Matching Logic Proofs Meet Succinct Cryptographic Proofs

Abstract

We first present the syntax and proof system of matching logic. Then, we introduce the research problem of producing embarrassingly parallel cryptographic proofs for matching logic.

Contents

1 Matching Logic

We present matching logic syntax in Section 1.1 and its proof system in Section 1.2. We introduce the matching logic proof checker in Section 1.3.

1.1 Matching Logic Syntax

We fix countably infinite sets EV and SV, where

1. EV is a set of *element variables*, often denoted x, y, and z.
2. SV is a set of *set variables*, often denoted X, Y, and Z.

We assume that EV and SV are disjoint, i.e., $EV \cap SV = \emptyset$. A *variable*, often denoted u and v, is either an element variable or a set variable. We write $u \equiv v$ iff u and v denote the same variable. We write $u \not\equiv v$ iff u and v denote two distinct variables.

Definition 1. A *signature* Σ is a set whose elements are called *symbols*, often denoted σ. The set of Σ-*patterns*, or simply *patterns* when Σ is understood, is inductively defined by the following grammar:

$$\varphi ::= x \qquad \text{// Element Variable}$$

$$
\begin{array}{ll}
\mid X & \text{// Set Variable} \\
\mid \sigma & \text{// Symbol} \\
\mid \varphi_1\, \varphi_2 & \text{// Application} \\
\mid \varphi_1 \to \varphi_2 & \text{// Implication} \\
\mid \exists x \,.\, \varphi & \text{// Existential Quantification} \\
\mid \mu X \,.\, \varphi & \text{// Least Fixpoint}
\end{array}
$$

where $\mu X \,.\, \varphi$ requires that φ is positive in X, which means that X is not nested in an odd number of times on the left-hand side of an implication.

The following are some well-formed least fixpoint patterns:

1. $\mu X \,.\, X$
2. $\mu X \mu Y \,.\, X$
3. $\mu X \,.\, (Y \to \bot)$
4. $\mu X \,.\, ((X \to \bot) \to \bot)$
5. $\mu X \,.\, \exists y \,.\, ((X \to y) \to (y \to X))$
6. $\mu X \,.\, (\sigma\, (X \to \bot)) \to \bot$

The following are some ill-formed least fixpoint patterns:

1. $\mu X \,.\, (X \to \bot)$
2. $\mu X \,.\, (X \to X)$
3. $\mu X \,.\, (\sigma\, X) \to \bot$

Application is left-associative; for example, $\varphi_1\, \varphi_2\, \varphi_3$ means $(\varphi_1\, \varphi_2)\, \varphi_3$. Implication is right-associative; for example, $\varphi_1 \to \varphi_2 \to \varphi_3$ means $\varphi_1 \to (\varphi_2 \to \varphi_3)$. Application has a higher precedence than implication; for example, $\varphi_1 \to \varphi_2\, \varphi_3$ means $\varphi_1 \to (\varphi_2\, \varphi_3)$. The scope of $\exists x$ and μX goes as far to right as possible, for example, $\exists x \,.\, \varphi_1 \to \varphi_2$ means $\exists x \,.\, (\varphi_1 \to \varphi_2)$, and not $(\exists x \,.\, \varphi_1) \to \varphi_2$.

Definition 1 includes 7 *primitive constructs* of matching logic. Besides these primitive constructs, we also allow *notations* (also called abbreviations, short-hands, derived constructs, or syntactic sugar), which are constructs that can be defined using the primitive ones. The following are some commonly used notations:

$$
\begin{array}{rll}
\top \equiv \exists x \,.\, x & \text{// Logical True} & (1) \\
\bot \equiv \mu X \,.\, X & \text{// Logical False} & (2) \\
\neg\varphi \equiv \varphi \to \bot & \text{// Negation} & (3) \\
\varphi_1 \vee \varphi_2 \equiv (\neg\varphi_1) \to \varphi_2 & \text{// Disjunction} & (4) \\
\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) & \text{// Conjunction} & (5) \\
\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) & \text{// ``If-and-Only-If''} & (6) \\
\forall x \,.\, \varphi \equiv \neg\exists x \,.\, \neg\varphi & \text{// Universal Quantification} & (7) \\
\nu X \,.\, \varphi \equiv \neg\mu X \,.\, \neg\varphi[\neg X/X] & \text{// Greatest Fixpoint} & (8)
\end{array}
$$

where $\varphi[\neg X/X]$ is the pattern obtained by substituting $\neg X$ for X in φ. We formally define substitution in Definition 5.

