } s \\ \forall x{:}s.\,\varphi :\leftrightarrow \forall x.\, x \in \top_s \rightarrow \varphi & // \ \forall \text{ within sort } s \\ \exists x{:}s.\,\varphi :\leftrightarrow \exists x.\, x \in \top_s \wedge \varphi & // \ \exists \text{ within sort } s \\ \varphi \cdot{:}\, s :\leftrightarrow \exists z{:}s.\,\varphi = z & // \ \varphi \text{ is an element of sort } s \\ \forall x_1, \ldots, x_n{:}s.\,\varphi :\leftrightarrow \forall x_1{:}s.\ \ldots \forall x_n{:}s.\,\varphi & // \text{ nested } \forall \text{ within sort } s \\ \exists x_1, \ldots, x_n{:}s.\,\varphi :\leftrightarrow \exists x_1{:}s.\ \ldots \exists x_n{:}s.\,\varphi & // \text{ nested } \exists \text{ within sort } s \end{array}$$

Many-sorted functional symbols as notations

A many-sorted function symbol $f : s_1 \times \cdots s_n \rightarrow s$ is represented as a notation as follows:

1. **New formulas (patterns)**:

$$\varphi ::= f(\varphi_1, \ldots, \varphi_n)$$

2. **New axioms**:

$$\begin{array}{rl} f(\varphi_1, \ldots, \varphi_n) :\leftrightarrow f\ \varphi_1\ \ldots\ \varphi_n & // \text{ functional notation} \\ \vdash \forall x_1{:}s_1.\ \ldots \forall x_n{:}s_n.\, \exists y{:}s.\, f(x_1, \ldots, x_n) = y & // \text{ function} \end{array}$$

Rewrite rules as notations

A binary relation $R \subseteq S \times S$ can be specified in $\mathbb{ML}$ as a symbol R together with an axiom $R\ S \subseteq S$. For specifying rewrite rules $\varphi_{lhs} \Rightarrow_{rew} \varphi_{rhs}$, we consider a sort *Cfg* for *configurations* and a *one-path next* symbol $\bullet$ representing the one-step transitions over configurations, i.e., $\bullet \mathit{Cfg} \subseteq \mathit{Cfg}$. Since $\varphi_{lhs} \Rightarrow_{rew} \varphi_{rhs}$ means that any configuration $\gamma \in \varphi_{lhs}$ has a next configuration γ' in φ_{rhs}, it can be specified by $\varphi_{lhs} \rightarrow \bullet\varphi_{rhs}$. Summarizing:

1. **New formulas (patterns)**:

$$\varphi ::= \varphi_{lhs} \Rightarrow_{rew} \varphi_{rhs}$$

2. **New axioms**:

$$\begin{array}{ll}
\varphi_{lhs} \Rightarrow_{rew} \varphi_{rhs} :\leftrightarrow \varphi_{lhs} \rightarrow \bullet\, \varphi_{rhs} & // \text{ rewrite rule} \\
\vdash \bullet\, \top_{Cfg} \subseteq \top_{Cfg} & // \text{ one-step transition} \\
\varphi_1 \rightarrow \bullet\, \varphi_2, \varphi_2 \rightarrow \bullet\, \varphi_3 \vdash \varphi_1 \rightarrow \bullet\, \bullet\, \varphi_3 & // \text{ transition transitivity}
\end{array}$$

Kore as notations

A $\mathbb{K}$ language definition (such as imp.k in Figure 2) is compiled into an intermediate representation, called the *Kore format*, by the $\mathbb{K}$ tool *kompile*. The Kore format [37] is based on many-sorted $\mathbb{ML}$ [6], which is itself just $\mathbb{ML}$ extended with a series of notations – i.e., not a new logic. The main syntactic categories of Kore include:[2]

Sorts:

Sort `::=` *SortVariable* `|` *SortId* `"{"` *Sorts* `"}"`
Sorts `::= "" |` *Sort* `{","` *Sort*`}`*
SortVariables `::= "" |` *SortVariable* `{","` *SortVariable*`}`*

Variables:

ElementVariable `::=` *ElementVariableId* `":"` *Sort*
SetVariable `::=` *SetVariableId* `":"` *Sort*

Patterns:

Pattern
`::=` *ElementVariable*
`|` *SetVariable*
`|` `"\bottom" "{"` *Sort* `"}" "(" ")"`
`|` `"\top" "{"` *Sort* `"}" "(" ")"`
`|` *Symbol* `"{"` *Sorts* `"}" "("` *Patterns* `")"`
`|` `"\not" "{"` *Sort* `"}" "("` *Pattern* `")"`
`|` `"\and" "{"` *Sort* `"}" "("` *Pattern* `","` *Pattern* `")"`
`|` `"\or" "{"` *Sort* `"}" "("` *Pattern* `","` *Pattern* `")"`
`|` `"\implies" "{"` *Sort* `"}" "("` *Pattern* `","` *Pattern* `")"`
`|` `"\iff" "{"` *Sort* `"}" "("` *Pattern* `","` *Pattern* `")"`
`|` `"\exists" "{"` *Sort* `"}" "("` *ElementVariable* `","` *Pattern* `")"`
`|` `"\forall" "{"` *Sort* `"}" "("` *ElementVariable* `","` *Pattern* `")"`
`|` `"\mu" "{" "}" "("` *SetVariable* `","` *Pattern* `")"`
`|` `"\nu" "{" "}" "("` *SetVariable* `","` *Pattern* `")"`
`|` `"\ceil" "{"` *Sort* `","` *Sort* `"}" "("` *Pattern* `")"`

[2] See [37] for a complete definition of Kore syntax.

```
    | "\floor" "{" Sort "," Sort "}" "(" Pattern ")"
    | "\equals" "{" Sort "," Sort "}" "(" Pattern "," Pattern ")"
    | "\in" "{" Sort "," Sort "}" "(" Pattern "," Pattern ")"
    | "\next" "{" Sort "}" "(" Pattern ")"
    | "\rewrites" "{" Sort "}" "(" Pattern "," Pattern ")"
    | "\dv" "{" Sort "}" "(" String ")"
Patterns ::= "" | Pattern {"," Pattern}*
```

The definition of Kore as an $\mathbb{ML}$ notation is given on top of theories defining sorts, many-sorted functions, and rewrite rules in a similar way to that given by the Metamath specification [38] (see also Section 3.5.1).

1. **New formulas (patterns)**:

$$
\begin{aligned}
\varphi ::= {} & \texttt{\textbackslash bottom}\{s\}() \\
| {} & \texttt{\textbackslash top}\{s\}() \\
| {} & \sigma\{s_1, \ldots, s_n\}(\varphi_1, \ldots, \varphi_n) \\
| {} & \texttt{\textbackslash not}\{s\}(\varphi) \\
| {} & \texttt{\textbackslash and}\{s\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash or}\{s\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash implies}\{s\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash iff}\{s\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash exists}\{s\}(x{:}s', \varphi) \\
| {} & \texttt{\textbackslash forall}\{s\}(x{:}s', \varphi) \\
| {} & \texttt{\textbackslash mu}\{\,\}(X{:}s, \varphi) \\
| {} & \texttt{\textbackslash nu}\{\,\}(X{:}s, \varphi) \\
| {} & \texttt{\textbackslash ceil}\{s_1, s_2\}(\varphi) \\
| {} & \texttt{\textbackslash floor}\{s_1, s_2\}(\varphi) \\
| {} & \texttt{\textbackslash equals}\{s_1, s_2\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash in}\{s_1, s_2\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash next}\{s\}(\varphi) \\
| {} & \texttt{\textbackslash rewrites}\{s\}(\varphi_1, \varphi_2) \\
| {} & \texttt{\textbackslash dv}\{s\}(\text{"}v\text{"})
\end{aligned}
$$

The above new patterns are sorted, where the sort is computed as follows:

$\texttt{\bottom}\{s\}() : s$
$\texttt{\top}\{s\}() : s$
$\sigma\{s_1, \ldots, s_n\}(s_1, \ldots, s_n) : \text{sort of } \sigma$
$\texttt{\not}\{s\}(s) : s$
$\texttt{\and}\{s\}(s, s) : s$
$\texttt{\or}\{s\}(s, s) : s$
$\texttt{\implies}\{s\}(s, s) : s$
$\texttt{\iff}\{s\}(s, s) : s$
$\texttt{\exists}\{s\}(\_, s) : s$
$\texttt{\forall}\{s\}(\_, s) : s$
$\texttt{\mu}\{\ \}(s, s) : s$
$\texttt{\nu}\{\ \}(s, s) : s$
$\texttt{\ceil}\{s_1, s_2\}(s_1) : s_2$
$\texttt{\floor}\{s_1, s_2\}(s_1) : s_2$
$\texttt{\equals}\{s_1, s_2\}(s_1, s_1) : s_2$
$\texttt{\in}\{s_1, s_2\}(s_1, s_1) : s_2$
$\texttt{\next}\{s\}(s) : s$
$\texttt{\rewrites}\{s\}(s, s) : s$
$\texttt{\dv}\{s\}(\text{"}v\text{"}) : s$

2. **New axioms**:

$$
\begin{aligned}
\texttt{\bottom}\{s\}() &:\leftrightarrow \bot \\
\texttt{\top}\{s\}() &:\leftrightarrow \top_s \\
\sigma\{s_1, \ldots, s_n\}(\varphi_1, \ldots, \varphi_n) &:\leftrightarrow \sigma(\varphi_1, \ldots, \varphi_n) \wedge \varphi_1 \mathrel{:}: s_1 \wedge \cdots \wedge \varphi_n \mathrel{:}: s_n \\
\texttt{\not}\{s\}(\varphi) &:\leftrightarrow \neg_s \varphi \\
\texttt{\and}\{s\}(\varphi_1, \varphi_2) &:\leftrightarrow \varphi_1 \wedge \varphi_2 \\
\texttt{\or}\{s\}(\varphi_1, \varphi_2) &:\leftrightarrow \varphi_1 \vee \varphi_2 \\
\texttt{\implies}\{s\}(\varphi_1, \varphi_2) &:\leftrightarrow \texttt{\or}\{s\}(\texttt{\not}\{s\}(\varphi_1), \varphi_2) \\
\texttt{\iff}\{s\}(\varphi_1, \varphi_2) &:\leftrightarrow \texttt{\and}\{s\}(\texttt{\implies}\{s\}(\varphi_1, \varphi_2), \\
&\qquad\qquad \texttt{\implies}\{s\}(\varphi_2, \varphi_1)) \\
\texttt{\exists}\{s\}(x \mathrel{:} s', \varphi) &:\leftrightarrow \exists x \mathrel{:} s'.\, \varphi \wedge \top_s \\
\texttt{\forall}\{s\}(x \mathrel{:} s', \varphi) &:\leftrightarrow \texttt{\not}\{s\}(\texttt{\exists}\{s\}(x \mathrel{:} s', \texttt{\not}\{s\}(\varphi))) \\
\texttt{\mu}\{\ \}(X \mathrel{:} s, \varphi) &:\leftrightarrow (\mu X.\, \varphi) \wedge \top_s \\
\texttt{\nu}\{\ \}(X \mathrel{:} s, \varphi) &:\leftrightarrow \texttt{\not}\{s\}(\texttt{\mu}\{\ \}(X \mathrel{:} s, \texttt{\not}\{s\}(\varphi))) \\
\texttt{\ceil}\{s_1, s_2\}(\varphi) &:\leftrightarrow \lceil \varphi \rceil \wedge \top_{s_2} \\
\texttt{\floor}\{s_1, s_2\}(\varphi) &:\leftrightarrow \texttt{\not}\{s_2\}(\texttt{\ceil}\{s_1, s_2\}(\texttt{\not}\{s_1\}(\varphi))) \\
\texttt{\equals}\{s_1, s_2\}(\varphi_1, \varphi_2) &:\leftrightarrow \texttt{\floor}\{s_1, s_2\}(\texttt{\iff}\{s_1\}(\varphi_1, \varphi_2)) \\
\texttt{\in}\{s_1, s_2\}(\varphi_1, \varphi_2) &:\leftrightarrow \texttt{\floor}\{s_1, s_2\}(\texttt{\implies}\{s_1\}(\varphi_1, \varphi_2)) \\
\texttt{\next}\{s\}(\varphi) &:\leftrightarrow \bullet \varphi \\
\texttt{\rewrites}\{s\}(\varphi_1, \varphi_2) &:\leftrightarrow \texttt{\implies}\{s\}(\varphi_1, \texttt{\next}\{s\}(\varphi_2)) \\
&\vdash \backslash\text{dv}\{s\}(\text{"}v\text{"}) \mathrel{:}: s \\
&\vdash (\varphi \subseteq \top_s) \to (\texttt{\next}\{s\}(\varphi) \subseteq \top_s)
\end{aligned}
$$

2.2.2 Proof System

With the basic matching logic syntax and its derived notations defined above in place, we also need a proof system that defines the provability relation $\vdash$ between theories and formulas. This is required so that we can write $\Gamma \vdash \varphi$, which represents φ can be proved using the proof system, with patterns in Γ added as additional axioms.

Figure 3 shows the Hilbert-like proof system used for matching logic [6, 5]. The proof rules are sound and can be divided into four categories: FOL reasoning, frame reasoning, fixpoint reasoning, and some technical rules. C, C_1, and C_2 denote patterns that have a single placeholder variable $\Box$ that appears only within nested symbol applications (and not logical connectives). The notation $C[\varphi]$ is equivalent to $C[\varphi/\Box]$. The FOL reasoning rules provide (complete) FOL reasoning (see, e.g., [36]). 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 [22]. The technical proof rules are needed for some completeness results (see [6] for details). Since matching logic is the logical foundation of $\mathbb{K}$, the correctness of $\mathbb{K}$ conducting one language task is reduced to the existence of a formal proof in matching logic, using the proof system in Figure 3.

FOL Reasoning	(Tautology)	φ if φ is a propositional tautology over patterns
	(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
	(∃-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
	(∃-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x.\varphi_1) \to \varphi_2}$ if $x \notin \mathrm{FV}(\varphi_2)$
Technical Rules	(Existence)	$\exists x.\, x$
	(Singleton)	$\neg\left(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi]\right)$
Reasoning Frame	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]$
	(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ if $x \notin \mathrm{FV}(C)$
	(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}$
Reasoning Fixpoint	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Pre-Fixpoint)	$\varphi[\mu X.\, \varphi/X] \to \mu X.\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\, \varphi \to \psi}$

Figure 3: Hilbert proof System

2.3 Zero-Knowledge Basics

Highly related to our work are the concepts of *Zero-Knowledge Cryptography* and, more specifically, *Zero-Knowledge Virtual Machine*, or zkVM.

A generic zero-knowledge proof system designed to verify a particular computation typically requires that the computation be specified using some *arithmetization*. An arithmetization is a way to represent the computation as a system of equations, typically over a finite field or over a ring of polynomials over a finite field. This representation depends on the proof system being used and the cryptographic constructions on which the proof system is based.

A zkVM, by contrast, is a zero-knowledge proof system that verifies computations described by a program that runs on a virtual machine. Rather than requiring the computation to be arithmetized to start with, a zkVM system handles the conversion of the program to arithmetic form, allowing programs written in high-level languages, even programs not initially designed with verifiable computing in mind, to be used in zero-knowledge proof systems.

The zkVM abstraction is relevant to our work for two reasons. First, we are dealing with formal systems which, up to now, have not been designed with zero-knowledge in mind and which have been implemented in high-level programming languages. It is, therefore, an obvious possibility that existing zkVM systems could be used to construct our Proof of Proof system. Indeed, as shown in Section 4, we have carried out implementations of our Proof of Proof system within a number of zkVM systems – that section describes our experiments and results.

Second, because our Proof of Proof system is designed to handle verifiable computation for arbitrary programming languages, any implementation of the system is bound to share features in common with zkVM systems. In Section 5, we present our research into the design of a Proof of Proof system that is not based on existing zkVMs but rather is designed from first principles to best leverage zero-knowledge techniques for maximal efficiency.

3 Math Proof Generation

As the name suggests, the Math Proof Generation (MPG) process generates mathematical proofs from the execution steps of programs. From the Curry-Howard correspondence [16], we know that there is a direct relationship between computer programs and mathematical proofs, and thus we can convert from programs to proofs and vice versa. MPG refines this correspondence and pushes it one step further, automatically generating machine-checkable mathematical proofs from *program executions*. That is, the execution of any program in any programming language that has a formal semantics in $\mathbb{K}$ is *automatically* substantiated with a formal mathematical proof. Moreover, that formal proof is verifiable independently of $\mathbb{K}$ or other frameworks, implementations, or systems.

The general flow of how a mathematical proof, $Proof$, is generated from a program, $Program$, of a given language, $Lang$, can be seen in Figure 1 and it

goes like this:

1. Given the semantics of *Lang* written in $\mathbb{K}$ framework, it will be compiled into its Kore format as the *Lang* Math Theory.

2. $\mathbb{K}$'s LLVM instrumented execution backend will take in a) the *Lang* Math Theory and b) the *Prog*, together with c) the other execution environment inputs to generate proof hints. Proof hints are execution traces that the program has taken based on the defined semantics *Lang* (more on proof hints later).

3. The MPG process will take in a) the proof hints generated from the *Prog* and b) the *Lang* Math Theory, together with c) additional rules from some pre-defined proof calculus, to generate an internal representation of a Math Proof.

4. This internal representation of the Math Proof is then serialized to a proof checker format such as Metamath or a specialized block model, which is machine-verifiable either directly via the designated proof checker or via a zkVM.

To provide more in-depth details of the points mentioned above, this section is broken into the following parts:

- Section 3.1 provides basic knowledge of how language definitions are defined as Matching Logic ($\mathbb{ML}$) theories. As the underlying logic behind $\mathbb{K}$ is $\mathbb{ML}$, it is important to understand how the language definitions defined in $\mathbb{K}$ can be viewed as $\mathbb{ML}$ theories. This in turn can help us understand how a program can be broken down into execution steps, proof hints, and lastly, generated to a mathematical proof in later subsections.

- Section 3.2 mentions the additional relations, predicates, and rules needed for our MPG process. Other than the matching logic proof system mentioned in Section 2.2.2, these new relations, predicates, and rules better facilitate the generation of mathematical proofs in the MPG process.

- Section 3.3 shares more details on the types of proof hints and how they are generated before passing them to the MPG process. These proof hints are essential in generating the mathematical proof of a given program as these are steps taken by the program that goes from an initial state to a final state. These hints will in turn guide us on how we should build the proof for the program.

- Section 3.4 details how the MPG process reads proof hints one at a time, processes every type of them, and eventually generates the proof of the program. This is the crux of the MPG, as this process generates the machine-verifiable proof of the correctness of the program.

- Section 3.5 enumerates the two main proof checker formats that the MPG can serialize to. Note that the MPG process will first generate an internal representation of a mathematical proof, which needs to be generic and reusable in the sense that it should be versatile to be serialized to any proof checker formats that we chose.

3.1 Language Definitions as Matching Logic ($\mathbb{ML}$) Theories

A programming language definition L specifies, via *notation*, a $\mathbb{ML}$ theory Γ^L consisting of:

1. A theory Γ^T for each builtin datatype T
2. A theory $\Gamma^{Syn(L)}$ specifying the syntax of L
3. A theory $\Gamma^{Sem(L)}$ specifying the semantics of L
4. A theory $\Gamma^{Simpl(L)}$ specifying the simplification rules of L

We exemplify these theories for the imperative language defined in Fig. 2.

Builtin theories Γ^T

The theories specifying the imported builtin datatypes T include:

1. A theory Γ^{INT} specifying the integers. This consists of a sort Int, functional symbols $+_{Int}$ and $-_{Int}$, a predicate symbol $=/=_{Int}$, and all axioms of the form $+_{Int}(1,2) = 3$ and $=/=_{Int}(1,0)$.
2. A theory Γ^{MAP} of maps. This includes the empty map $._{Map}$, the map element constructor $\_ \mapsto \_$, the associative&commutative concatenation constructor $\_\ \_$, and the operations *lookup* and *update*.
3. A theory Γ^{ID} of identifiers.
4. A theory Γ^{IDS} of identifier lists, with the empty lists constructor $._{Ids}$, and the associative concatenation $\_, \_$.
5. A theory Γ^K, which specifies a program's computational units and the order in which they are computed.

A complete specification of these theories is impractical. Therefore they are only partially specified and can be (conservatively) extended at runtime with trusted evaluations given from outside. We will denote by $\mathcal{B}$ the theory that extends the theories Γ^T, and will explain its construction in Section 3.4.2.

The theory $\Gamma^{Syn(L)}$

Each non-terminal, e.g., *Exp*, specifies a sort, and each syntax rule defines a symbol together with the axioms specifying its functional type and the fact that is a constructor. For instance, the syntax rule `Exp ::= Exp "+" Exp` specifies a symbol, say $Plus$, together with the following axioms:

$$\forall x : Exp.\forall y : Exp.\exists z : Exp.Plus(x, y) = z \tag{3}$$

$$\forall x_1, x_2 : Exp.\forall y_1, y_2 : Exp.Plus(x_1, y_1) = Plus(x_2, y_2) \rightarrow x_1 = x_2 \wedge y_1 = y_2 \tag{4}$$

$$\forall x_1, x_2 : Exp.\forall y_1, y_2 : Exp.\neg(Plus(x_1, y_1) = Minus(x_2, y_2)) \tag{5}$$

The carrier set of a non-terminal sort is inductively defined, e.g.,

$$Exp = \mu X.Int \vee Id \vee Plus(X, X) \vee Minus(X, X)$$

The theory $\Gamma^{Sem(L)}$

Specifies the transition steps that give the operational semantics of a program. For example, the semantic rule that evaluates a program variable

```
rule <k> X:Id => I ...</k>
     <state>... X |-> I ...</state>
```

is a notation for the configuration rewrite rule

```
<T>                                        <T>
  <k> X ~> VarK </k>                         <k> I ~> VarK </k>
  <state>... X |-> I ...</state>     ⇒       <state>... X |-> I ...</state>
</T>                                       </T>
```

and specified as an $\mathbb{ML}$ axiom

$$\begin{array}{l} TCell(kCell(kseq(X, VarK)), stateCell(M)) \\ \rightarrow \\ \bullet\, TCell(kCell(kseq(lookup(M, X), VarK)), stateCell(M)) \end{array}$$

We can also see that the conditional semantic rule

```
rule if (I) S _ => S requires I =/=Int 0
```

is specified by an $\mathbb{ML}$ axiom as follows:

$$\begin{array}{l} =/=_{Int}\ (I, 0) \wedge TCell(kCell(kseq(If(I, S, S_2), VarK)), stateCell(M)) \\ \rightarrow \\ \bullet\, TCell(kCell(kseq(S, VarK)), stateCell(M)) \end{array}$$

We conventionally write a conditional semantic rule as

$$(\phi \wedge \ell) \Rightarrow_{semrew} r$$

where ϕ is the predicate pattern specifying the condition, ℓ is the configuration pattern specifying the left-hand side of the rewrite rule, and r is the configuration pattern specifying the right-hand side of the rewrite rule. It is just a notation for the $\mathbb{ML}$ pattern is $(\phi \wedge \ell) \rightarrow \bullet r$. If ϕ is $\top$, then rule is written as $\ell \Rightarrow_{semrew} r$.

The theory $\Gamma^{Simpl(L)}$

Specifies how the builtin or user-defined functions are computed. For example, the rule defining an update of a map

```
rule (K |-> _ M:Map) [ K <- V ] => (K |-> V M) [simplification]
```

is specified by an $\mathbb{ML}$ axiom of the form

$$\texttt{Map:update}(\_\ \_(\_\mapsto\_(K,X),M),V) = (\_\ \_(\_\mapsto\_(K,V),M))$$

A conditional simplification rule is written as

$$(\phi \wedge \ell) =_{simpl} r$$

and it can be seen as a notation for $(\phi \wedge \ell) = r$. If ϕ is $\top$, then rule is written as $\ell =_{simpl} r$.

3.1.1 Language Definitions as Kore Theories

The current version of $\mathbb{K}$ translates a programming language definition into a Kore theory (see Section 2.2.1), using the following syntax:

Sentence
  ::= "sort" *SortId* "" *SortVariables* "" "[" *Attributes* "]"
    | "symbol" *Symbol* "{" *SortVariables* "}" "(" *Sorts* ")" ":" *Sort* "[" *Attributes* "]"
    | "axiom" "{" *SortVariables* "}" *Pattern* "[" *Attributes* "]"
    | "claim" "{" *SortVariables* "}" *Pattern* "[" *Attributes* "]"

For example, for the syntax rule `Exp ::= Exp "+" Exp`, the next theory fragment is generated:

```
symbol Plus {}(SortAExp{}, SortAExp{}) : SortAExp{}
                    [constructor{}(), functional{}()]

axiom{R} \exists{R}(
     Val:SortAExp{},
     \equals{SortAExp{}, R} (Val:SortAExp{},
                             Plus{}(K0:SortAExp{}, K1:SortAExp{})
                            )
) [functional{}()] // functional

axiom{} \implies{SortAExp{}}(
    \and{SortAExp{}}(
        Plus{}(X0:SortAExp{}, X1:SortAExp{}),
        Plus{}(Y0:SortAExp{}, Y1:SortAExp{})
    ),
    Plus{}(
        \and{SortAExp{}} (X0:SortAExp{}, Y0:SortAExp{}),
```

```
        \and{SortAExp{}} (X1:SortAExp{}, Y1:SortAExp{})
    )
) [constructor{}()] // no confusion same constructor

axiom{} \not{SortAExp{}} (
    \and{SortAExp{}} (
        Plus{}(X0:SortAExp{}, X1:SortAExp{}),
        Minus{}(Y0:SortAExp{}, Y1:SortAExp{})
    )
) [constructor{}()] // no confusion different constructors
```

The three axioms are equivalent to (3)-(5) on page 18.

The rule

```
rule <k> X = I:Int; => .K ...</k>
     <state>... X |-> (_ => I) ...</state>
```

is translated into a Kore axiom as follows:

```
axiom{} \rewrites{TCell} (
    \and{TCell} (
        <T>(
          <k>(kseq(assign(VarX:Id, VarI:Int), DVar2:K)),
          <state>(concMap(mapItem(VarX:Id, Gen0:KItem), DotVar3:Map))
        ),
        \top{TCell}
    ),
    \and{TCell} (
        <T>(
          <k>(DotVar2:K),
          <state>(concMap(mapItem(VarX:Id, VarI:Int), DotVar3:Map))
        ),
        \top{TCell}
    )
)
```

where `<T>` is the topmost configuration cell.

3.1.2 Running Programs using the Language Definition

An execution $t_0 \Rightarrow^1 t_1 \Rightarrow^1 \cdots \Rightarrow^1 t_n$ is obtained using the theory Γ^L defining L. An execution step $t_i \Rightarrow^1 t_{i+1}$ is broken down into two substeps:

1. $t_i \Rightarrow_{sem} tt_{i+1}$ consisting of the application of a semantic (rewrite) rule $\phi \wedge \ell \Rightarrow_{semrew} r$;
2. $tt_{i+1} =^!_{simpl} t_{i+1}$ consisting of the application of the simplification rules such as function evaluation rules as much as possible (marked by $=^!_{simpl}$).

Intuitively, the fact that $t_i \Rightarrow_{sem} tt_{i+1}$ is obtained using the semantic rule $\ell \wedge \phi \Rightarrow_{semrew} r$ means that

1. t_i matches the left-hand side ℓ of the rule via a substitution θ, i.e., $t_i = \ell\theta$
2. The substitution θ satisfies the condition ϕ, i.e., $\Gamma^L \vdash \phi\theta$
3. tt_{i+1} is obtained by applying θ to r, i.e., $tt_{i+1} = r\theta$.

A rewrite rule can be applied only when the current configuration is in a normal form. The theory Γ^L may include *simplification rules* $\phi \wedge \ell =_{simpl} r$ that transform a pattern into an equivalent one. A configuration is in a *normal form* if no simplifications can be applied. A simplification step $tt_{i+1} =^{!}_{simpl} t_{i+1}$ computes the normal form t_{i+1} of tt_{i+1}.

We write $t \Rightarrow t'$ if there is an execution from t to t':

$$\frac{t \Rightarrow^1 t'}{t \Rightarrow t'} \qquad \frac{t \Rightarrow^1 t' \quad t' \Rightarrow t''}{t \Rightarrow t''}$$

3.2 Proof Calculus

The proof calculus consists of the matching logic proof system, well-formedness (type-checking) rules, variable substitutions, rule instantiation and derived rules. The reason for extending the proof system by incorporating these additional relations, predicates, and derived rules is that they aim to improve coverage and streamline the proof generation process. As a result, it enables a more comprehensive and efficient reasoning framework.

3.2.1 Relations/Predicates

Well-formedness

$\mathit{WfTerm}(t, s)$ denotes the fact that t is a well-formed term pattern of sorts s.

$\mathit{WfSubst}(\theta)$ denotes the fact that the substitution θ is well-formed, i.e., for each $x \mapsto u \in \theta$, $\mathit{WfTerm}(u, s)$ and $\mathit{WfTerm}(x, s)$ for certain sort s.

$\mathit{WfPred}(\phi)$ denotes the fact that ϕ is a well-formed predicate pattern.

Element variables substitution

A substitution $\theta = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k\}$ can be seen as a notation for the $\mathbb{ML}$ pattern $\theta^{=} \equiv x_1 = u_1 \wedge \cdots \wedge x_k = u_k$, where the element variables x_i and the term patterns u_i are of the same sort. The result $\varphi\theta$ of applying θ to a pattern φ can be seen as a notation for $\varphi \wedge \theta^{=}$, assuming that variables x_i are fresh. We also write $\varphi[\overline{u}/\overline{x}]$ for $\varphi\theta$.

Set variable substitution

If X is a set variable and φ, ψ are patterns, then $\varphi[\psi/X]$ denotes the capture-avoid substitution, obtained by replacing the free occurrences of X in φ by ψ.

Rule instantiation

The (derived) inference rules are in fact rule schemes, i.e., they are written using meta-variables. For instance, in the MODUS PONENS rule

$$\frac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi 2}$$

φ_1 and φ_2 are meta-variables. An *instantiation* of the rule is obtained by replacing the meta-variables with well-formed patterns.

3.2.2 Derived Rules

As our proof generation process is heavily influenced by the proof hints generated from the $\mathbb{K}$'s LLVM backend, having specialized derived rules is beneficial in facilitating the process of generating the mathematical proofs. Each of the derived rules stated in this subsection will be applied in the proof generation process depending on the type of proof hint received at a given point in time. Further elaboration on which derived rules should be applied for which type of proof hint will be discussed in Section 3.4.2.

Application of a semantic rewrite rule

$$\frac{(\phi \wedge \ell \rightarrow \ \bullet\, r) \in \Gamma^{Sem(L)} \quad \phi[\overline{u}/\overline{x}] \quad t = \ell[\overline{u}/\overline{x}] \quad t' = r[\overline{u}/\overline{x}]}{t \rightarrow \ \bullet\, t'} \qquad \text{(RewRl)}$$

Application of a simplification rewrite rule

$$\frac{(\phi \wedge \ell = r) \in \Gamma^{Simpl(L)} \quad \phi[\overline{u}/\overline{x}] \quad t = \ell[\overline{u}/\overline{x}] \quad t' = r[\overline{u}/\overline{x}]}{t = t'} \qquad \text{(SimplRl)}$$

Congruence

$$\frac{u_1 = v_1 \quad \ldots \quad u_n = v_n \quad t' = t[\overline{u}/\overline{x}] \quad t'' = t[\overline{v}/\overline{x}]}{t' = t''} \qquad \text{(Congr)}$$

Equality transitivity

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3} \qquad \text{(EqTrans)}$$

Transition relation step

$$\frac{t \rightarrow \ \bullet\, t' \quad t' = t''}{t \rightarrow \Diamond t''} \qquad \text{(TrRelStep)}$$

where $\Diamond t'' \equiv \mu X.t'' \vee \bullet X$ [6].

Transition relation transitivity

$$\frac{t_1 \rightarrow \Diamond t_2 \quad t_2 \rightarrow \Diamond t_3}{t_1 \rightarrow \Diamond t_3} \qquad \text{(TrRelTrans)}$$

Semantic rewrite rules are well-formed

$$\frac{(\phi \wedge \ell \rightarrow \ \bullet\, r) \in \Gamma^{Sem(L)}}{\mathit{WfPred}(\phi) \quad \mathit{WfTerm}(\ell, s) \quad \mathit{WfTerm}(r, s)} \qquad \text{(WfRewRl)}$$

where s is the sort for program configurations.

Simplification rewrite rules are well-formed

$$\frac{(\phi \wedge \ell = r) \in \Gamma^{Simpl(L)}}{\mathit{WfPred}(\phi) \quad \mathit{WfTerm}(\ell, s) \quad \mathit{WfTerm}(r, s)} \qquad \text{(WfSimplRl)}$$

for certain sort s.

Well-formedness/Type preservation

$$\frac{\mathit{WfPred}(\phi) \quad \mathit{WfSubst}(\overline{x} \mapsto \overline{u})}{\mathit{WfPred}(\phi[\overline{u}/\overline{x}])} \qquad \frac{\mathit{WfTerm}(t, s) \quad \mathit{WfSubst}(\overline{x} \mapsto \overline{u})}{\mathit{WfTerm}(t[\overline{u}/\overline{x}], s)} \qquad \text{(WfPres)}$$

3.3 Proof Hints

The LLVM backend of the $\mathbb{K}$ Framework has been enhanced with the capability to instrument the generated code to produce proof hints. The role of these proof hints is to allow communication between the interpreter and the proof generation engine, providing information such as the specific axioms that should be applied, their order, their respective substitutions, and the part of the final proof that they contribute to. The purpose of hints is, thus, twofold. First, their sequence corresponds to an execution trace of the program being interpreted. Second, and most crucially, they enrich this execution trace with the mathematical information necessary to guide proof generation. These proof hints are categorized into different types, depending on the events occurring at the point of execution and follow a grammar defined in the BNF style that can be found in [32]. The section below describes the types of proof hints.

3.3.1 Types of Proof Hints

Each kind of hint event contains specific information describing the execution of a proof-related piece of code. A common feature between different hint types is the presence of a *rule ordinal*, which is a number corresponding to the nth axiom in a Kore definition (see Section 3.1.1). These numbers are assigned during the axiom preprocessing phase by the LLVM Backend. The purpose of ordinals is to refer to an axiom in the Kore definition (which corresponds to a rule in theory Γ^L), from within the proof hints which emit them.