Among the primitive constructs, $\exists x$ and μX are *binders*. We define the concepts that are important to binders: *free variables*, α*-renaming*, and *capture-avoiding substitution*. These definitions are standard but we still present all the technical details to make this article self-contained. Readers who are already familiar with these concepts can skip the rest and jump to Section 1.2.

Definition 2. Given a pattern φ, we inductively define its set of *free variables*, denoted $FV(\varphi)$, as follows:

1. $FV(x) = \{x\}$
2. $FV(X) = \{X\}$
3. $FV(\sigma) = \emptyset$
4. $FV(\varphi_1\, \varphi_2) = FV(\varphi_1) \cup FV(\varphi_2)$
5. $FV(\varphi_1 \to \varphi_2) = FV(\varphi_1) \cup FV(\varphi_2)$
6. $FV(\exists x\,.\,\varphi) = FV(\varphi) \setminus \{x\}$
7. $FV(\mu X\,.\,\varphi) = FV(\varphi) \setminus \{X\}$

We say that φ is *closed* if $FV(\varphi) = \emptyset$.

We can categorize the occurrences of a variable in a pattern into *free* and *bound* occurrences, based on whether the occurrence is within the scope of a corresponding binder.

Definition 3. For any x, X, and φ,

1. A *free occurrence* of x in φ is an occurrence that is not within the scope of any $\exists x$ binder;
2. A *free occurrence* of X in φ is an occurrence that is not within the scope of any μX binder;
3. A *bound occurrence* of x is an occurrence within the scope of an $\exists x$ binder;
4. A *bound occurrence* of X is an occurrence within the scope of a μX binder.

We say that a variable is *bound* in a pattern if all occurrences are bound. We say that a variable is *fresh* w.r.t. a pattern if there are no occurrences, free or bound, of the variable.

As an example, consider the pattern $x \wedge \exists x \exists y\,.\,x \wedge y$ where $x \not\equiv y$. This pattern has two occurrences of x and one occurrence of y.[1] The left-most x is a free occurrence. The right-most x and the right-most y are both bound occurrences.

We can rename a bound variable to a fresh variable. Such operation is called *α-renaming*. For example, the above example pattern $x \wedge \exists x \exists y\,.\,x \wedge y$ can be α-renamed into $x \wedge \exists z \exists y\,.\,z \wedge y$, where $z \not\equiv x$ and $z \not\equiv y$.

Definition 4. Let φ_1 and φ_2 be patterns. We say that φ_1 and φ_2 are *α-equivalent*, written $\varphi_1 \equiv_\alpha \varphi_2$, iff φ_1 can be transformed into φ_2 after applying a finite number of α-renamings.

It is straightforward to show that α-equivalence is indeed an equivalence relation. In fact, α-equivalence is the reflexive, commutative, and transitive closure of α-renaming.

Sometimes, it takes more than one α-renaming to transform a pattern into an α-equivalent one. For example, $x \wedge \exists x \exists y\,.\,x \wedge y$ and $x \wedge \exists y \exists x\,.\,y \wedge x$ are α-equivalent, but it takes three α-renamings to transform the former into the latter:

$$x \wedge \exists x \exists y\,.\,x \wedge y \xrightarrow{\alpha\text{-renaming}} x \wedge \exists z \exists y\,.\,z \wedge y \xrightarrow{\alpha\text{-renaming}} x \wedge \exists z \exists x\,.\,z \wedge x \xrightarrow{\alpha\text{-renaming}} x \wedge \exists y \exists x\,.\,y \wedge x \qquad (9)$$

In the above z is fresh; that is, $z \not\equiv x$ and $z \not\equiv y$.