Currently, the backend can emit the following 8 types of proof hints during a program execution:

- **Function event:** This hint event is produced as soon as the interpreter starts the evaluation of a function. It contains the name and arguments of the function being evaluated and its relative position on the stack of evaluations. The arguments of the functions are Kore terms themselves.
- **Function exit event:** This hint event is produced when the interpreter finishes the evaluation of a function. It contains the ordinal of the rule used to simplify the last-open function context, and whether the function exited via a tail call or a conventional return statement. This event is mainly useful for computing the call stack of the various simplifications in a proof hint trace.
- **Rule event:** This hint event is produced when the interpreter starts to evaluate a rewrite rule and its arguments. It provides the ordinal of the rule being applied along with a substitution to instantiate it, $\theta = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k\}$.
- **Pattern matching failure event:** This hint event is produced when no axiom matches the subterm referred to by the most recent function event. That is, it tells the MPG process (Section 3.4) that no rule should be applied to simplify a function further. As $\mathbb{K}$ implements subsort overloading (see [18]), these events are often emitted when dealing with overloaded constructors, which are constructors-modulo-axioms. As information, they provide the name of the function that could not match any rule.
- **Side condition entry event:** This hint event is produced when the interpreter starts evaluating the side condition of a rule. It provides the rule ordinal and the substitution $\theta = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k\}$ to be applied to the side condition term.
- **Side condition exit event:** This hint event is emitted once side condition evaluation finishes. It provides the rule ordinal, as well as the final result of evaluation (e.g., "true").
- **Hook event:** This hint event is emitted when a hook function occurring at a certain relative position is called. These are special kinds of built-in functions for which the simplification rules in Γ^T (see Section 3.1) are bypassed (or not defined at all) without a high-level evaluation. Instead, their evaluation is done by directly computing the result in the machine code level, which is also transmitted at the end of the hook event output on the trace. This is useful for, e.g., arithmetic computation, where evaluation can potentially lead to a large number of events being emitted.
- **Configuration pattern event:** This event contains the Kore representation of a $\mathbb{K}$ configuration. Usually, it happens as an initial or final configuration, but it can also appear as an intermediate configuration, between rewrites, if the user sets the corresponding flag in the program execution command.

3.3.2 Proof Hints Generation

The generation of proof hints is achieved through instrumentation of the generated code by the LLVM backend, with additional instructions that are responsible for outputting a proof hint event. Each language definition compiled by the $\mathbb{K}$ Framework, using the LLVM backend with the appropriate flags, will have this instrumented code that will output the proof trace of executions.

The code generator of the LLVM backend is responsible for generating code that implements pattern matching as directed by a Maranget-like decision tree as shown in [28, 17], as well as the rewriting that should happen when a leaf node is reached in the tree, which corresponds to a rewrite rule. The main loop of execution implements the idea from Section 3.1.2 as follows:

1. Given a Kore term, walk the decision tree to reach a leaf node.
2. Apply the rewrite rule that corresponds to the reached node to the Kore term.
3. Repeat for the new Kore term we get after applying the rewrite rule.

The execution terminates if, in Step 1, we fail to find a match, i.e., we end up in a special node of the decision tree that represents a pattern matching failure. Otherwise, the execution will eventually terminate if all Kore terms have been correctly evaluated.

The additional instrumented code is generated between Steps 1 and 2, and it is guarded by some conditions that can be controlled by flags passed to the language semantics's binary interpreter. So, the interpreter can be used to only execute a program, execute and output the proof trace, or even execute and output the proof trace with intermediate configuration events that can be used for debugging purposes. The two last modes are set when invoking the interpreter with the appropriate flags.

3.4 Proof Generation

3.4.1 Main Idea

The MPG problem can be stated as follows:

Input: A language definition L and a claim of the form $t \Rightarrow^* t'$, where t and t' are two program configurations in L.

Output: A proof of the fact that there exists an execution $t_0 \Rightarrow^1 t_1 \Rightarrow^1 \cdots \Rightarrow^1 t_n$ in L with $t = t_0$ and $t_n = t'$, if any.

In terms of $\mathbb{ML}$, the above problem is stated as follows:

Input: An $\mathbb{ML}$ theory Γ^L specifying L, and a claim represented as an $\mathbb{ML}$ pattern $t \to \Diamond t'$ with t, t' terms of sort Cfg (= the sort for configurations).

Output: A proof of $\Gamma^L \vdash t \to \Diamond t'$, if any.

A proof for $\Gamma^L \vdash t \to \Diamond t'$ can be obtained using the execution $t_0 \Rightarrow^1 t_1 \Rightarrow^1 \cdots \Rightarrow^1 t_n$, where $t = t_0$ and $t_n = t'$. The general idea of how the proof can be generated is as follows:

Sketch of MPG algorithm:

1. For each step $t_i \Rightarrow^1 t_{i+1}$ a proof for $\Gamma^L \vdash t_i \to \Diamond t_{i+1}$ is built. Recall that such a step is obtained by applying a rule $\phi \wedge \ell \Rightarrow_{semrew} r$. The main idea of the proof generation process is as follows:
   (a) $t_i \Rightarrow_{sem} tt_{i+1}$ is just a notation for $t_i \to \bullet tt_{i+1}$ and its proof consists of
      i. generating proof for the side condition, if any;
      ii. generating proof for the equality between the current configuration t_i and the instance of the left-hand side of the rule;
      iii. tt_{i+1} is the instance of the right-hand side of the rule;
      iv. instantiating (RewRl);
   (b) the proof for the substep $tt_{i+1} =^!_{simpl} t_{i+1}$ consists of:
      i. generating proof for each simplification rule applied;
      ii. instantiating (EqTrans), whenever it is needed;
      iii. instantiating (Congr), whenever it is needed.
   (c) apply (TrRelStep).
2. Using the following instantiations of the transitivity rule (TrRelTrans)

$$\frac{t_0 \to \Diamond t_i \quad t_i \to \Diamond t_{i+1}}{t_0 \to \Diamond t_{i+1}}$$

   for $i = 1, \ldots, n-1$, we obtain a proof for $t_0 \to \Diamond t_n$.

3.4.2 Proof Generation Guided by Proof Hints

While the foregoing serves as a general outline, the specific manner in which proof generation proceeds will be dependent on the program being executed. As mentioned in Section 3.3, proof hints provide critical information, including the axioms to be used, the substitutions to instantiate them by, the subterms that they rewrite, and the order to apply them. This information is necessary as it helps to guide the proof generation process.

Builtin theory $\mathcal{B}$

As mentioned in Section 3.1, the builtin theories Γ^T are incomplete. Here we describe how they are conservatively extended with a dynamically built theory $\mathcal{B}$. Consequently, the generated proof is modulo builtin theory $\mathcal{B}$, which includes all the claims describing computations given by the builtin functions/operations. This theory is trusted or it is checked by an external tool. This theory is built using the *Hook events*. Such a hint includes the name of the

builtin function/operation, its arguments $\overline{a}$ (if any), and the result v. Then a claim of the form $f(\overline{a}) = v$ is included in $\mathcal{B}$. A simple example is $+_{Int}(1, 2) = 3$.

Simplification/Evaluation strategy

The evaluation of the side conditions and the simplification substeps are based on the same operations, application of simplification rules, and/or hook functions. Therefore the proof generation is based on a sequence of hints consisting of *Function events*, *Rule events*, and *Hook events*. *Pattern Matching Failure events* also belong to this category, but they do not mutate the evaluated term in any way, as their meaning is that no simplification could be applied (see Section 3.3). Here we describe a strategy for generating a proof chunk for such a sequence. It is the most challenging part since some hint events could be nested. Therefore, the sequence of hints is organized into a hierarchy of *regions*, such that a proof chunk corresponds to each region. After the proof chunks of a region are generated, these are aggregated:

- Using instantiations of the equality transitivity rule (EqTrans), to contract chains of equalities.
- Using instantiations of the congruence rule (Congr), to propagate the equalities to the parent region. The idea is that the hierarchy of regions reflects the subterm structure, where the simplifications hold.

This building process of the region hierarchy is guided by the *Function Exit events* and the relative positions. The final proof chunk of the hint sequence is that corresponding to the top region. An advantage of this strategy is that the regions can be generated in parallel.

Now we describe how the hints are used to generate the proof chunks for the main steps of the algorithm sketched in Section 3.4.1.

Item 1(a)i The proof chunk for the evaluation of a side condition $\phi[\overline{u}/\overline{x}]$ is built using the hints emitted during a side condition evaluation, which are marked by a *Side Condition entry event* and its corresponding *Side Condition Exit event*. The substitution $\overline{u}/\overline{x}$ is given by the corresponding hint events. The proof chunk is built using the *Simplification/Evaluation strategy*.

Item 1(a)ii The proof chunk can be built by a simultaneous traversal of the configurations and applying in a bottom-up manner the congruence rule (Congr). Note that this chunk must be constructed modulo the associative-commutative axioms of certain constructors, which adds additional complexity.

Item 1b The proof chunk is built using the *Simplification/Evaluation strategy*.

Item 1c This step aggregates the proof chunks generated at Item 1a and Item 1b.

Item 2 The region approach is extended to be applied to the entire proof of the execution. The hints for each step $t_i \Rightarrow^1 t_{i+1}$ form a subregion of the top region, and the transitivity aggregates the proof chunks of these regions.

3.5 Proof Checker

The MPG process will generate an internal representation of a mathematical proof of a given program before being serialized to a proof format that is machine verifiable. Even though the design of how the internal representation of a mathematical proof is still in progress, there are a few key properties that hold:

1. **Generic and easy to serialize:** The internal representation of any mathematical proof should be generic in the sense that it can be easily serialized to any proof format that we wish to serialize to, e.g., Metamath, Coq, Lean, etc.

2. **Modular and reusable:** The internal representation of components of a proof such as patterns, axioms, (sub-)proofs and theories, should be modular so that they can be easily reused (without reproving) while building the overall proof for a given program.

With these key properties, the internal representation of any mathematical proof of a given program can be easily serialized to a target proof format which allows fast and efficient verification. For the remaining of this subsection, we will discuss the two main proof checker formats that we aim to serialize to: a) Metamath and b) proposed block model. We will give a brief description of how matching logic, Kore notations and the proof generated can be represented in these two proof checker languages.

3.5.1 Metamath

Metamath [29] is a simple language for representing formal proofs. Originally developed by Norman Megill in 1990, it was designed to be a language capable of representing proofs in user-specified formal systems. The largest repository of Metamath proofs is `set.mm`, which contains theorems in ZFC set theory and has grown to over 23000 theorems.

A key feature of Metamath is its simple design. Metamath uses only 15 keywords, each denoted by an initial `$` character, and does not allow for extensibility of this basic syntax. This makes it straightforward to implement checkers for Metamath: Over 22 implementations exist in a variety of languages, most running to only a few hundred lines of code [3]. This simplicity makes it well-suited to the purpose of use in a ZK proof system, as it becomes possible to adapt and compile existing implementations to zkVMs, or replicate the semantics of the language in an arithmetic constraint system.

To demonstrate the simplicity of Metamath, the rest of this subsection provides a brief illustration of how matching logic can be formalized in Metamath. We will see how we can formalize the syntax, formulas/patterns, axioms and proofs that were defined in Section 2.2 in Metamath.

At high level, any Metamath source file will contain a list of *statements* and the main ones are:

[3] `https://us.metamath.org/other.html#verifiers`

- *Constant statements* (`$c`) that declare Metamath constants.
- *Variable statements* (`$v`) that declare Metamath variables, and *Floating statements* (`$f`) that declare their intended ranges.
- *Axiomatic statements* (`$a`) that declare Metamath axioms, which can be associated with some *essential statements* (`$e`) that declare the premises.
- *Provable statements* (`$p`) that state Metamath theorems and their proofs.

Figure 4 shows how the 8 matching logic syntax constructs (2) can be formalized in Metamath.

```
$c #Pattern         $.
$c #ElementVariable $.
$c #SetVariable     $.
$c #Variable        $.
$c #Symbol          $.

$v ph0 ph1 $.
$v x y     $.
$v X Y     $.
$v xX yY   $.
$v sg0     $.

ph0-is-pattern $f #Pattern ph0 $.
ph1-is-pattern $f #Pattern ph1 $.

x-is-element-var
   $f #ElementVariable x $.
y-is-element-var
   $f #ElementVariable y $.

X-is-element-var $f #SetVariable X $.
Y-is-element-var $f #SetVariable Y $.

xX-is-var $f #Variable xX $.
yY-is-var $f #Variable yY $.

sg0-is-symbol $f #Symbol sg0 $.
```

```
$c #Positive $.
$c \bot      $.
$c \imp      $.
$c \app      $.
$c \exists   $.
$c \mu       $.
$c ( )       $.

$( Matching Logic Syntax $)

element-var-is-var $a #Variable x $.
set-var-is-var     $a #Variable X $.
var-is-pattern     $a #Pattern xX $.
symbol-is-pattern  $a #Pattern sg0 $.
app-is-pattern
  $a #Pattern ( \app ph0 ph1 ) $.
bot-is-pattern $a #Pattern \bot $.
imp-is-pattern
  $a #Pattern ( \imp ph0 ph1 ) $.
exists-is-pattern
  $a #Pattern ( \exists x ph0 ) $.
${
    mu-is-pattern.0
      $e #Positive X ph0 $.
    mu-is-pattern
      $a #Pattern ( \mu X ph0 ) $.
$}
```

Figure 4: Formalization of matching logic syntax (2) in Metamath

Notations derived from the 8 syntax constructs as seen in Section 2.2 can also be formalized in Metamath. A new constant statement `#Notation` needs to be declared first before using it to capture the *congruence relation of sugaring/desugaring*, i.e., notation. Figure 5 shows how the notation $\neg\varphi \equiv \varphi \to \bot$ can be formalized in Metamath.

```
$c #Notation$.

$c \not $.

not-is-pattern $a #Pattern ( \not ph0 ) $.
not-is-sugar $a #Notation ( \not ph0 ) ( \imp ph0 \bot ) $.
```

Figure 5: Formalization of $\neg$ as notation in Metamath

Lastly, axiom/proof rules can also be formalized in Metamath so to provide us with a proof system to prove and verify theorems with. Figure 6 illustrates how proof rules can be added in order to prove theorem, $\varphi_1 \rightarrow \varphi_1$ in Metamath.

```
$c |- $. $( This declares the entailment relation. $)

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 ) ) ) $.
${
    proof-rule-mp.0 $e |- ( \imp ph0 ph1 ) $.
    proof-rule-mp.1 $e |- ph0 $.
    proof-rule-mp   $a |- ph1 $.
$}

imp-refl $p |- ( \imp ph1 ph1 )
$=
  ($ Proof of the ph1 => ph1 goes here. $)
$.
```

Figure 6: Formalization of part of FOL and the proof of $\varphi_1 \rightarrow \varphi_1$ in Metamath

More on how matching logic can be formalized in Metamath can be found in [24]. Recall that language definitions written in $\mathbb{K}$ are compiled into the Kore format instead of directly to matching logic primitives. Thus, it is also necessary to have these Kore notations to be formalized in Metamath in order for theorems, which are expressed in Kore format, to be proven. Full formalization of matching logic and Kore notations can be found in this directory of the GitHub repository[4]. We have also experimented with verifying the mathematical proofs generated from the programs through zkVMs, which will be covered in Section 4.2.

3.5.2 Proposed Block Model

The Block Model is a proof format that we specifically designed for efficient ZK verification of mathematical proofs. The Block Model organizes proofs into a series of blocks, each of which requires a set of premises and produces a set of conclusions. We hard code the rules of inference for the proof system into the construction of these blocks, and the prover can then construct a proof by providing a series of blocks.

The structure of the blocks allows for a more efficient verification process, as the only connection between blocks is checking that all premises of a block were produced somewhere as a conclusion. This can be checked with the same techniques zkVMs use to check that a memory read instruction produces a value matching an earlier write, simplified further because blocks are unordered. The order independence also makes it easy to divide checking a large proof across many copies of a fixed size circuit. A succinct ZK proof for a formal proof can be produced more efficiently than separately producing a SNARK for each instance of the circuit by taking advantage of this "data parallel" structure. As

[4] `https://github.com/runtimeverification/proof-generation/tree/main/theory`

this Block Model is specifically designed to fit our Proof-of-Proof pipeline, we will postpone the discussion to Section 5, which is dedicated to provide more details on the Block Model.

4 Implementing Math Proof Checker in zkVMs

The math proof generator and the proof checker described in Section 3 effectively eliminate $\mathbb{K}$ and its gory implementation details from the trust base. Indeed, the role of $\mathbb{K}$ is reduced to searching for and generating a mathematical proof for the computational claim that was made. That is, the claim is a theorem and its actual computation, or execution produced by $\mathbb{K}$, is turned by the MPG module into its mathematical proof. The proof checker then verifies the produced math proof and if that is correct, then the claim is correct. $\mathbb{K}$'s correctness is therefore irrelevant, as far as it produces a math proof that checks.

In other words, the proof checker is the centerpiece that supports and enables the universal correctness of computing: it can verify any computation done with any program in any PL/VM, directly against the definition of the PL/VM, enforcing correctness-by-construction; additionally, it does it at no additional runtime overhead, as it verifies the proof as it is being generated. Because of the above we can argue that, from some perspective, the math proof checker, which is itself a program which totals only a few hundred lines of code, is one of if not the most important program ever written. But it is a program nevertheless, so it can be compiled and executed on off-the-shelf zkVMs. This is the fastest and cheapest way to obtain a PoC implementation of our Proof of Proof paradigm.