Note that a notation can be a binder. For example, $\forall x$, which is a notation defined by $\forall x\,.\,\varphi \equiv \neg \exists x\,.\,\neg \varphi$, is a binder. Therefore, all occurrences of x in $\forall x\,.\,\varphi$ are bound, and $FV(\forall x\,.\,\varphi) = FV(\varphi) \setminus \{x\}$.

Next, we define substitution. Because we have binders, we need to be careful to avoid the infamous phenomenon of free variable capture, illustrated by the following example. Suppose $\varphi \equiv \exists x\,.\,x \to y$ where

[1] Note that we do not count $\exists x$ and $\exists y$ as occurrences, although some works do count them and call them *binding* occurrences. In this article, we only consider free and bound occurrences and regard $\exists x$ and μX as syntactic constructs.

$x \not\equiv y$ and we want to substitute y for x in φ. If we do it blindly by replacing every (free) occurrence of y with x, we get $\exists x \,.\, x \to x$. This is unintended because y, which is previously not within the scope of $\exists x$, is now bound. The correct way to do substitution is to first α-rename $\exists x \,.\, x \to y$ into $\exists z \,.\, z \to y$, for some fresh variable z. Then, we apply the substitution and obtain the correct result $\exists z \,.\, z \to x$. Note that this time, x is no longer bound, which is intended.

The operation described above is called *capture-avoiding substitution*. As we can see from the above example, capture-avoiding substitution needs α-renaming to avoid free variable capture. In the literature, there are two standard approaches to define capture-avoiding substitution:

1. We can define capture-avoiding substitution as a partial function that is undefined in the case of free variable capture. In this approach, α-renaming happens explicitly to avoid free variable capture.

2. We can define capture-avoiding substitution as a total function on α-equivalence classes. In this approach, α-renaming happens implicitly to avoid free variable capture.

In this article, we take the second approach; that is, we always work with α-equivalence classes throughout the article.

Definition 5. Let φ and ψ be patterns and v and u be variables. We define $\varphi[\psi/v]$ to be the result of substituting ψ for v in φ as follows:

1. $v[\psi/v] = \psi$
2. $u[\psi/v] = u$ if $u \not\equiv v$
3. $\sigma[\psi/v] = \sigma$
4. $(\varphi_1\, \varphi_2)[\psi/v] = (\varphi_1[\psi/v])\, (\varphi_2[\psi/v])$
5. $(\varphi_1 \to \varphi_2)[\psi/v] = \varphi[\psi/v] \to \varphi_2[\psi/v]$
6. $(\exists x \,.\, \varphi)[\psi/x] = \exists x \,.\, \varphi$
7. $(\exists x \,.\, \varphi)[\psi/v] = \exists y \,.\, \varphi[y/x][\psi/v]$, if $v \not\equiv x$ and y is fresh
8. $(\mu X \,.\, \varphi)[\psi/X] = \mu X \,.\, \varphi$
9. $(\mu X \,.\, \varphi)[\psi/v] = \mu Y \,.\, \varphi[Y/X][\psi/v]$, if $v \not\equiv X$ and Y is fresh

Before we move on, we have a remark on the presentation of binders. Binders and their binding behaviors are known to be a non-trivial matter in the study of formal logic and its syntax. Our presentation in this article is a classic textbook presentation where variables are represented by their names. A *nameless* presentation such as De Bruijn index (`https://en.wikipedia.org/wiki/De_Bruijn_index`) is an encoding method that represents bound variables by numbers instead of their names. These numbers, called De Bruijn indices, denote the number of (nest) binders on top of a bound variable. De Bruijn index has the advantage that α-equivalent terms have the same encoding so checking α-equivalence is the same as checking syntactic equality. Locally nameless approach (`https://charguéraud.org/research/2009/ln/main.pdf`) uses De Bruijn indices only for bound variables and retains names for free variables for readability. Nominal approach (`https://www.sciencedirect.com/science/article/pii/S089054010300138X`) uses *swapping* and *freshness* as the primitives notions/operations regarding binders, instead of using α-equivalence and capture-avoiding substitution as the primitive. The reason is that swapping and freshness have simpler definitions than α-equivalence and (especially) capture-avoiding substitution. Higher-order abstract syntax (HOAS) studies data structures that explicit expose the relationship between variables and their corresponding binders. There is extensive study on matching and unification algorithms modulo α-equivalence in the literature on nominal and HOAS approaches.