4.1 Background

Zero-knowledge virtual machines (zkVMs) are a type of software that supports producing zero-knowledge proofs of execution for some target language, for example, RISC-V or the Ethereum VM instruction set. Since the proofs are ideally much faster to check than it would be to execute the original program, this enables trustless computing, where one untrusted party (the prover) can perform a computation on behalf of another party (the verifier), and then quickly convince the verifier that the result produced was the result of a correct program execution, without tampering or errors. These computations can be fairly arbitrary. Any program that can be defined in the target language can be proven, within resource constraints. zkVMs can also support additional privacy-preserving features limited only by the expressiveness of the target language: the prover may provide their own inputs to the program, which are not revealed to the verifier. In this way, the prover can prove to the verifier that they know a piece of data satisfying some criteria without revealing the data.

Zero-knowledge proofs in general, and zkVMs in particular, have been seen as a way to address scalability problems with blockchains. In traditional models, smart contract code must be independently re-executed by many parties. If ZK is used instead, only one party needs to execute the program, while si-

multaneously producing the small zero-knowledge proof, and then instead of re-executing the whole program, other parties can simply run a short verification procedure on the proof.

4.2 Experiments and Benchmarks

Metamath (MM) is a minimalistic format for specifying mathematical proofs. In Metamath, proofs are defined by specifying variable and constant symbols, axioms with or without hypotheses over strings of those symbols, and then finally giving the proof itself, which is a list of labels of previously declared statements, which can be applied mechanically to gradually build up a string of symbols culminating in the proved theorem. More information on Metamath can be found in Subsection 3.5.1. Metamath was chosen as the format we would check proofs in for this experiment because it is a well-established, widely-known format, it is expressive enough to be used to represent our proofs of execution, and provide flexibility in how we do so, and because it is simple to understand and to write checker programs for.

The checking procedure, which we implemented in several languages to be compatible across all zkVMs we tried, is to first load all of the symbols and axioms into memory, and then read each given proof step in order. At each step, a new string of symbols is added to a stack. Depending on which rule is invoked, some stack entries are also consumed. For axioms with hypotheses, variable substitutions must be done on stack entries to ensure all the hypotheses are satisfied. The appropriate substitution is then made to the rule conclusion which is then placed on the stack. At the very end, it is verified that the stack contains exactly one entry, and that this matches the theorem that was originally claimed.

We implemented our Metamath proof checker in seven different zkVM's: Cairo, Jolt, Lurk, Nexus, Risc Zero, SP1 and zkWASM. Each implementation consists of a guest program which runs on the specific virtual machine and a host program which is our interface to actually running the guest, providing input for it and processing its output.

Five out of the seven tested zkVMs (Jolt, Nexus, Risc Zero, SP1 and zkWASM) provide a Rust compiler. Therefore, for them we have been able to develop and use a shared library for checking Metamath proofs, and thus the comparison among these five zkVMs should be considered more precise, as they share most of the code. Both Cairo and Lurk use domain specific languages (Cairo, respectively, LISP). While in Cairo and Lurk we implemented the same program (a Metamath checker), they were implemented independently of the rust code base and independent of each-other, so the comparison between them and the the Rust-based zkVM's should be taken with a grain of salt.

Only Risc Zero, zkWASM, SP1 and Cairo provide GPU support among the zkVMs that we have considered. Still, we were only able to run Risc Zero and zkWASM with GPU support due to internal setup issues for SP1 and evolving code base for Cairo.

For each of the zkVMs we've been using the default type of certificate offered

by that particular zkVM. For example, the default means composite certificates for Risc Zero and SP1 and succinct certificates for zkWASM. We've experimented with generating succinct certificates for Risc Zero and compressed certificates for SP1; the elapsed times to generate the shorter certificates seemed to be 1.6 times larger than that for composite certificates.

The zkVM space is evolving rapidly, with frequent releases of new zkVM versions and the provers they rely on. For our benchmarking, we used the following versions:

- **Cairo** (the Lambdaworks prover): Main branch, commit a591186, authored 2024-09-25 (the current version, while faster, does not yet support Cairo).
- **Jolt**: Main branch, commit 3b14242, authored 2024-11-04.
- **Lurk**: Main branch, commit 57c48b9, authored 2024-11-05.
- **Nexus**: Version 0.2.3, authored 2024-08-21
- **Risc Zero CPU**: Version 1.0.5, authored 2024-06-30.
- **Risc Zero GPU**: Version 1.2.2, authored 2025-01-27.
- **SP1 CPU**: Dev branch, commit 2c78683, authored 2024-11-05.
- **SP1 GPU**: Version 4.0.1, authored 2025-01-17.
- **zkWASM**: Main branch, commit f5acf8c, authored 2024-10-19.

4.2.1 How we Generated the Files

Our full benchmark suite consists of 1225 Metamath files split into two classes:

- A class generated from standard Metamath databases [1] and [2] by dividing them into small lemmas. These tests can be found under
  checker/mm/common/metamath-files/benchmark_mm/small
  in the Docker image that can be pulled from here; The instructions to run the Docker image can be found here.
- A class of Pi Squared proofs of execution of various lengths, which prove the correctness of an ERC20-like program written in the IMP programming language. These tests can be found under
  checker/mm/common/metamath-files/benchmark_mm/imp.

  If you want to find more on how we generate mathematical proofs of program executions, check out our proof generation demos and our documentation.

4.2.2 Optimizations

In the course of our examination, we made a few optimizations to the original Rust implementation of Metamath we started with for the matching logic use case. These included:

- Deduplication of strings stored in memory by managing using references to a single copy of each string.
- We reorganized the framestack to be a vector of frames, rather than a BTreeMap. This allowed us to avoid the overhead of maintaining the BTreeMap, and allowed us to use a more efficient data structure for the framestack.
- Circumvented derserialization of the proof by directly reading the proof file preparsed into a proof object into the zkVM.

Ultimately, the most significant single contribution to the runtime of the checker was the substitution of strings used in Metamath's core consistency checking step. Activity in this method ultimately amounted to around 15% of the runtime of the checker, with other major contributors being the parsing of the proof file and the construction of the frame stack.

4.3 Results

These checkers implemented on each of the zkVMs presented here were all tested on the same machine, with a AMD EPYC 9354 32-Core CPU, 4x NVIDIA GeForce RTX 4090 24GB GPU's, and 248GB RAM.

In order to save time, for each zkVM we run only some of the 1225 files, which makes the lines from Figure 7 to be rather approximations of the points corresponding to the measured files. This is the reason for which, for some particular files, one particular zkVM could behave better than other one, even if the figure doesn't show this. For a more precise comparison, we encourage you to check our GitHub repository.

4.3.1 How Different zkVMs Behave

The eight Metamath files referenced in Tables 1 through 7 were chosen to be representative of a wide range of input sizes. Times are measured in seconds, and input size is measured in number of Metamath tokens. A 900 second time limit was imposed, and where the result is "TO / OOM", either the time limit was exceeded or the checker used up all the system's memory. The Nexus checker took 512 seconds on hol_idi.mm, so a table was not included here.

4.4 Lessons Learned

The Rust implementation of the checker got the best results, specifically on the SP1 and RISCZero VMs, both in terms of speed and maximum memory usage.

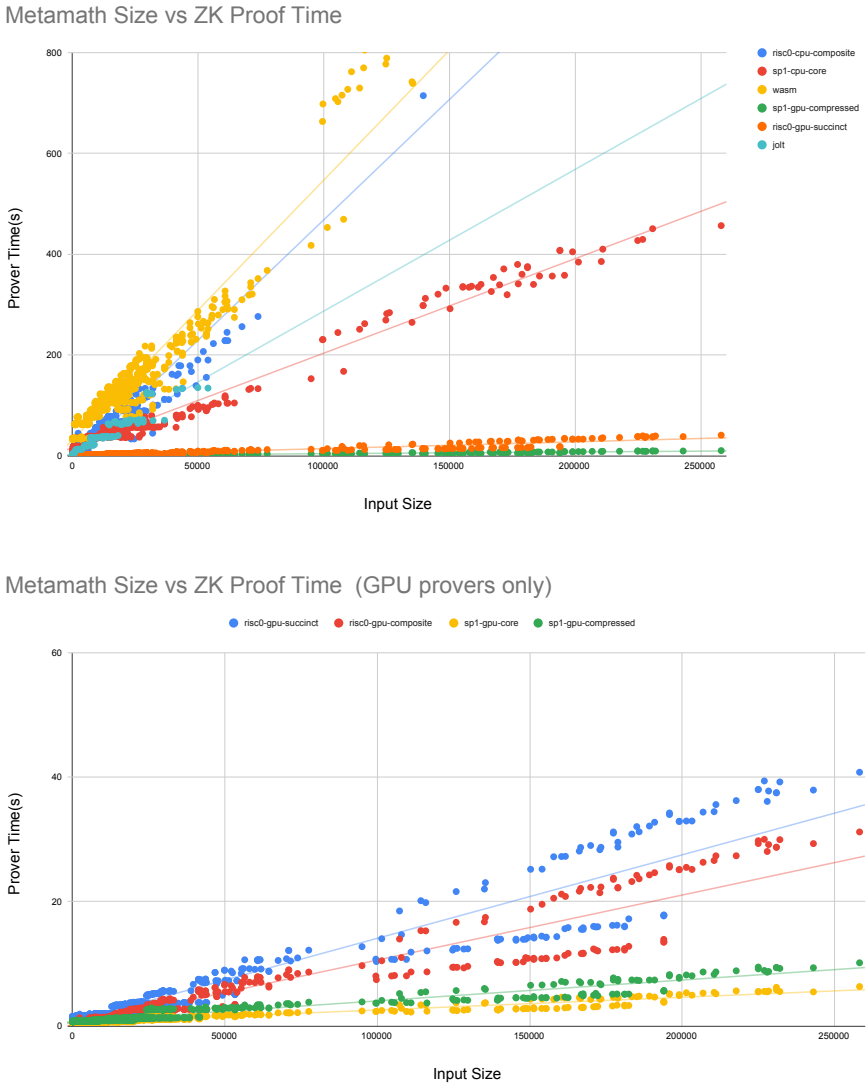

Figure 7: zkVM Comparison Table

Table 1: Cairo

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	0.913	0.029
hol_wov.mm	147	5.975	0.229
hol_ax13.mm	508	25.623	0.993
hol_cbvf.mm	1786	110.230	4.321
45.erc20transfer_success_tm_0_6.mm	6249	228.487	9.102
25.erc20transfer_success_tm_0_9.mm	21332	TO/OOM	TO/OOM
3.erc20transfer_success_tm_0.mm	73862	TO/OOM	TO / OOM
9.erc20transfer_success.mm	258135	TO/OOM	TO/OOM

Table 2: Jolt

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	2.04	0.08246
hol_wov.mm	147	2.89	0.06515
hol_ax13.mm	508	5.25	0.08273
hol_cbvf.mm	1786	12.09	0.08691
45.erc20transfer_success_tm_0_6.mm	6249	21.15	0.0848
25.erc20transfer_success_tm_0_9.mm	21332	65.51	0.09053
3.erc20transfer_success_tm_0.mm	73862	TO/OOM	TO/OOM
9.erc20transfer_success.mm	258135	TO/OOM	TO/OOM

Table 3: Lurk

Benchmark	Input Size	Proving time
hol_idi.mm	39	0.924
hol_wov.mm	147	5.588
hol_ax13.mm	508	18.167
hol_cbvf.mm	1786	195.757

Table 4: RISC0 (GPU), Succinct proof mode

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	1.47	0.01526
hol_wov.mm	147	1.44	0.01527
hol_ax13.mm	508	1.44	0.01525
hol_cbvf.mm	1786	1.61	0.01528
45.erc20transfer_success_tm_0_6.mm	6249	1.58	0.01521
25.erc20transfer_success_tm_0_9.mm	21332	1.96	0.01526
3.erc20transfer_success_tm_0.mm	73862	10.93	0.01529
9.erc20transfer_success.mm	258135	40.78	0.01525

Table 5: RISC Zero (CPU), Composite proof mode

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	3.140	0.016
hol_wov.mm	147	5.770	0.017
hol_ax13.mm	508	11.220	0.018
hol_cbvf.mm	1786	33.900	0.035
45.erc20transfer_success_tm_0_6.mm	6249	33.480	0.035
25.erc20transfer_success_tm_0_9.mm	21332	66.280	0.053
3.erc20transfer_success_tm_0.mm	73862	276.440	0.225
9.erc20transfer_success.mm	258135	TO/OOM	TO/OOM

Table 6: SP1 (CPU), Core proof mode

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	7.260	0.203
hol_wov.mm	147	12.220	0.199
hol_ax13.mm	508	17.450	0.199
hol_cbvf.mm	1786	34.860	0.208
45.erc20transfer_success_tm_0_6.mm	6249	4334.790	0.207
25.erc20transfer_success_tm_0_9.mm	21332	43.340	0.338
3.erc20transfer_success_tm_0.mm	73862	133.150	0.731
9.erc20transfer_success.mm	258135	456.790	2.490

Table 7: SP1 (GPU), Compressed proof mode

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	0.6914	0.09137
hol_wov.mm	147	0.7096	0.09151
hol_ax13.mm	508	0.6942	0.09091
hol_cbvf.mm	1786	0.7601	0.09168
45.erc20transfer_success_tm_0_6.mm	6249	0.7447	0.09047
25.erc20transfer_success_tm_0_9.mm	21332	1.19	0.09091
3.erc20transfer_success_tm_0.mm	73862	3.47	0.09065
9.erc20transfer_success.mm	258135	10.15	0.0907

Table 8: zkWASM (GPU)

Benchmark	Input Size	Proving time	Verification time
hol_idi.mm	39	33.080	4.059
hol_wov.mm	147	33.020	4.045
hol_ax13.mm	508	34.090	4.063
hol_cbvf.mm	1786	76.550	4.092
45.erc20transfer_success_tm_0_6.mm	6249	79.030	5.063
25.erc20transfer_success_tm_0_9.mm	21332	120.720	5.072
3.erc20transfer_success_tm_0.mm	73862	351.660	8.034
9.erc20transfer_success.mm	258135	TO/OOM	TO/OOM

However, even in these best cases, producing a ZK proof of a math proof of a simple ERC20-style transfer would take many minutes. This execution proof is generated using the semantics of a toy imperative language, but the proof sizes with a real-life language like Solidity could be significantly larger.

Checking large proofs (proofs of program execution are necessarily going to large) seemed to pose a problem for several of the zkVMs. In these cases the memory required to check a proof typically grew monotonically in the size of that proof. Some zkVMs, like RISCZero and SP1 use techniques to avoid excessive memory use which involve splitting up the execution transcript into chunks (These are referred to as shards in SP1 and segments in RISCZero) and recursively combining these.

We believe that a ZK circuit built to check math proofs in a specific format has the potential to be significantly faster, up to 1400× times. This is in part because there is no need to check memory consistency, and additionally because it would avoid the overhead of an intermediate step to make the proof logic compatible with Metamath and a VM architecture; see Subsection 5.4.8 for more details.

We would like to thank all the mentioned zkVM providers for having provided us feedback on these benchmarks and suggestions to improve our existing proof checking algorithm. We know that the field is continuously evolving and they are getting better and better with any release. We are happy to receive any further news on the improvements that they are going to make and to update our benchmarks accordingly.

5 Block Computation Model

In this section we propose a customized approach for checking proofs. The design was inspired by considering how techniques used to implement instructions in zkVMs could be applied more directly to proof checking, avoiding overhead inherent to using any zkVM for proof checking. When checking a proof, the only condition which needs to consider more than one proof step is checking

that the hypotheses of one step are all conclusions of other steps. This can be implemented directly with a *lookup argument*(eg [13, 35]). Each hypothesis becomes a single entry in the *lookups* list, each conclusion a single entry in the *table* list. In comparison a zkVM implements memory operations using a *memory checking argument*([2]). Each read or write lists the current and previous access times and values for the memory address as two entries in a *permutation argument*. A proof checking program will need several memory accesses to handle each hypothesis of a proof step.

The second simplification is to have no execution model. Expressing the task to be proved in zkVM as "run this code on this input" allows a short description of the computation, but the ZK proving process requires materializing an entire transcript of the computation before the heavy cryptographic work. So the input to computation that dominates prover time is still large, and a more complicated constraint system is needed to enforce that the transcript sticks to the control flow of the execution model. Instead we allow locally well-formed proof steps to be checked in any order, and any desired "execution model" such as Prolog-style resolution is handled before ZK by choosing what blocks to generate.

5.1 Block Model

The block model gives a language for defining *rules*, a format for writing *transcripts*, and a condition for a transcript to be valid according to a set of rules, designed so that validity of a transcript can be efficiently checked in ZK, but also so that that the problem of checking a mathematical proof in a conventional logic can be translated to checking a transcript against a set of rules (which depend only on the logic). The "public data" of a ZK instance is a prefix of a transcript, which may also leave some values in those blocks unspecified. The full transcript must extend this specification. We will see that a goal of proving a specific formula can be specified in this way, so that any valid transcript extending the given prefix must include a proof of that goal.

To conveniently support the recursive structure of proof steps in a conventional logic, and also recursively defined auxiliary functions such as substitution which are also common in such logics, the transcript is organized as a loosely coupled collection of *block instances*, which interact only by emitting and demanding *claims*, which can represent the logical judgments that appear in hypotheses and conclusions of proof rules, or claims that some application of an auxiliary function returns a particular result. Each of a proof rule, a clause of a recursive function definition, or production of a syntax definition generally corresponds to a single block rule.