We are open to all these approaches regarding binders.

(Łukasiewicz)	$((\varphi_1 \to \varphi_2) \to \varphi_3) \to ((\varphi_3 \to \varphi_1) \to (\varphi_4 \to \varphi_1))$	axiom schema
(Modus Ponens)	$\dfrac{\varphi_1 \to \varphi_2 \quad \varphi_1}{\varphi_2}$	rule of inference with 2 premises
(Quantifier)	$\varphi[y/x] \to \exists x \,.\, \varphi$	axiom schema
(Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x \,.\, \varphi_1) \to \varphi_2}$ if $x \notin FV(\varphi_2)$	rule of inference with 1 premise
(Propagation$_\vee$)	$C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]$	axiom schema
(Propagation$_\exists$)	$C[\exists x \,.\, \varphi] \to \exists x \,.\, C[\varphi]$ if $x \notin FV(C[\exists x \,.\, \varphi])$	axiom schema
(Frame)	$\dfrac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}$	rule of inference with 1 premise
(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$	rule of inference with 1 premise
(Pre-Fixpoint)	$\varphi[\mu X \,.\, \varphi/X] \to \mu X \,.\, \varphi$	axiom schema
(Knaster Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X \,.\, \varphi) \to \psi}$	rule of inference with 1 premise
(Existence)	$\exists x \,.\, x$	axiom schema
(Singleton Variable)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$	axiom schema

Figure 1: Matching Logic Proof System

1.2 Matching Logic Proof System

We present the entire matching logic proof system in Figure 1. To understand the proof system, we first need the following definition.

Definition 6. Let $\Box$ be a distinguished element variable. An *application context* is a pattern where $\Box$ has exactly one occurrence, which is under a number of nested applications. More formally,

1. $\Box$ is an application context, called the *identity context*;
2. if C is an application context and φ is a pattern with no occurrences of $\Box$, both $(C\,\varphi)$ and $(\varphi\,C)$ are application contexts.

For an application context C and a pattern ψ, we write $C[\psi]$ to mean $C[\psi/\Box]$.

An *axiom schema* (or *schema*) is a pattern with the occurrence of *metavariables*, such as x, X, C, and φ. For example, (Łukasiewicz) is a schema with four metavariables φ_1, φ_2, φ_3, and φ_4.

$$\text{(Łukasiewicz)} \quad ((\varphi_1 \to \varphi_2) \to \varphi_3) \to ((\varphi_3 \to \varphi_1) \to (\varphi_4 \to \varphi_1))$$

An *instance* of a schema is the pattern obtained by instantiating all the metavariables in the said schema. The following are all instances of (Łukasiewicz).

$$((\bot \to \top) \to \mathsf{zero}) \to ((\mathsf{zero} \to \bot) \to ((\mathsf{succ}\,\mathsf{zero}) \to \bot))$$

$$((\bot \to \top) \to (\mathsf{zero} \to \bot)) \to (((\mathsf{zero} \to \bot) \to \bot) \to (\bot \to \bot))$$

where $\mathsf{zero}, \mathsf{succ} \in \Sigma$ are two (concrete) symbols in the signature. (Quantifier) is a schema with metavariables x, y, and φ.

$$\text{(Quantifier)} \quad \varphi[y/x] \to \exists x \,.\, \varphi$$

The following are all instances of (Quantifier).

$$(\mathsf{y} \wedge \mathsf{y}) \to \exists \mathsf{x} \,.\, (\mathsf{x} \wedge \mathsf{y})$$
$$(\mathsf{x} \wedge \mathsf{y}) \to \exists \mathsf{x} \,.\, (\mathsf{x} \wedge \mathsf{y})$$

where x and y are concrete (non-meta) element variables.