To suit the lower ZK layers, claims are represented by tuples of field elements. Each *relation* (kind of claim) has a fixed arity, and takes only field elements rather than structured data as arguments, such as `is_impl(T,TA,TB)` or `proved(T,k)`. By giving code numbers to the relations and padding with trailing zeros, all kinds of claims are uniformly represented. Structured data such as syntax trees or a set of substitutions must be translated into this representation. This is a standard technique when applying Datalog to program

analysis. Families of relations are set up to describe tree nodes and immediate-child relationships, and data is represented by assigning each subterm a name, which is just any arbitrary field element, and emitting claims describing the relationships. To support declaring such data we also have a notion of *unique claims*, where a transcript is not allowed to have multiple blocks emitting identical unique claims. This is used to avoid any value having multiple incompatible definitions as a term name. Some relations may also be given primitive semantics, where some claims in that relation will be considered satisfied even if no block emits them. In the ZK implementation, this corresponds to using specialized accelerator circuits rather than the general purpose block claims to generate facts about e.g. cryptographic primitives, and adding these extra facts into the lookup argument.

The body of a block rule definition is the list of emitted and demanded claims, where the arguments can be variables or constants. The list of hypotheses can also include some primitive constraints that must hold between variables, such as inequality. In pseudocode, we allow arbitrary mathematical expressions as primitive constraints. The block rule also has a name, declares an order of the variables for writing down the local variables, and indicates which of the variables are included in the public data if the block appears in the public part of a transcript. Some block rules, such as those that emit axioms as part of a programming language definition, might only be allowed in the public part of the transcript.

A transcript is valid if every block instance follows the form of some block rule (while satisfying the primitive constraints), no unique claims are duplicated, and every demanded claim is satisfied.

5.1.1 Block Model Notation

We write rule definitions with concrete syntax inspired by Datalog. We do, however, write `-:` rather than `:-`, because the second looks too much like a turnstile $\vdash$ pointed in the wrong direction, and the semantics are not close enough for exact syntactic compatibility to be useful. All ordinary claims are written as a name applied to variables. Any other expressions among the hypotheses represent primitive constraints or perhaps are pseudocode that could be elaborated into additional claims.

If negation is derived syntax, we might have a block rule for decomposing negation named `check_not` with claims described by

```
check_not(TN,T):
  is_not(TN,T) -: is_impl(TN,T,TB), is_bot(TB)
```

The arguments on the header line are used when listing blocks in a transcript. If only some of the arguments will be specified when the block is used in the public portion of the transcript, the argument list is divided with a semicolon (`;`). When writing example transcripts informally we might leave out trailing arguments that can be inferred from the rest of the transcript.

Unique claims will be tagged `UNIQUE`. To declare an implication pattern we might have a block

```
def_pat_impl(T,TA,TB,k,ka,kb):
  UNIQUE wf_pat(T), wf_pat(T,k), is_impl(T,TA,TB)
   -: wf_pat(TA,kA), wf_pat(TB,kB), k > max(ka,kb)
```

The unique claims are in a different namespace, so we can have `UNIQUE wf_pat` take one argument, and `wf_pat` take two. We don't want to allow a conflicting definition of a pattern named `T` as a term with another depth so we can't include `k` in the arguments of the unique claim, but we must include the argument in the ordinary claim to enforce that defined terms are acyclic.

Restricted blocks will be tagged `SETUP` on the header line. This is used especially and perhaps only for emitting axioms when setting up a program instance.

```
assumption(T): SETUP
  proved(T,0) -: wf_pat(T,_)
```

5.2 Using the Block Model

We will now describe a few patterns for applying the above block model that are useful for implementing a proof system, especially how to handle term syntax and functions. This description should explain that the model is sufficiently expressive to express mathematical proofs, and motivate the parts of the design such as unique claims which are not similar to ideas found in Datalog. For an extended example see the propositional logic proof system in Section 5.3.2.

5.2.1 Term Representation

Term construction and pattern matching can be handled through claims as well. This might seem a bit inefficient compared to string-based or pointer-based representations in conventional programs, but recall that RAM itself is implemented in zkVMs by a permutation argument. The zkVM essentially communicates claims like "the result of reading address `a` at cycle `t` is `v`" between widely separated instructions in the transcript (the successive instructions accessing a memory location), by using the mathematical techniques that can be applied directly to claims like "term `x` is constructor `C` applied to `x1` and `x1`".

We encode tree or DAG data by giving every node a name, which are single field elements. Every parameter of a proof rule or judgment has a known type, so it's not a problem if a field value (such as a small number) is used as a name in multiple types, as long as it has a unique interpretation in each type.

For a type `T` we will have a relation for being a well-formed `T` value, whose arguments are the name and a "depth", like `wf_T(x,d)`. For each constructor `TCk` we will also have a relation saying that the value is formed by applying that constructor to specified arguments, like `is_T_C1(x,x1,...,xk)`. (allowing claims with many arguments would require the ZK implementation to handle

wider tuples, efficiency might motivate packing the argument lists themselves as a sort of data, but applicative matching logic formulas only need constructors with two arguments).

To ensure these claims are coherent, every block that emits a `wf_T(x,d)` claim will also emit a unique claim `UNIQUE wf_T(x)` to ensure there is only one interpretation of that name as a value of type `T`.

For each constructor there will be an associated block rule that will emit the general `wf_T` claims along with the appropriate constructor claim `is_T_Ci`, while consuming well-formed value claims about all the arguments, and constraining the depths. Any extra side conditions on the constructor could also be checked here. There could also be a block for declaring opaque terms that just emit the `wf_T` claims without any specific constructor claims.

Any logical rule that requires a specific shape of formulas in hypotheses or conclusions, such as the expression $\varphi \to \psi$ in the hypothesis $\vdash \varphi \to \psi$ of a Modus Ponens rule, can be transformed by adding extra variables to name the larger expressions and extra hypothesis about the syntactic relationships between those variables, so the hypotheses and conclusions that claim something has been proved will only talk about bare variables. This leaves a logical rule that can be translated directly to a block rule. For example, rather than a Modus Ponens rule concluding $\vdash B$ from hypotheses $\vdash A \to B$ and $\vdash A$, it could instead be written in terms of three variables T, A, B, and conclude $\vdash B$ from $\vdash A$, $\vdash T$, and an additional hypothesis $T = A \to B$ (which corresponds to the `is_impl` relation).

5.2.2 Function Representation

Functions over the defined data types can be modeled with a relation taking an extra parameter for the output, such as a relation `subst(T,TB,X,TR)` to mean `T = TB[TR/x]`. There will be a block rule for each clause of the function definition, which will consume constructor claims to examine the arguments and also enforce the shape of the result, and consume additional claims corresponding to any function calls in the clause, recursive or otherwise. Termination is usually ensured by structural recursion along the already-acyclic data, but could be enforced by an extra argument if needed.

Note that when using a block in this style, there must already be term-defining blocks declaring a name and the structure of the result term. For functions thought of as producing new terms, it would also be possible to have variants of the function-defining block rules that also emit the data-defining claims about the result value. However, we would always need to keep the non-defining version of every block rule available, to avoid a uniqueness error or preserve sharing in case the result of the application had already been independently produced elsewhere.

5.2.3 Acyclicity

For both proofs and term declarations, we will be breaking up a DAG structure into blocks and we want to enforce that it is acyclic. In Hilbert proofs, this relies on ordering of the steps, but the standard constructions for a lookup constraint in a zkSNARK are unordered. It is easy to enforce acyclicity at the definition level by adding an extra "depth" argument to relations and extra constraints in the block rules that enforce that hypotheses have a smaller "depth".

To enforce that the "depth" is exactly the depth we would need not only the inequalities `k > k_i` for each hypothesis depth `k_i`, but also to enforce that `k == k_i + 1` for at least one hypothesis (or to use a different approach that accumulates the maximum across all `k_i`). But nothing is harmed by only checking the inequality constraints because that is already sufficient to ensure derivations are acyclic, and provers are still allowed to use the actual depth. It does not seem useful to constrain the depth of proof trees, but even if we did, the only natural constraint is imposing a maximum depth, which can still be enforced by putting a maximum on the "depth".

A useful property of depths is that they are (or can be chosen to be) relatively small numbers, and it is more efficient to define inequality over a smaller domain than the entire field. The actual depth is bounded by the number of blocks, so supporting numbers up to 2^{40} or so would cover any possible proof with a feasible transcript size, but *any* size is still sound. Rather than hard-coding the less-than relation up to a fixed size, note that inequality reduces to a "range check" - if there is a `smallPos` predicate that tests $0 \leq x < N$, then $0 \leq a < b \leq N$ if there exists a p with $a + p = b$, and `smallPos(p)`, `smallPos(b)`. If we just want a well-founded relation and $(2^{40})^2$ doesn't overflow the size of the finite field, we might even just check `a + p = b` and `smallPos(p)`.

The main sources of proof size in our applications are taking many rewrite steps in an execution, and working over large program configurations. In both cases by using balanced trees a logarithmic depth suffices. There are efficient ZK implementations of range checks, but for writing self-contained block model examples, spending $O(N)$ blocks to populate a `smallPos` table up to maximum value N can be affordable.

Note that acyclicity could be enforced with *any* well-founded relation, if another relation is cheaper than inequality. Over the integers, term size looks like a simple option, with the size of a new term defined by a simple equation $k = 1 + k_1 + \ldots + k_n$. But that does not work in modular arithmetic; sharing makes it easy to construct terms whose unshared size is exponentially large and induce overflows.

5.3 Block Model Example: Propositional Logic

As a further example of the block model, we present an implementation of propositional logic. Rules from this example will also be used later when discussing how the block model can be implemented. We first introduce the main relation for claiming that a formula has been proved and other relations that

will be needed to state the rules of propositional logic, then show the block rules implementing the proof rules, and then show how to implement the auxiliary relations, including block rules and additional relations.

Because the point is to be a good example, we include metavariables and an instantiation rule and a block for generically asserting a term as an axiom, but also provide block rules that directly check axiom schemes. Without instantiation we wouldn't have any auxiliary recursive operations over terms, and the axiom schemes demonstrate how to translate proof rules with complicated expressions.

5.3.1 Main Claim

The main claim is `proved(T,k)`, meaning that the term represented by `T` has a proof of depth `k`.

For metavariable instantiation we have claim `instantiation(T,TP,S)` meaning that term `T` is the result of applying the metavariable substitution `S` to term `TP`. This already implies the substitution is well formed, so the proof rules don't need any claims about substitution syntax.

The most basic claim about terms is that `T` is the name of a particular well-formed term. In proof rules we write `wf_term(T,_)`, because term definitions need to include a depth argument too, but proof rules never care.

The other relations that we will need in the proof rules are those for applying a substitution, and for checking that a term has a particular shape. The claim `instantiation(T,TP,S)` means that term `T` is the result of applying substitution `S` to the term `TP`. The only syntactic claims we need for the proof rules are `is_impl(T,T1,T2)` which means `T` is the implication `T1->T2` and `is_not(T,T1)` which means `T` is the negation of `T1`. The block rules for defining these claims can only be produced when all of the term variables are already well-formed expressions, so block rules can often avoid separate `wf_term` hypotheses.

Block rules and further relations for defining term data will be given later, and consuming these claims in the proof rules does not require that these are actually primitive constructors in the term representation. Some presentations of propositional logic define implication in terms of disjunction and negation, or negation in terms of implication and a false term, or take both implication and negation as primitive and define conjunction and disjunction.

5.3.2 Proof Blocks

The `assert` block is only allowed in the setup phase, and emits a claim stating that a term has been proved. If the term definition is not also included in the public part of the transcript, the prover could choose any definition of the undefined (sub)term names. The emitted claim takes a provided depth k rather than just emitting at depth zero, so this rule could be used to exactly replace a provable lemma whose proof has a certain depth.

```
assert(T,k): SETUP
  proved(T,k) -: wf_term(T,_).
```

The `demand` block just consumes a claim that a term has been proved. This is meant to be used in the public part of an instance to force the prover to come up with a proof of the specified formula. It's useless but harmless in the private part of the transcript.

```
demand(T):
  -: proved(T,_).
```

The basic derivation rule is `modus_ponens`.

```
modus_ponens(T;TA,TB,k,k1,k2):
  proved(TB,k) -:
    is_impl(T,TA,TB), proved(T,k1), proved(TA,k2), k1 < k, k2 < k.
```

The other derivation rule instantiates metavariables.

```
instantiate(T,TP,S; k1, k):
  proved(T,k1) -:
    proved(TP,k), instantiation(T,TP,S), k1 = k+1.
```

The first axiom (K combinator) is $\mathtt{TA} \to \mathtt{TB} \to \mathtt{TA}$.

```
axiom1(T; TA,TB):
    proved(T,0) -:
      is_impl(TI,TB,TA), is_impl(T,TA,TI).
```

The second axiom (S combinator) is $(\mathtt{TA} \to (\mathtt{TB} \to \mathtt{TC})) \to ((\mathtt{TA} \to \mathtt{TB}) \to (\mathtt{TA} \to \mathtt{TC}))$. The variable name mnemonics are H for Hypothetical and I for Implication.

```
axiom2(T; TA,TB,TC,THB,THC,TI,THI,TIH):
  proved(T,0):
    is_impl(THB,TA,TB), is_impl(THC,TA,TC), is_impl(TI,TB,TC),
    is_impl(THI,TA,TI), is_impl(TIH,THB,THC),
    is_impl(T,THI,TIH).
```

The third propositional axiom covers negation, $(\neg\mathtt{A} \to \neg\mathtt{B}) \to (\mathtt{A} \to \mathtt{B})$.

```
axiom3(T; TA,TB,TI,TNA,TNB,TIN):
  proved(T,0):
    is_impl(TI,TB,TA),
    is_not(TNA,TA), is_not(TNB,TB),
    is_impl(TIN,TNA,TNB),
    is_impl(T,TIN,TI).
```

5.3.3 Auxiliary Claims

To define instantiation we need relations representing the term structure of substitution, and we also need to be able to recognize metavariable terms. The term syntax claim `is_mvar(T,V)` means term `T` consists of the metavariable `V`. Metavariable names and term names are distinct namespaces, so there is no necessary numerical relationship between the numbers.

A further auxiliary relation `subst_lookup(V,T,S)` gives the term resulting from looking up variable `V` in substitution `S`, this is needed in the block rule for defining substitution in the `is_mvar` case. To use the `instantiate` proof rule we need to be able to derive `instantiation` claims about arbitrary well-formed subterms, so if any terms are none of `is_impl`, `is_not`, nor `is_mvar` additional block rules will be required.

We define substitutions as built up from a substitution for a single variable `is_single_subst(S,V,T)` and unions of substitutions `is_union_subst(S,S1,S2)`. To have a sound proof system we will need to ensure we can't have both `subst_lookup(V,T1,S)` and `subst_lookup(V,T2,S)` with different `T1` and `T2`.

We could enforce at construction time that unions only join disjoint substitutions, or enforce at lookup time that the substitution only contains one definition for the replacement variable. For both cases we need an auxiliary relation `not_in_subst(V,S)` for checking that substitution `S` does not contain a replacement for variable `V`. We use the second option because the first would require another relation for asserting that two substitutions were disjoint. It is also efficient, unlike checking at each lookup in a conventional program, because `not_in_subst` and `subst_lookup` claims are reusable.

A well-formed substitution claim is only needed in the substitution constructor blocks themselves. The rules defining the `not_in_subst` would actually work correctly if cyclic terms were allowed, but we could derive false `subst_lookup` facts for cyclic terms (`subst_lookup(V,T,S)` for any `V` which does not actually appear in `S`, by using the union rules along a cyclic path in `S` and deriving true `not_in_subst(V,S')` claims for the other arguments of those unions).

Metavariables are atomic, so we don't need any sort of `wf_mvar(V)` claim.

5.3.4 Substitution Application

Substitution recurses over implication and negation

```
subst_impl(T,TI,S; TA,TB,TA2,TB2):
  instantiation(T,TI,S)
  -: is_impl(TI,TA,TB),
     instantiation(TA2,TA,S),
     instantiation(TB2,TB,S),
     is_impl(T,TA2,TB2).

subst_not(T,TN,S; TB,TB2):
  instantiation(T,TN,S)
  -: is_not(TN,TB),
```

```
    instantiation(TB2,TB,S),
    is_not(T,TB2).
```

At a metavariable, we use the lookup

```
subst_mvar(T,TV,S):
  instantiation(T,TV,S)
  -: is_mvar(TV,V),
     subst_lookup(V,T,S).
```

The above rules are already enough for a complete proof system, if the prover always constructs substitutions giving a replacement for all variables in the term, but we can give an additional block rule saying a metavariable outside the domain of the substitution is left unchanged, using the `not_in_subst` relation that we already need for other reasons.

```
subst_mvar_missing(TV,S):
  instantiation(TV,TV,S)
  -: is_mvar(TV,V),
     not_in_subst(V,S).
```

Now we define substitution lookup. The three cases cover finding the target variable in a single-variable substitution, or to the left or right of a union. To ensure lookup is deterministic as a function of `S` and `V` the union cases also check that the variable doesn't appear in the other side of the union. With this design a valid transcript could declare a union that has conflicting replacements for a variable

```
subst_here(S; V,T):
  subst_lookup(V,T,S)
  -: is_single_subst(S,V,T).

subst_left(S; V,T,SL,SR):
  subst_lookup(V,T,S)
  -: is_union_subst(S,SL,SR),
     not_in_subst(V,SR),
     subst_lookup(V,T,SL).

subst_right(S; V,T,SL,SR):
  subst_lookup(V,T,S)
  -: is_union_subst(S,SL,SR),
     not_in_subst(V,SL),
     subst_lookup(V,T,SR).
```

Checking that a variable is not in a substitution only needs two rules. The union case must recurse into both subterms.

```
subst_not_here(V,S; V1,T):
  not_in_subst(V,S)
  -: is_single_subst(S,V1,T),
     V != V1.

subst_not_union(V,S; SL,SR):
  not_in_subst(V,S)
  -: is_union_subst(S,SL,SR),
     not_in_subst(V,SL),
     not_in_subst(V,SR).
```

5.3.5 Substitution Definition

```
def_subst_single(S,V,T):
  is_single_subst(S,V,T),
  UNIQUE wf_subst(S), wf_subst(S,0)
  -: wf_term(T,_).

def_subst_union(S,SL,SR; k,kl,kr):
  is_union_subst(S,SL,SR),
  UNIQUE wf_subst(S), wf_subst(S,k)
  -: wf_subst(SL,kl), wf_subst(SR,kr),
     kl < k, kr < k.
```

5.3.6 Term Definition

Just to mix things up, we define terms so the primitive connective are implication ($\rightarrow$) and false ($\bot$, aka bottom), and negation is notation for an implication $T \rightarrow \bot$. For this we will have a claim `is_bot(B)`.

We could potentially try to fix a reserved term code for `bot`, but then transcripts would probably need to anyway include a special block to emit `UNIQUE wf_term(<bot>)` to prevent that code from being reused.

The block rule emitting `is_not` doesn't define a new term name.

```
check_not(T; TB,B):
  is_not(T,TB)
  -: is_impl(T,TB,B), is_bot(B).
```

Here are the ones that actually declare terms

```
def_term_bot(B):
  is_bot(B),
  UNIQUE wf_term(B), wf_term(B,0) -: .

def_term_mvar(T,V):
  is_mvar(T,V),
  UNIQUE wf_term(T), wf_term(T,0) -: .
```

```
def_term_impl(T,TA,TB; k,ka,kb):
  is_impl(T,TA,TB),
  UNIQUE wf_term(T), wf_term(T,k)
  -: wf_term(TA,ka), wf_term(TB,kb),
     ka < k, kb < k.
```

5.3.7 Depth Handling

We define a `depthLt` relation that can replace the $<$ primitive condition, to support later discussion of how to implement the block model in ZK[5]. This is implemented in terms of a `smallPos` relation which will also be explicitly populated with small positive numbers by block instances.

```
depthLtCheck(S,L; P):
  depthLt(S,L) -: L == S + P, smallPos(P).
```

We don't impose a range check `smallPos(L)`, because we could only hit an overflow if $N \cdot M$ exceeds the field size, where N is the number of `depthLtCheck` block instances and M is the number of `smallPos` instances, which would require a transcript length on the order of the square root of the field size.

```
smallPosOne():
  smallPos(1)
smallPosNext(k; k0):
  smallPos(k) -: smallPos(k0), k == k0 + 1.
```

5.3.8 Example Trace

To prove `A->A`, name `A->A` by `B`. Then `(A->(B->A))->((A->B)->(A->A))` is an instance of propositional axiom 2 (S combinator), and both `A->(B->A)` and `A->B == A->(A->A)` are instances of propositional axiom 1 (K combinator), so the conclusion follows by modus ponens in two steps.

Numbering the terms, we assign the first codes to `A`, `B` and the axiom instances, and then have to name only two further terms to have names for all subterms.

```
0 := v0 or A
1 := A->A (or B)
     0->0
2 := A->(A->A) or A->B
     0->1
3 := A->(B->A)
     0->6
4 := (A->(B->A)) -> ((A->B)->(A->A))
```

[5] Addition is easily handled in arithmetic circuits or polynomial constraints, but inequality is not so primitive.

```
     3->5
5 := ((A->B)->(A->A))
     2->1
6 := (B->A)
     1->0
```

These terms can be declared with blocks

```
def_term_mvar(0,0)
def_term_impl(1,0,0; 1,0,0)
def_term_impl(2,0,1; 2,0,1)
def_term_impl(3,0,6; 2,0,1)
def_term_impl(4,3,5; 4,2,3)
def_term_impl(5,2,1; 3,2,1)
def_term_impl(6,1,0; 2,1,0)
```

The proof claims can be emitted with

```
axiom1(2; 0,1)
axiom1(3; 0,6)
axiom2(4; ...)
modus_ponens(4; 3,5, 1,0,0)
modus_ponens(5; 2,1, 2,1,0)
```

which would satisfy a block `demand(1)`. For the depth checking these turn out to be enough inequalities and `smallPos` values.

```
depthLtCheck(0,1)
depthLtCheck(0,2)
depthLtCheck(0,3)
depthLtCheck(1,2)
depthLtCheck(1,3)
depthLtCheck(2,3)
depthLtCheck(3,4)
smallPosOne()
smallPosNext(2)
smallPosNext(3)
```

5.4 Implementation with Circuits and Folding

To implement the block model with arithmetic circuits (or R1CS constraints), one natural approach is to have specialized subcircuits corresponding to each block rule. A rule definition translates to a subcircuit with inputs corresponding to the parameters of the rule and outputs corresponding to the hypotheses, conclusions, and uniqueness tag (if needed) of the rule. To check consistency of a block transcript, multiple copies of each rule circuit will need to be connected to additional components that enforce that the hypotheses are all contained in

```
template modus_ponens() {
  signal input T,k, TA,k1, TB,k2;

  signal output conclusions[1][N()];
  conclusions[0] <== proved_claim(TB,k);

  signal output hypotheses[5][N()];
  hypotheses[0] <== is_impl_claim(T,TA,TB);
  hypotheses[1] <== proved_claim(T,k1);
  hypotheses[2] <== proved_claim(TA,k2);
  hypotheses[3] <== depthLt_claim(k1, k);
  hypotheses[4] <== depthLt_claim(k2, k);
}
```

Figure 8: Modus Ponens Subcircuit, expressed in Circom

the conclusions and enforce that the unique claims are unique. With additional Boolean inputs for disabling individual copies of the rule subcircuits, a single fixed circuit can check block transcripts of any size up to a certain maximum capacity. To remove this limit, we will modify the circuit so multiple copies can be used together to check arbitrarily large transcripts, and apply *folding* [21] to efficiently check as many copies as needed with a fixed size SNARK. This final circuit is called the *segment* circuit. An example rule subcircuit is shown in Fig. 8.

Creating a specialized circuit for a set of block rules requires fixing the set of rules in advance, but this is similar to fixing the set of proof rules for a mathematical logic, like the Matching Logic used for all languages in Section 3. In the mathematical proofs, a language semantics is given as a set of definitions, which corresponds to a collection of block instances in a transcript, using term-defining and axiom-declaring block rules in Section 5.3.6 and Section 5.3.2 in the propositional logic example.

5.4.1 Claim Representation

All claims can be represented uniformly as tuples of field elements. Consider all the relations in the chosen set of block rules. There is some maximum arity k. Each relation will be assigned a distinct tag, which is a nonzero field element. A claim will be represented as a tuple of $k+1$ field elements, with the first element being the tag of the relation, followed by the arguments of the claim, padded to length $k+1$ with trailing zeros.

In the propositional example, relations have at most three arguments, so 4-tuples suffice. A claim like `is_impl(2,0,1)` would be represented as a tuple $(\langle\texttt{is\_impl}\rangle, 2, 0, 1)$ where $\langle\texttt{is\_impl}\rangle$ is the tag for the `is_impl` relation, which could be a small value like 5 or 12. A claim `proved(2,2)` would be represented by $(\langle\texttt{proved}\rangle, 2, 2, 0)$.

For subset and permutation arguments, these tuples are represented as single field elements using a Fiat-Shamir parameter α. The tuple $(x_0, \ldots, x_k)$ is represented by $\sum_{i=0}^{k} \alpha^i x_i$, so $(\langle \texttt{proved} \rangle, 2, 2, 0)$ would be encoded as $\langle \texttt{proved} \rangle + \alpha 2 + \alpha^2 2$. Computing this representation of a `proved` claim in a specialized rule circuit only requires two multiplication operators - the powers of α are computed once and shared around the circuit, and no explicit terms are required for the trailing zeros. Tag values are nonzero so the representation zero (and the all-zero tuple) is distinct from any valid claim, and can be used as a dummy value.

5.4.2 Lookup Argument

For checking the subset relation, we use the LogUp lookup argument [13]. In primitive form, this checks that the set of values in a list $l_1, \ldots, l_k$ of field elements is a subset of the set of values in a list $t_1, \ldots, t_n$. The argument is based on the equation

$$\sum_{i=1}^{k} \frac{1}{l_i - \beta} = \sum_{i=1}^{n} \frac{m_i}{t_i - \beta}$$

When considering the expressions as rational functions of β and if k and n are much smaller than the characteristic of the field, this equality holds if and only if the l are a subset of the t and the m correctly count the multiplicity, in the sense that for all values $f \in \mathbb{F}$

$$\sum_{i | l_i = f} 1 = \sum_{i | t_i = f} m_i$$

The concrete evaluation of a uniformly random β is almost certainly equal only if the rational functions are equal. In an arithmetic circuit, quotients are not easy to compute purely in terms of addition and multiplication gates, but it is easy to take an auxiliary input q_i and force it to be the correct quotient with a constraint like $q_i(v_i - \beta) = m_i$. Consider rearranging the sum like

$$\sum_{i=1}^{k} \frac{1}{l_i - \beta} - \sum_{i=1}^{n} \frac{m_i}{t_i - \beta} = 0$$

Rearranging further, a rule circuit can output a single value which is the portion of this difference corresponding to the hypotheses and conclusions from that rule instance, and the overall circuit just needs to sum these contributions together. The flag for disabling a rule circuit can be applied by changing this output to zero without adding complexity to the internal circuitry. Furthermore, the overall sum will be an output of the segment circuit, instead of directly including a constraint that the sum is zero, allowing the lookup argument to span across many copies of the segment circuit by adding these "accumulator" outputs together during the folding process, and only checking that the final sum balances to zero.

5.4.3 Uniqueness

To enforce uniqueness of the unique claims emitted by the block rules we construct a lexicographically sorted copy of that list of claims. The copy is provided as additional circuit inputs. That a sorted list has no duplicates can be checked with just constraints between adjacent entries. That a list is sorted can also be checked with constraints just between adjacent entries. Enforcing that a list is a permutation of another is a non-local property, which can be checked with a *permutation argument*.

One classic permutation argument uses the polynomial equation

$$\prod_{i=1}^{n}(a_i - \beta) = \prod_{i=1}^{m}(b_i - \beta)$$

or we can use the LogUp equation with multiplicities all set to 1, so the accumulation operation is addition rather than multiplication.

When using multiple copies of the segment circuit to check a longer transcript, the sorting and the sorted list needs to span across all the circuits. As with the subset check over hypotheses, the accumulated sum from the permutation argument is an output of the segment circuit rather than being constrained to balance within the segment circuit, so the accumulator can be combined across all segments. To handle the constraints between adjacent entries in the sorted list, the segment circuit has an extra "previous unique claim" input treated as preceding the first entry in the sorted list, and outputs a copy of the last claim in the list. When folding together multiple segment circuit instances, it will be checked that the "previous unique claim" matches the last claim output from the previous segment, or is set to the dummy claim for the first segment.

5.4.4 Fiat-Shamir for Circuits and R1CS

The Fiat-Shamir heuristic implements a "random" choice of parameters such as α and β by applying a hash to (commitments to) "earlier data". This transforms an interactive protocol involving a true random choice into a non-interactive protocol, while trying to make attacking the resulting non-interactive proof take as many attempts as would be required to attack a true random choice.

If not enough data is included in the hash, this can be insecure even when the protocol would be secure for a true random choice, if the attacker can compute the hash output but then adjust "earlier" values that are not covered by the hash to hit a false positive condition that would be overwhelmingly unlikely to occur by chance.

Often the Fiat-Shamir heuristic is described with all values being directly fed into the hash, but if we are already computing a vector commitment $\bar{x}$ to a vector $\vec{x}$, then including the small commitment $\bar{x}$ in the hash also works to make the result depend on all entries of $\vec{x}$.

For a circuit, the Fiat-Shamir parameters will be provided as circuit inputs, and the data that might be included in the hash are any other circuit inputs or

intermediate values. A sufficient condition for security is that the values included in the hash combined with the Fiat-Shamir parameters uniquely determine the remaining inputs and intermediate values, in the sense of leaving at most one satisfying assignment to any inputs not included in the hash.

In the R1CS translation of a circuit, the circuit structure is encoded into the R1CS matrices A, B, C, and each position of the witness vector x maps onto an input, output, or intermediate value in the circuit, such that satisfying assignments to the circuit are equivalent to vectors x such that the extended vector $z = (1, x)$ satisfies the R1CS equation $Az \circ Bz = Cz$.

To correspond to a selection of circuit values to commit to, the entries of x can be ordered as $(\vec{d}, \alpha, \beta, \vec{a})$ where the d_i are the pre-Fiat-Shamir values and a_i are the remaining values.

Many folding schemes for R1CS only call for an additively homomorphic vector commitment $\bar{x}$. An example is the Pedersen vector commitment, defined by $\text{Com}(\vec{g}; \vec{x}) = \prod_i {g_i}^{x_i}$ for some choice of group and list $\vec{g}$ of generators. If there is no further requirement on the vector commitment scheme, we can define the commitment to an overall vector to be a tuple of commitments to subvectors such as $(\bar{d}, \overline{(\alpha, \beta)}, \bar{a})$, with addition and scalar multiplication defined pointwise.

This makes the vector commitments $\bar{d}$ and $\overline{(\alpha, \beta)}$ directly available in the proof certificate for the modified SNARK or folding scheme, so a verifier can also easily check that α and β were derived from a hash of $\bar{d}$.

5.4.5 Folding Schemes

A *folding* or *split accumulation* scheme for a relation $R \subseteq X \times W$ between instance descriptions X and witnesses W provides a *folding verifier* which is a partial function fold that takes as input two instances x_1 and x_2 and a *folding proof* π and either rejects or outputs another instance x_3. A prover knowing $(x_1, w_1), (x_2, w_2) \in R$ can efficiently compute π, x_3, w_3 with $x_3 = \text{fold}(x_1, x_2, \pi)$ and $(x_3, w_3) \in R$. The security property is that a computationally bounded party can only produce tuples $(x_1, x_2, \pi, x_3, w_3)$ with $x_3 = \text{fold}(x_1, x_2, \pi)$ and $(x_3, w_3) \in R$ if they also know w_1 and w_2 with $(x_1, w_1), (x_2, w_2) \in R$.

The size of an element of X should be much smaller than W. These schemes almost always have part of X consist of a vector commitment to W, particularly an additively homomorphic commitment.

Many papers about folding schemes, including the Nova paper [21] itself, discuss applying folding schemes as an ingredient in building the more complicated constructions of Incrementally Verifiable Computation(IVC) or Proof Carrying Data(PCD), which allow giving a SNARK for the claim that a recursively defined predicate holds, with a linear or general graph recursion structure, respectively.

But for handling the segment circuits in a block model proof, we just need to combine many instances of the segment circuit as siblings, rather than in a recursive structure. A small number of public parameters of the segment circuit need special handling while combining instances, such as accumulators for the lookup arguments, and Merkle trees over commitments to the pre-Fiat-Shamir

data, but for the rest of the circuit we only need to fold multiple instances of the circuit together.

5.4.6 Relaxed Committed R1CS

The folding scheme presented in the Nova paper is called Relaxed Committed R1CS. Further work such as HyperNova and ProtoGalaxy potentially offer improvements such as cheaper folding or being able to heterogeneously combine instances of different circuits into a single folded instance, but Relaxed Committed R1CS is particularly simple and sufficient to illustrate the folding-based implementation strategy.

As the name suggests, it is based on a relaxation of R1CS, and involves vector commitments. The "relaxed" part is changing the notion of a witness for a set of R1CS matrics A, B, C to be a tuple $(u, \vec{x}, \vec{e})$ of a scalar u and vectors $\vec{x}$ and $\vec{e}$ such that the vector $\vec{z} = (u, \vec{x})$ satisfies the equation

$$A\vec{z} \circ B\vec{z} = uC\vec{z} + \vec{e}$$

The tuple $(1, \vec{x}, \vec{0})$ with $u = 1$ and $\vec{e}$ the zero vector is a relaxed R1CS witness for A, B, C if and only if x is an ordinary R1CS witness for A, B, C. The "committed" part is modifying this to use vector commitments to fit the structure of a split relation. The instance part is a tuple $(u, \bar{x}, \bar{e})$ of scalar u and group elements $\bar{x}$ and $\bar{e}$ (in the codomain of a vector commitment scheme). The witness part is a pair of vectors $(\vec{x}, \vec{e})$, where $((u, \bar{x}, \bar{e}), (\vec{x}, \vec{e}))$ is in the Relaxed Committed R1CS relation defined by matrices A, B, C iff $\bar{x}$ is a vector commitment to $\vec{x}$, $\bar{e}$ is a vector commitment to $\vec{e}$, and $(u, \vec{x}, \vec{e})$ is in the Relaxed R1CS relation defined by those matrices.

The folding operation only needs to consider the small instance description $(u, \bar{x}, \bar{e})$. To fold two instances $(u_1, \bar{x}_1, \bar{e}_1)$ and $(u_2, \bar{x}_2, \bar{e}_2)$ an honest prover needs to know corresponding witnesses $(\vec{x}_1, \vec{e}_1)$ and $(\vec{x}_2, \vec{e}_2)$, and use them to compute a certain cross-term vector $\vec{t}$, and then a vector commitment $\bar{t}$, which is provided as the folding proof. To fold two instances together with a folding witness, a scalar r is computed by hashing all of $u_1, \bar{x}_1, \bar{e}_1, u_2, \bar{x}_2, \bar{e}_2, \bar{t}$, and the updated instance is computed as $u \leftarrow u_1 + ru_2$, $\bar{x} \leftarrow \bar{x}_1 + r\bar{x}_2$, $\bar{e} \leftarrow \bar{e}_1 + r\bar{t} + r^2\bar{e}_2$.