A *rule of inference (schema)* is a pair of some *premises* and one *conclusion*, all possibly sharing some metavariables. Rules of inference are written as shown below:

$$\frac{\varphi_1 \quad \cdots \quad \varphi_n}{\psi}$$

We write the premises above the line and the conclusion below the line. An *instance* of a rule of inference is obtained by instantiating all the metavariables in the rule of inference.

The matching logic proof system as shown in Figure 1 has 7 axiom schemas and 5 rules of inference. We call axiom schemas and rules of inference *proof rules*. Matching logic has $7 + 5 = 12$ proof rules.

Definition 7. A *theory* is a pair (Σ, Γ) where Σ is a signature and Γ is a set of Σ-patterns, called *non-logical axioms*. When Σ can be understood from the context, we abbreviate (Σ, Γ) as Γ.

Definition 8. Let Γ be a theory. A *Hilbert Γ-proof (for φ_n)* is a finite sequence of patterns:

$$\langle \varphi_1, \varphi_2, \ldots, \varphi_n \rangle \tag{10}$$

where $n \geq 1$, and for every $1 \leq i \leq n$, one of the following holds:

1. $\varphi_i \in \Gamma$;
2. φ_i is an instance of one of the 7 axiom schemas;
3. φ_i can be derived using one of the 5 rules of inference from previous derived patterns (i.e., those in $\{\varphi_1, \ldots, \varphi_{i-1}\}$) as premises.

We write $\Gamma \vdash \varphi$, meaning that φ is provable/derivable from Γ, iff there exists a Hilbert Γ-proof for φ.

Note that a Hilbert Γ-proof for φ can always be re-arranged such that it starts with the axioms in Γ, followed by logical axioms and conclusions of applying the rules of inference. More formally, if φ has a Hilbert Γ-proof, i.e., $\Gamma \vdash \varphi$, then there must exist a Hilbert Γ-proof

$$\Big\langle \underbrace{\varphi_1, \varphi_2, \ldots, \varphi_k}_{\text{axioms in } \Gamma}, \underbrace{\varphi_{k+1}, \ldots, \varphi_l}_{\text{logical axioms}}, \underbrace{\varphi_{l+1}, \ldots, \varphi_n}_{\text{conclusions}} \Big\rangle \tag{11}$$

such that $\varphi_1, \ldots, \varphi_k \in \Gamma$, $\varphi_{k+1}, \ldots, \varphi_l$ are logical axioms, $\varphi_{l+1}, \ldots, \varphi_n$ are conclusions of the rules of inferences, and $\varphi_n \equiv \varphi$. In this proof, $\varphi_1, \ldots, \varphi_k$ are from Γ so they are public. The last pattern φ_n is the theorem being proved so it is also public. The intermediate patterns $\varphi_{k+1}, \ldots, \varphi_{n-1}$ are private or hidden.

As a concluding remark, we mention that the precise presentation of the proof system can be (slightly) modified, without changing the provability relation $\Gamma \vdash \varphi$. For example, we can replace the (Łukasiewicz) axiom schema

$$\text{(Łukasiewicz)} \quad ((\varphi_1 \to \varphi_2) \to \varphi_3) \to ((\varphi_3 \to \varphi_1) \to (\varphi_4 \to \varphi_1))$$

with a single axiom (i.e., not a schema):

$$(\text{Łukasiewicz'}) \quad ((\mathsf{X}_1 \to \mathsf{X}_2) \to \mathsf{X}_3) \to ((\mathsf{X}_3 \to \mathsf{X}_1) \to (\mathsf{X}_4 \to \mathsf{X}_1))$$

where $\mathsf{X}_1, \ldots, \mathsf{X}_4$ are four (different) concrete set variables. Any instance of (Łukasiewicz) is derivable from (Łukasiewicz') and the (Substitution) rule of inference. As another example, we can replace (Frame) with two rules of inference

$$(\text{Frame}_{\text{left}}) \quad \frac{\varphi_1 \to \varphi_2}{(\varphi_1\, \psi) \to (\varphi_2\, \psi)}$$

$$(\text{Frame}_{\text{right}}) \quad \frac{\varphi_1 \to \varphi_2}{(\psi\, \varphi_1) \to (\psi\, \varphi_2)}$$

While now we have two rules instead of one, we avoid the use of application contexts (i.e., $C[\ldots]$), which will make proof checking easier. However, we will get longer proofs, because every (Frame) step will need a number of ($\text{Frame}_{\text{left}}$)/($\text{Frame}_{\text{left}}$) steps, depending on the number of symbols in C.

To conclude, there is some flexibility in the design of the matching logic proof system. The main criteria here is the simplicity of the generation and verification of the succinct cryptographic proofs of matching logic proofs, as discussed in Section 2.

1.3 Matching Logic Proof Checker

Generally speaking, a matching logic proof checker is a deterministic and terminating program

$$\texttt{pc}(\Gamma, \varphi, \Pi) \in \{\texttt{success}, \texttt{failure}\} \tag{12}$$

The three inputs are:

1. a theory Γ, which can be the axiomatization of a mathematical domain or the formal semantics of a programming or specification language;
2. a pattern/claim φ;
3. a proof object Π, which encodes a Hilbert Γ-proof for φ.

A matching logic proof checker should satisfy the following soundness and completeness property:[2]

$$\Gamma \vdash \varphi \quad \text{iff} \quad \text{there exists } \Pi \text{ such that } \texttt{pc}(\Gamma, \varphi, \Pi) = \texttt{success} \tag{13}$$

To implement a matching logic proof checker, one needs to consider many factors. For example, one should decide an encoding/decoding mechanism for representing Γ, φ, and Π. Sometimes, the proof object Π can include additional proof annotations to simplify the main proof checking algorithm. One should also formalize the metalevel operations/definitions in the proof checking algorithms, including but not limited to free variables, capture-avoiding substitution, and application contexts. These metalevel operations/definitions are necessary to formulate the matching logic proof system in Section 1.2.

We have implemented a matching logic proof checker in Metamath (`http://metamath.org`) in 200 lines of code.[3]. Modulo all the technical and implementation details, the essence of the Metamath implementation is a reduction from checking matching logic proofs into three basic operations over strings:

1. checking string equivalence;
2. string substitution;
3. dictionary lookups.

[2]Do not confuse it with the soundness and completeness of a formal logic, which, intuitively speaking, is the equivalence between what is semantically true and what is provable. In this article, we do not care about semantics at all. The soundness and completeness property is a property about the proof checker.

[3]See `https://github.com/runtimeverification/proof-generation/blob/main/theory/matching-logic-200-loc.mm`.

2 Problem Statement

Mathematical proof objects can be very large. For example, the proof object for $\Gamma^{\mathsf{IMP}} \vdash \varphi_{\texttt{sum.imp}}$, where

$$\Gamma^{\mathsf{IMP}} = \text{formal semantics of IMP—a simple imperative language; 40 lines of } \mathbb{K} \tag{14}$$

$$\varphi_{\texttt{sum.imp}} = \text{a claim stating that the output of the following } \texttt{sum.imp} \text{ program is 5050} \tag{15}$$

$$\texttt{sum.imp} = \texttt{int n = 100, s = 0; while(--n)\{s = s + n;\}; output(s);} \tag{16}$$

consumes 83.4 MB of storage.[4] Therefore, mathematical proof objects are not good candidates for serving as *correctness certificates*. They are too large to exchange. On the other hand, it is unnecessary to use a complete mathematical proof object if all that we want to show is $\Gamma \vdash \varphi$. According to Equation (13), we only need show the *existence* of a mathematical proof object that will pass the proof checker.