The security argument in the paper establishes that it is infeasible to find vectors $\vec{x}$ and $\vec{e}$ such that $((u, \bar{x}, \bar{e}), (\vec{x}, \vec{e}))$ satisfy the Relaxed Committed R1CS relation, unless $\bar{t}$ was honestly computed from knowing satisfying witnesses for both input instances, or the adversary can break the hash or vector commitment.

5.4.7 Segment Folding

Following the circuit design earlier in this section, the segment circuit has a fixed interface independent of the number of block rule subcircuits included in each segment.

These circuit inputs or outputs that need to be inspected to combine multiple segments are

- The subset accumulator A
- The uniqueness accumulator B
- The used Fiat-Shamir parameters α, β
- The "previous" and "last" unique claim tuples $\vec{u}_{\text{in}}, \vec{u}_{\text{out}}$.

An honest prover begins transforming a valid block model transcript into a list of segment circuit instances by distributing the block instances of the transcript among the segments, setting the rule subcircuit arguments and enabling flags accordingly, also setting the conclusion multiplicities, and the sorted list of unique claims. This determines all the pre-Fiat-Shamir data, so the prover can then compute vector commitments to the pre-Fiat-Shamir parts of the circuit witnesses, and hash up the Merkle tree over those commitments to derive the α, β. With those parameters determined the rest of the circuit can be evaluated, and vector commitments to the rest of the circuit witnesses computed, and then the Relaxed Committed R1CS instances and witnesses folded together.

A list of satisfying assignments to the segment circuit is consistent if they all use the same α and β, and the $\vec{u}_{\text{in}}$ of each segment equals the $\vec{u}_{\text{out}}$ of the previous. It additionally describes a consistent block model transcript if the sum over all A is zero, the sum over all B is zero, the $\vec{u}_{\text{in}}$ of the first segment is the dummy zero tuple, and α and β were properly derived from all the pre-Fiat-Shamir values of all the circuit instances.

An individual segment description consists of the six public parameters along with claimed vector commitments $(\bar{d}, \bar{a})$ to the pre-Fiat-Shamir and remaining parts of the vector.

A consistent collection of segment descriptions is described by the above six parameters along with a hash H^{FS} committing to pre-Fiat-Shamir data, and a Relaxed Committed R1CS instance $(u, \bar{x}, \bar{e})$.

We recursively define a predicate $\varphi_{\text{fold}}(A, B, \alpha, \beta, \vec{u}_{\text{in}}, \vec{u}_{\text{out}}, H^{\text{fs}}, (u, \bar{x}, \bar{e}))$.

This holds in the single-segment case if there exists $(\bar{d}, \bar{a})$ such that $H^{\text{fs}} = \text{Hash}(\vec{u}_{\text{in}}, \vec{u}_{\text{out}}, \bar{d})$, $u = 1$, $\bar{e} = \bar{0}$ and $\bar{x} = (\bar{d}, \bar{a}, \bar{v})$ and $\vec{v}$ is vector of public parameters $(A, B, \alpha, \beta, \vec{u}_{\text{in}}, \vec{u}_{\text{out}})$.

This holds in the recursive case if there exist

$$(A_1, A_2, B_1, B_2, \vec{u}_{\text{mid}}, H_1^{\text{FS}}, H_2^{\text{FS}}, (u_1, \bar{x}_1, \bar{e}_1), (u_2, \bar{x}_2, \bar{e}_2))$$

and folding proof $\bar{t}$ such that

- $\varphi_{\text{fold}}(A_1, B_1, \alpha, \beta, \vec{u}_{\text{in}}, \vec{u}_{\text{mid}}, H_1^{\text{fs}}, (u_1, \bar{x}_1, \bar{e}_1))$
- $\varphi_{\text{fold}}(A_2, B_2, \alpha, \beta, \vec{u}_{\text{mid}}, \vec{u}_{\text{out}}, H_2^{\text{fs}}, (u_2, \bar{x}_2, \bar{e}_2))$
- $A = A_1 + A_2$
- $B = B_1 + B_2$
- $H^{\text{FS}} = \text{Hash}_{\text{Node}}(H_1^{\text{FS}}, H_2^{\text{FS}})$
- $(u, \bar{x}, \bar{e})$ is the result of folding $(u_1, \bar{x}_1, \bar{e}_1)$ and $(u_2, \bar{x}_2, \bar{e}_2)$ using $\bar{t}$

An honest prover processing a valid block transcript ends up with a folded instance $(u, \bar{x}, \bar{e})$ and values $H^{\mathrm{FS}}, \alpha, \beta, \vec{u}_{\mathrm{out}}$ such that α, β were derived from H^{FS} and $\varphi_{\mathrm{fold}}(0, 0, \alpha, \beta, \vec{0}, \vec{u}_{\mathrm{out}}, H^{\mathrm{FS}}, (u, \bar{x}, \bar{e}))$. Giving a SNARK for knowing a witness to the folded instance and succinct proofs for the other claims constitutes a SNARK for knowing a valid block model transcript.

5.4.8 Estimating the Performance of the Folding Approach

To better understand the potential for performance of the block model, we model the components of the proof generation process. Suppose that we have N computational steps to verify in a proof. Suppose further that the main components of the proof procedure take time which is linear in the size of their inputs: We assume that for a segment of k steps, providing a proof for a relaxed committed R1CS of this size, either as a direct proof or in the form of some other succinct proof, takes $t_{R1CS}k$ time. We further assume that committing to the data in such a segment takes $t_{msm}k$, and that folding 2 such relaxed committed R1CS instances of this size takes $c_{fold} + t_{fold}k$ time. Then the total time to commit and fold N steps in s segments, and then prove the final instance, is

$$t_{msm}s\frac{N}{s} + (s-1)(c_{fold} + t_{fold}\frac{N}{s}) + t_{R1CS}\frac{N}{s}$$

The first term and second half of the second term are constant in s, so we can ignore these when minimizing. Since s is a free parameter, we can minimize the total time by minimizing $s \cdot c_{fold} + (t_{R1CS} - t_{fold})N/s$, which is minimized at

$$s = \sqrt{\frac{(t_{R1CS} - t_{fold})N}{c_{fold}}}$$

The total time to prove is then approximately

$$t_{msm}N + t_{fold}N + 2\sqrt{(t_{R1CS} - t_{fold})Nc_{fold}}$$

For large N, the last term is asymptotically smaller and t_{fold} is expected to be relatively small, so the total time is dominated by the first term.

Investigating concrete times for this model: Looking at our benchmarks, we see that our largest Metamath file with 258135 tokens were processed by the fastest prover in 19.82 seconds. Let us assume each of these tokens corresponds to a proof rule application, and estimate that the block model requires approximately 10 field elements per proof rule:

- 2 Field elements per proof rule to encode the rule's output
- 2-4 Field elements per proof rule to encode the rule's input(s)
- 2 Field elements per proof rule accumulate the output in a product for uniqueness checking

- 2-4 Field elements per proof rule to accumulate the input(s) in a product for subset checking

We can then estimate that the block model would require $2581350 \approx 2^{21}$ field elements for the same proof. Using benchmarks from [25], we see that for Pippenger operating on a typical curve with a single V100 GPU, the time to commit to a vector runs to approximately $90/2^{21} = 0.000043ms$ per field element, which equates to $\approx 110ms$ for a vector of 2581350 size. Considering that our zkVM benchmarks were run on a more powerful machine with 4 NVIDIA GeForce RTX 4090 GPUs, it is reasonable to think that commitments of this size could be handled in $\approx 14ms$ with our setup (note that the clock speed of the 4090 is about twice that of the V100). This is around a 1400× speed-up over the zkVM approach, which is on the same order of magnitude as the 3330× speed-up that ZKSMT found when comparing their custom zkSNARK to a generic approach [26].

5.5 Implementing the Block Model in AIR

In this section we describe how the block model can be implemented in the type of constraints natively supported by STARK and Plonkish-style SNARKs. It was named AIR (Algebraic Intermediate Representation) in the STARK paper [1]. This form of arithmetization is oriented much more towards implementing a virtual machine with multiple registers which update according to particular operations. In this style, the ZK witness will be a rectangular array of field elements. For k columns, the constraints will be specified as polynomials on $2k$ variables which will be evaluated on pairs of successive rows of the transcript and must always evaluate to zero. The polynomials may also take a few additional arguments for Fiat-Shamir parameters. The public data of an instance consists of specified cells of the table, or entire columns.

When implementing the block model on this style of ZK backend each block will map to a rectangular area of the transcript. Additional columns can be used for auxiliary purposes such as accumulations for lookup and permutation arguments. The constraints, acting only on pairs of rows, must enforce that all blocks respect a given set of block rules, in addition to checking the validity of the transcript.

In principle a specific set of block rules might be hard-coded into the set of constraint polynomials, but it seems that would require prohibitively many constraints, and would be less flexible.

Instead we will describe a generic block implementation in a semi-"software" style, where the basic functionality of emitting and demanding claims and routing arguments among claims (and public block arguments) happen in a fairly generic way, controlled by constants in some "setting" columns. The contents of these settings are a kind of "instructions" or "microcode" for the block.

This narrows the task of enforcing allowed block rules to controlling these block settings columns. This is somewhat similar to the handling of the program

text in Harvard-architecture zkVMs, but even simpler because we don't have general purpose instruction sets or even nontrivial control flow.

In this design, the main components are

- Claim and constraint handling
- Argument routing
- Block rule enforcement

We first describe the main layout of the block, especially columns which are involved in more than one component, and then how each component can be implemented.

5.5.1 Common Block Layout

There will be `block_control` column, whose value on the first row of a block is the code for the block rule, and then as many public arguments as the block expects will be recorded in the following rows. If the block layout needs additional rows, these entries of the `block_control` column will be forced to 0.

This is laid out so that the public input column corresponding to the gamma/claim part of a proof can be in the same format as the control column, and a simple constraint copying nonzero values from the public input column into the actual control column will enforce that the full transcript has the required public portion.

There will be a pool of selector columns that are considered part of the block structure. To commit to these values as part of block rule enforcement, they are first combined into a single `compressed_selectors` column with a constraint like `compressed_selectors == s_1 + 2*s_2 + .. + 2^{k-1}*s_k`, and then only the `compressed_selectors` column is included in the block fingerprint.

To delimit the block instances there will be selector columns `block_start` and `block_end`, set to 1 only on the first and last rows of a block.

For the interaction of the claim enforcement and argument routing components, there will a reserved tuple of columns for laying out claims, with columns `claim_code` and `arg1` ... `argk` (for however many arguments the design supports). We could perhaps have several such tuples per row, but one seems sufficient if we have some selectors route whether it is emitted, demanded, or otherwise. We will also implement primitive constraints that don't go through the usual claim mechanism, such as addition or inequality, to expect the arguments to be in these argument columns, because that is what argument routing component can control.

The block "microcode" consists of the block code, which means only the first entry of the control column within the block layout, and all values from the `compressed_selectors` and `claim_code` columns. For the "permutation style" argument routing, the columns holding the "tags" values are also part of the "microcode".

5.5.2 Claim Handling

We will have selectors to control how the claim columns on a row are used, whether that is as an emitted claim, a demanded claim, a unique claim, or not at all. When working with claims of less than maximum arity the remaining columns will just be set to zero.

There will also be selectors to enforce any primitive constraints. A basic set is addition and inequality. The addition selector can simply apply `arg1 == arg2 + arg3`. To enforce `arg1 <> arg2` we can use `arg3` for an advice value and enforce `1 == (arg1 - arg2)*arg3`.

To support block definitions where arbitrary constants are used in arguments we can also have a selector that enforces `arg1 == claim_code`. This takes advantage of the fact that the block rule enforcement already allows fixing arbitrary values for the claim code to avoid introducing a new mechanism. Argument routing then can be used to copy the value to other places.

The standard construction for permutation or lookup arguments over a tuple of columns already relies on encoding the tuple into a single field element using a Fiat-Shamir constant, so there is a natural degree of indirection here.

It might be possible to use selectors to control whether a value is multiplied into a lookup or permutation argument at all without using any auxiliary columns at all, but even if not then we can definitely use a selector to control whether an auxiliary column is set to zero or the encoded tuple value. In this way, the worse case is three auxiliary columns for the three emit/demand/unique roles, rather than needing three copies of the entire claim layout columns.

We might choose to have multiple sets of claim layout columns if we would prefer to have a shorter fatter transcript, but our current performance hypothesis doesn't call for that. If there are multiple sets there could be design choices like only allowing a subset to be used to emit claims.

The extra columns used for the lookup argument are the product accumulation column, and two columns for the permuted witness. The input and output are already accounted for.

5.5.3 Unique Claims

To enforce uniqueness, we will have extra columns for the permuted version of the unique claims, plus one columns for the product accumulator for the permutation constraint.

For our planned applications it seems sufficient to only support unique claims with a single argument, so two columns `ucc`(unique check code) and `uca`(unique check argument) could represent the table.

The dummy element will be `(0,0)`. To check uniqueness we need to enforce the disjunction of the constraints

$$\begin{aligned}
&\texttt{ucc} < \text{next}(\texttt{ucc}) \wedge \text{next}(\texttt{uca}) = 0 \\
&\texttt{ucc} = \text{next}(\texttt{ucc}) \wedge \texttt{uca} < \text{next}(\texttt{uca}) \\
&\texttt{ucc} = \text{next}(\texttt{ucc}) = \texttt{uca} = \text{next}(\texttt{uca}) = 0
\end{aligned}$$

To avoid a general inequality test we can impose stricter assumptions on the code and argument values, which are still compatible with using unique claims for term definitions. The restrictions are that the codes which can appear in unique claims are assigned consecutive values starting at 1, at least one claim for each code appears in the transcript (this can be arranged by defining dummy data if needed), and the set of arguments used with a given claim code are also consecutive (possible for our use because the arguments are just arbitrary names).

In this case, we use $X + 1 = \text{next}(X)$ rather than $X < \text{next}(X)$, allowing the constraints to be expressed with simpler polynomials.

Unique Claim Hack

In term defining blocks, we emit some claims for later use and also check uniqueness. The emitted claims must include a well-formed term claim with depth like `wf_term2(T,k)`, so we can maintain acyclicity over further term definitions. But we need to enforce that the name is also not reused at another depth, so the claim checked for uniqueness must omit the depth, like `wf_term1(T)`.

But it is not necessary to have a single-argument claim available for ordinary lookup because other block definitions could always just demand `wf_term2(T,_)`.

We can cover the unique claim and the ordinary emitted claim with a single row if we reused the same code value `<wf_claim>` with one argument in the uniqueness check, and two in the ordinary lookup. We are already planning to cover fewer columns in the permutation argument for uniqueness (or only one), so just put the argument `k` in column not covered (such as `arg2`), and enable both the "emit" and "unique" selectors on the row.

5.5.4 Block Rule Enforcement

The block rule enforcement is based on taking a polynomial fingerprint of the "microcode", and looking it up in a table of allowed block rules.

There will be a public input column or columns defining the allowed block rules, two columns used, respectively, to accumulate the fingerprint of the actual block instances and of the block definitions, and extra columns implementing a lookup argument. It might be possible to share with the main claim lookup, but that would need an extra security argument that a malformed block could not emit a claim justifying its own definition.

There will be a Fiat-Shamir constant α (or `blockcode_alpha`) used for computing the polynomial fingerprint.

For the block instance accumulation, if `block_fingerprint` is the accumulator column and we only have to put selectors and claim codes in the microcode, we will have the constraint

$$\begin{aligned}\texttt{block\_fingerprint} = {} & \texttt{block\_control} \cdot \alpha^2 \\ & + \texttt{compressed\_selectors} \cdot \alpha \\ & + \texttt{claim\_code}\end{aligned}$$

on rows where `block_start` is 1, and

$$\begin{aligned}\texttt{next(block\_fingerprint)} = \texttt{block\_fingerprint} \cdot \alpha^2 \\ + \texttt{next(compressed\_selectors)} \cdot \alpha \\ + \texttt{next(claim\_code)}\end{aligned}$$

on rows where `next(block_start)` is 0. At the end of the block the accumulated fingerprint will be added into the lookup side of a lookup argument for allowed block rules.

Using powers of α rather than unrelated values to combine the columns allows the block definitions to be given in a different arrangement. The public input column can be completely linearized like.

```
<block_name>
<compressed_selectors>1
<claim_code>1
<compressed_selectors>2
<claim_code>2
...
```

The expected fingerprint for a block can be accumulated over this representation by incorporating one new value per row rather than two, resetting the accumulator at the start of a new block code and contributing the accumulated fingerprint into the table side of the allowed block fingerprint lookup argument at the end of the block definition. It probably requires a selector column to delimit the block definitions.

The set of block rules will not vary, so maybe there are techniques for handling a lookup with a fixed predetermined table that don't need to make the table part of the transcript at all.

5.5.5 Argument Routing

The job of the argument routing functionality is to enforce the equalities that should exist among arguments of different claims, and the public arguments of the block.

As the basic constraints of the zkVM style machine operate between adjacent rows, the natural way to implement this is "register style", with some auxiliary "register" columns that hold copies of values, and selector columns that enable equalities between various argument and register positions, and the control column for loading the argument values.

In this style at a bare minimum there must be enough register columns to hold all the live values which are not in argument (or control) columns on that row.

There are many possible design choices for exactly what routing to support, and also a single selector could enforce a collective operation like shifting values along a set of registers.

Unlike hardware-based intuitions, preserving the value in a register column from one row to the next is just as much an equality constraint as any other. Also, value lifetimes in blocks probably don't look much like register lifetimes in conventional register allocation, unless the block definitions look less like derivation rules and more like

```
foo(X,Y,Z) -: A1==X+Y, A2==A1+Y, A3==A2+A1, Z==A3+A2
```

Yet further, having a zero-one selector column is just as expensive as a register column (the values will not be small after the interpolation and evaluation on a disjoint domain that STARK uses), so it is probably not worth trying to add selectors for fancy operations to try to save registers.

All this suggests starting with a simple register design, and seeing how expensive it is on realistic block sets.

Simple Register Design

In this simple register design, we will just reserve enough register columns that all the arguments that appear in more than one claim of the block will be copied on every row, rather than needing more selectors to modulate what is copied.

Most selectors will just control copies between the argument positions and the register columns on the same row. If we are not restricted to degree 2 constraints, we can select among `k` rows with `lg k` selectors per argument, using terms like `(arg_i - reg_j)*s_a(1-s_b)*s_c`, which enforces that argument `i` equal register `j` only if selector `a` is set, `b` unset, and `c` set.

We also need to reserve some binary combinations for setting the argument to zero and for leaving it unconstrained, so take `k` a bit less than a power of two or have an extra selector.

For loading the block's public arguments into the registers the simple design is to have an additional set of `lg k` selectors. A trick that probably makes manual register allocation excessively confusing reduces that to one selector. Just write the constraints that copy register values into the next row (unless we are at the end of the block) so that the values are copied with a cyclic shift. Then loading arguments only needs a single selector for copying the control column into the first register.

By Permutation

At least as early as PlonK, systems mapping gates of a circuit onto rows or blocks of a table have used permutations to enforce non-local equalities. The idea is to define a permutation over the cell locations whose cycles are the desired equivalence classes, and check that the list of cell values is unchanged by applying the permutation.

A specific permutation can be applied using a multi-column (arbitrary) permutation constraint, by fixing the values in extra columns of "tags" on each side of the permutation. Often values are chosen so a simple constraint like next(X) = X+1 or next(X) = wX suffices to fix the output tag column.

To apply this for claim arguments, we will cost two index columns per argument column, and a column for the permutation accumulator. The permutation is block local, so there will also need to be a column holding a block ID (this might have other applications).

We may need to consider each tag an independent value in the block fingerprint. If the indices could be constrained to small values we could use a constraint like `compressed_indices == index1 + 16*index2 + 16^2*index3` to summarize multiple 0..15 values in a single field, but constraining the values would require either a range check or an excessively high-degree constraint like `(index1-0)*(index-1)*..*(index1-15)`.

5.5.6 Example Block Tabulations

This section shows how some block rules from the propositional logic example in Section 5.3 would be laid out in an AIR transcript according to the preceding implementation strategy.

We allocate columns for relations with up to three arguments, the minimum to support the widest relations in the example such as `instantiation` and `subst_lookup`. We use the simple register model, and write the tabulation of each block rule using only as many register columns as that example needs, in a fully concrete implementation this would leave unused columns. We assume that the selectors for populating arguments allow setting the argument to a fixed value 1, a fixed value 0, or leaving it unconstrained, in addition to copying any register.

Nine register columns suffice for all blocks. The propositional axioms are the most complicated, but other rules would still require seven register columns even if the propositional axiom rules were removed (forcing the axioms to be asserted in the public part of a transcript instead).

```
assert(T,k): SETUP
  proved(T,k) -: wf_term(T,_)
```

Control	R1	R2	Claim	Arg1	Arg2	Selectors
`<assert>`	`T`	`k`	`<proved>`	`T`	`k`	Block_Start, Setup, Arg1=R1, Arg2=R2, Emit
`T`	`T`	`k`	`<wf_term>`	`T`	X	Control=R1, Demand, Arg1=R1, Arg2=Free
`k`	`T`	`k`	0	0	0	Block_End, Control=R2

The value X corresponds to the `_` in the block definition. It does not have to equal any other claim arguments, so it is not in any registers, but to construct a valid witness the prover will have to choose a depth that lets the claim lookup succeed.

In `modus_ponens` we see a possible elaboration of depth checking.

```
modus_ponens(T):
  proved(TB,k) -:
    is_impl(T,TA,TB), proved(T,k1), proved(TA,k2),
    k == 1+max(k1,k2)
```

Control	R1-R6	Claim	Arg1-3	Selectors
`modus_ponens`	T, TA, TB, k, k1, k2	`<proved>`	TB, k, 0	Block_Start, Arg1=R3, Arg2=R4, Emit
`T`	T, TA, TB, k, k1, k2	`<is_impl>`	T, TA, TB	Arg1=R1, Arg2=R2, Arg3=R3, Demand
	T, TA, TB, k, k1, k2	`<proved>`	T, k1, 0	Arg1=R1, Arg2=R5, Demand
	T, TA, TB, k, k1, k2	`<proved>`	TA, k2, 0	Arg1=R2, Arg2=R6, Demand
	T, TA, TB, k, k1, k2	`<depthLt>`	k1, k, 0	Arg1=R5, Arg2=R4, Demand
	T, TA, TB, k, k1, k2	`<depthLt>`	k2, k, 0	Block_End, Arg1=R6, Arg2=R4, Demand

The `subst_not_here` block demonstrates the primitive inequality constraint.

```
subst_not_here(V,S):
  not_in_subst(V,S)
  -: is_single_subst(S,V1,T),
     V != V1.
```

Inequality takes an advice value. X should be set to `1/(V-V1)` (the constraint checks `1 == (Arg1-Arg2)*Arg3`).

Control	R1-R3	Claim	Arg1-3	Selectors
`subst_not_here`	V, S, V1	`not_in_subst`	V, S, 0	Block_Start, Arg1=R1, Arg2=R2, Emit
V	V, S, V1	`is_single_subst`	S, V1, T	Arg1=R2, Arg2=R3, Arg3=Free, Demand
S	V, S, V1	0	V, V1, X	Block_End, Arg1=R1, Arg2=R3, Arg3=Free, NotEqual

The `def_term_impl` block demonstrates unique claims.

```
def_term_impl(T,TA,TB):
  is_impl(T,TA,TB),
  UNIQUE wf_term(T), wf_term(T,k)
  -: wf_term(TA,ka), wf_term(TB,kb),
     k == 1+max{ka,kb}.
```

Control	R1-R6	Claim	Arg1-3	Selectors
`def_term_impl`	T, TA, TB, k, ka, kb	`<is_impl>`	T, TA, TB	Block_Start, Arg1=R1, Arg2=R2, Arg3=R3 Emit
`T`	T, TA, TB, k, ka, kb	`<wf_term>`	T, k, 0	Control=R1, Arg1=R1, Arg2=R4, Emit, Unique
`TA`	T, TA, TB, k, ka, kb	`<wf_term>`	TA, ka, 0	Control=R2, Arg1=R2, Arg2=R5, Demand
`TB`	T, TA, TB, k, ka, kb	`<wf_term>`	TB, kb, 0	Control=R3, Arg1=R3, Arg2=R6, Demand
	T, TA, TB, k, ka, kb	`<depthLt>`	ka, k, 0	Arg1=R5, Arg2=R4, Demand
	T, TA, TB, k, ka, kb	`<depthLt>`	ka, k, 0	Block_End, Arg1=R6, Arg2=R4, Demand

The `smallPosNext` block uses addition and loading 1. Alternatively the machine's register design might allow loading 1+Reg1 (or 1+Regi for a limited set of registers). If that was an additional option for every register it would obviously cost an extra selector, but picking among nine registers with binary selectors requires four bits and we haven't used all 16 codes.

```
smallPosNext(k):
  smallPos(k) -: smallPos(k0), k == k0 + 1
```

Control	R1,R2	Claim	Arg1-Arg3	Selectors
`smallPosNext`	k, k0	`<smallPos>`	k, 0, 0	Block_Start, Arg1=R1, Emit
`k`	k, k0	`<smallPos>`	k0, 0, 0	Control=R1, Arg1=R2, Demand
	k, k0	0	k, k0, 1	Block_End, Arg1=R1, Arg2=R2, Arg3=1, Addition

5.6 Related Work on Proof Checking in ZK

In this section, we recall three recent proof checking approaches and compare them with our block model proposed solution.

ZKSMT [26] is a system for creating ZK proofs of Satisfiability Modulo Theories (SMT) validity proofs. It is based on existing zkVM design, but specialized for checking proofs made up of a collection of proof steps instead of executing programs made up of sequences of instructions, which enables performance optimizations that would not be available in a general-purpose zkVM. This includes storing the list of referenced expressions and the list of proof steps in read-only memory tables, which enables a more efficient memory consistency check.

zkPi [23] is a system for creating succinct zero-knowledge proofs of Lean proofs. zkPi works somewhat similarly to ZKSMT, but supports a more expressive proof system based on dependent type theory.

Both of these systems make multiple ROM accesses per proof step, because it is necessary to load, e.g., premises of a proof step to check that the step is well-formed. This is limiting, and in the case of zkPi, prevents the system from completing 54% of one set of benchmark proofs. In addition, these systems require more memory to generate the ZK proof the longer the math proof is. In our design we greatly reduce maximum memory usage by splitting proofs into segments and then folding these segments together as part of the final ZK proof.

The ZKSMT paper included results of an experiment which is roughly analogous to our experiment with implementing Metamath proof checking in various off-the-shelf zkVMs. As a baseline to compare to ZKSMT, they implemented the same algorithm in C++, converted it to ZK circuit using Cheesecloth [10], and used a system called Diet Mac'N'Cheese as the ZK backend, which, like ZKSMT, uses a VOLE-based ZK system. They found an about 3000x speedup for ZKSMT over this zkVM-like setup. This suggests that specialized ZK machinery for proof checking can be much more efficient than going through a zkVM.

In the direction of theories with less power, Luo et al. [27] construct a system for proving unsatisfiability (UNSAT) in a zero-knowledge system. This system, which is a basic NP-Complete language, does not rise to the level of a fully featured formal system. Nevertheless, the ZK proof system constructed for it has some features in common with the above. For example, it also identifies a collection of possible proof steps, in this case, the logic of resolution proofs.

6 Conclusion

Proof of Proof is a novel correctness approach proposed and implemented by Pi Squared, which combines formal mathematical proofs and cryptography in a unique way. The main idea, which also inspired its name, is to produce a *ZK Proof* of a *Math Proof*. The ZK Proof acts as a succinct and independently verifiable certificate that attests for the integrity of the Math Proof, and thus for the Claim, or the Math Theorem, that was proved by the Math Proof. Since

every Claim that is provably true can (and should) be organized and presented as a Math Theorem with a Math Proof, Proof of Proof is therefore meant to serve as the ultimate approach to verifiable truth.

In this paper, we presented an instance of the general Proof of Proof approach, where the truths are program execution claims: the ZK proof represents a Math Proof that certifies that a given program executed correctly according to its programming language's or virtual machine's formal semantics. Specifically, the presented Proof of Proof instance is based on the following key components:

- $\mathbb{K}$– a *universal framework for programming languages*, where the formal semantics of a programming language is defined as a Matching Logic theory and an Interpreter for that language can be automatically derived;
- *Math Proof Generation (MPG)*, a mechanism that generates Math Proofs for program executions, based on traces produced by the Interpreter;
- *Proof Checker*, a simple and small program which ensures that the Math Proofs (and thus also $\mathbb{K}$ and the Interpreter) are *not trusted but verified*, providing the strongest correctness argument, with the smallest trust base ever reported or recorded, for the Math Proofs derived from computations;
- *ZK Proof Generation*, which utilizes Zero-Knowledge (ZK) proof technology to reduce the size of the mathematical proofs, showing only that such mathematical proofs exist, which is sufficient for many/most applications.

To summarize, the Proof of Proof instance presented in this paper is therefore:

- ✓ **Universal**: There is a single language for the math proofs, which works with all programs written in all programming or virtual machine languages.
- ✓ **Verifiable**: The generated math proofs are verified by a simple proof checker.
- ✓ **Correct-by-construction**: The ZK proof generator acts as a massive compressor of math proofs in general, and is not specific to any programming or virtual machine language. So we can "plug and play" new languages without any need to trust or formally verify anything language-specific.

We believe that Proof of Proof will naturally find applications in at least three domains: remote computing, blockchain, and AI.

Indeed, remote servers will be able to execute programs for their clients in any existing or future programming languages, and produce ZK proofs of correct execution along with the result produced by the program. The clients verify the ZK proof and thus know that the result was correctly computed. Once Proof of Proof matures as a technology, clients and remote computing providers will have no reason nor incentive to do things any other way. Blockchains will not need to replicate program execution in thousands or even millions of nodes only

because some of them may cheat on what the computation does. Finally, AI model inference will be verifiable as well, for our peace of mind.

Perhaps some of the most interesting applications of Proof of Proof will be those that go beyond what verifiable computing alone can do. Specifically, applications in which we combine various methods to obtain mathematical proofs. For example, a formally verified program provides mathematical guarantees that certain properties of interest hold. Those properties can then be taken into account (as lemmas) to produce smaller and/or more comprehensive proofs of correct execution, and thus faster and more trusted verifiable computing end-to-end. We experimented with this approach with a formally verified variant of the square root function in Uniswap, reducing its ZKP generation and verification to constant time — indeed, the math proof resulting from the formal verification of `sqrt` that it implements the square root function is done once and for all, and then reused as a lemma when computing, eg, `sqrt(100) = 10`.

Even more interesting, the program that produced the computation is not even required to be public, provided that it is formally verified for compliance with publicly available specifications or requirements. Indeed, from the math proof produced by the formal verification of the program and the math proof produced by the execution of the program, we can construct a math proof of the claim that "there exists a program that is compliant, whose execution produced this result". This is sufficient in many / most cases for users, but it can be the difference between all or nothing for companies which are willing to prove compliance but cannot make their code public.

References

[1] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, page 46, 2018.

[2] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS '13, page 401–414, New York, NY, USA, 2013. Association for Computing Machinery.

[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] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. Generating proof certificates for a language-agnostic deductive program verifier. In *Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV'21)*, Virtual, July 2021. ACM.

[5] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120:1–36, 2021.

[6] 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.

[7] Xiaohong Chen and Grigore Roşu. Matching μ-logic. Technical report, University of Illinois Urbana-Champaign, 2019.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. *Maude manual*. SRI International, 2023.

[9] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore 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)*, pages 74–91. ACM, 2016.

[10] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-knowledge proofs of real-world vulnerabilities, 2023.

[11] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 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.

[12] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, August 2013.

[13] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.

[14] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 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, Oregon, USA, 2015. ACM.

[15] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 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.

[16] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[17] Pi Squared Inc. Semantics-based execution and the llvm backend. Unpublished manuscript, 2025.

[18] Runtime Verification Inc. K user manual. `https://kframework.org/docs/user_manual`.

[19] Shuanglong Kan, David Sanan, Shang-Wei Lin, and Yang Liu. K-Rust: An executable formal semantics for Rust, 2018.

[20] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. Language-parametric compiler validation with application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 1004–1019, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. *Dodis, Y., Shrimpton, T. (eds) Advances in Cryptology – CRYPTO 2022. CRYPTO 2022. Lecture Notes in Computer Science, Springer, Cham*, pages 359—-388, 2022.

[22] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[23] Evan Laufer, Alex Ozdemir, and Dan Boneh. zkPi: Proving lean theorems in zero-knowledge. Cryptology ePrint Archive, Paper 2024/267, 2024.

[24] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. Towards a trustworthy semantics-based language framework via proof generation. In *Proceedings of OOPSLA*, volume 7. ACM, April 2023.

[25] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs. Cryptology ePrint Archive, Paper 2022/1321, 2022.

[26] Daniel Luick, John Kolesar, Timos Antonopoulos, William R. Harris, James Parker, Ruzica Piskac, Eran Tromer, Xiao Wang, and Ning Luo. ZKSMT: A VM for proving SMT theorems in zero knowledge. Cryptology ePrint Archive, Paper 2023/1762, 2023.

[27] Ning Luo, Timos Antonopoulos, William Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. Cryptology ePrint Archive, Paper 2022/206, 2022.

[28] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, September 2008. Association for Computing Machinery.

[29] Norman D. Megill and David A. Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu Press, Morrisville, North Carolina, USA, 2019.

[30] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 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, Oregon, USA, 2015. ACM.

[31] Pi Squared Inc. The Pi Squared whitepaper. `https://pi2.network/papers/pi2-whitepaper`, 2025.

[32] Proof trace format. `https://github.com/runtimeverification/llvm-backend/blob/master/docs/proof-trace.md`, 2024.

[33] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4):1–61, 2017.

[34] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[35] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. *Advances in Cryptology – EUROCRYPT 2024, Lecture Notes in Computer Science, Springer, Cham*, 14656:180–209, 2024.

[36] Joseph R. Shoenfield. *Mathematical logic*. Addison-Wesley Pub. Co, 1967.

[37] The Kore language. `https://github.com/kframework/kore`, 2023.

[38] The metamath specification of matching logic. `https://github.com/runtimeverification/proof-generation/blob/main/theory`, 2023.

[39] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. KRust: A formal executable semantics of Rust. In *Proceedings of the 12th International Symposium on Theoretical Aspects of Software Engineering (TASE'18)*, pages 44–51, Guangzhou, China, August 2018. IEEE.