The *research problem* is to design a succinct cryptographic proof system $(\mathcal{P}, \mathcal{V})$ with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. The prover $\mathcal{P}$ takes three inputs[5] and produces a succinct cryptographic proof π, i.e.,

$$\mathcal{P}(\Gamma, \varphi, \Pi) = \pi \tag{17}$$

The verifier $\mathcal{V}$ takes Γ, φ, and π, and decides whether it accepts or rejects the inputs, i.e.,

$$\mathcal{V}(\Gamma, \varphi, \pi) \in \{\texttt{accept}, \texttt{reject}\} \tag{18}$$

Such a cryptographic proof system should satisfy the following soundness and completeness property:

1. If $\texttt{pc}(\Gamma, \varphi, \Pi) = \texttt{success}$, we have $\mathcal{V}(\Gamma, \varphi, \mathcal{P}(\Gamma, \varphi, \Pi)) = \texttt{accept}$
2. If $\texttt{pc}(\Gamma, \varphi, \Pi) = \texttt{failure}$, we have $\Pr(\mathcal{V}(\Gamma, \varphi, \mathcal{P}'(\Gamma, \varphi, \Pi)) = \texttt{reject}) \geq 1 - \epsilon$ for any poly-time $\mathcal{P}'$.

3 Our Approach

What distinguishes our task from developing a SNARK/STARK for a generic program is the following crucial fact: *checking mathematical proof objects is an embarrassingly parallel problem.* Consider an arbitrary Hilbert proof $\langle \varphi_1, \varphi_2, \ldots, \varphi_n \rangle$. By Definition 8, for every $1 \leq i \leq n$, the pattern/claim φ_i is either an axiom or derivable using one of the 5 non-nullary rules of inference. Note that it is *local* to check whether φ_i is an axiom, in the sense that there are no premises. If φ_i is derivable, checking it requires to look up at most 2 existing claims.[6]

We utilize the fact that checking mathematical proof objects is an embarrassingly parallel problem to optimize our succinct cryptographic proof system. For example, we develop a circuit for every case in Definition 8. This results in 13 circuits; 12 of them are for the 12 (non-nullary and nullary) proof rules in Figure 1 and the last one is for checking membership $\varphi \in \Gamma$, which is Case (1) in Definition 8. Each instance that runs a circuit will receive a cryptographic commitment to the Hilbert proof to ensure that all the proof steps being checked can be aggregated together. Finally, we use a cryptographic certificate aggregation method such as recursive SNARKs or SnarkPack to aggregate all the one-step certificates into a final cryptographic certificate.

Throughout this section, let us assume the matching logic theorem being proved is

$$\Gamma \vdash \varphi_n \tag{19}$$

and a corresponding Hilbert-style proof is

$$\Pi = \langle \varphi_1, \ldots, \varphi_n \rangle \tag{20}$$

[4] See `https://github.com/runtimeverification/proof-generation/blob/main/examples/sum-imp-complete-proof.mm`.

[5] The prover/verifier can take more inputs such as a common reference string.

[6] (Modus Ponens) has two premises. The other rules of inferece have one premise each. Logical axiom schemas have no premises.

3.1 Preliminaries

We first introduce the preliminaries on the encoding function for matching logic, commitment schemas, and cryptographic certificates aggregation.

3.1.1 Encoding function

An *encoding function* $[\![\_]\!]$ is a mapping from matching logic patterns and theories into data. In this paper, we do not specify a particular encoding function to enjoy the flexibility to work with any encoding mechanism (textbook encoding, de Bruijn encoding (`https://en.wikipedia.org/wiki/De_Bruijn_index`), locally nameless encoding (`https://charguéraud.org/research/2009/ln/main.pdf`), nominal encoding (`https://www.cl.cam.ac.uk/~amp12/papers/nomlfo/nomlfo-draft.pdf`), ...) and representation method (S-expression (`https://en.wikipedia.org/wiki/S-expression`), Metamath (`metamath.org`), a binary representation, ...) We use $\mathcal{D}$ to denote the data space associated with the encoding function $[\![\_]\!]$. For $d \in \mathcal{D}$, we say that d is a *valid pattern encoding* (resp. *valid theory encoding*) iff there exists φ (resp. Γ) such that $d = [\![\varphi]\!]$ (resp. $d = [\![\Gamma]\!]$). We say that d is a *valid encoding* if it is a valid pattern encoding or a valid theory encoding. For technical simplicity, we require that patterns and theories do not share a common encoding; that is, $[\![\varphi]\!] \neq [\![\Gamma]\!]$ for any φ and Γ.

A *collision* is when there exist two different φ_1 and φ_2 such that $[\![\varphi_1]\!] = [\![\varphi_2]\!]$ (resp., two different Γ_1 and Γ_2 such that $[\![\Gamma_1]\!] = [\![\Gamma_2]\!]$). An *injective* encoding, such as textbook encoding or nominal encoding, guarantees that there is no collision; that is, $[\![\varphi_1]\!] = [\![\varphi_2]\!]$ implies $\varphi_1 \equiv \varphi_2$ (and the same for theories). De Bruijn encoding and locally nameless encoding are *injective-modulo* encoding, in the sense that $[\![\varphi_1]\!] = [\![\varphi_2]\!]$ implies φ_1 is α-equivalent to φ_2. In other words, they are injective modulo α-equivalence. Since our goal is to determine whether a given pattern is provable or not, we can use any encoding function that is injective modulo an equivalence relation E that is finer than logical equivalence; that is, $[\![\varphi_1]\!] = [\![\varphi_2]\!]$ implies $\varphi_1 \, E \, \varphi_2$, which implies $\emptyset \vdash \varphi_1 \leftrightarrow \varphi_2$. For textbook encoding or nominal encoding, E is syntactic identity. For de Bruijn encoding and locally nameless encoding, E is α-equivalence. Given an encoding function and a valid pattern encoding (resp. valid theory encoding) d, we write φ_d (resp. Γ_d) to denote a pattern (resp. theory) such that $[\![\varphi_d]\!] = d$ (resp. $[\![\Gamma_d]\!] = d$).

We are open to and interested in designing new encoding functions that are cryptography friendly. For example, an encoding function whose data space $\mathcal{D}$ is a polynomial vector space may allow us to directly implement operations over patterns and theories using the primitive operations over the polynomial vector space, without going through an extra layer of abstraction/interpretation of a certain ZK language or framework.

3.1.2 Non-Interactive Commitment Schemes

The following definition is from `https://shorturl.at/jmozS`.

Definition 9. A *commitment scheme* is a tuple of procedures ($\texttt{CS.Setup}, \texttt{CS.Commit}, \texttt{CS.VerCommit}$), where

1. $\texttt{CS.Setup}(1^\lambda) \to ck$ takes a security parameter $\lambda \in \mathbb{N}$ and outputs a commitment key ck. This key includes descriptions of the input space $\mathcal{I}$, commitment space $\mathcal{C}$, and opening space $\mathcal{O}$.
2. $\texttt{CS.Commit}(ck, u) \to (c, o)$ takes ck and a value $u \in \mathcal{I}$ and outputs a commitment c and an opening o.
3. $\texttt{CS.VerCommit}(ck, c, u, o) \to b$ takes ck as well as a commitment c, a value u, and an opening o, and accepts ($b = \texttt{accept}$) or rejects ($b = \texttt{reject}$).

A commitment scheme should satisfy the following properties:

1. (Correctness). For every $\lambda \in \mathbb{N}$ and $u \in \mathcal{I}$, we have $\Pr[\texttt{CS.VerCommit}(ck, c, u, o) = \texttt{accept} \mid ck = \texttt{CS.Setup}(1^\lambda), (c, o) = \texttt{CS.Commit}(ck, u)] = 1$.

2. (Binding). For every poly-time adversary $\mathcal{A}$, we have

$$\Pr[\texttt{CS.VerCommit}(ck, c, u, o) = \texttt{accept} \wedge \texttt{CS.VerCommit}(ck, c, u', o') = \texttt{accept} \wedge u' \neq u \mid ck = \texttt{CS.Setup}(1^\lambda), (c, u, o, u', o') = \mathcal{A}(ck)] < \epsilon$$

3. (Hiding). For $ck = \texttt{CS.Setup}(1^\lambda)$ and $u, u' \in \mathcal{I}$, the following two probabilistic distributions are statistically close: $\texttt{CS.Commit}(ck, u) \approx \texttt{CS.Commit}(ck, u')$.

Intuitively, a commitment scheme allows us to lock a value u within a safe box (commitment c), which can only be opened using the corresponding key (opening o). Given a theory Γ, we can compute its commitment c_Γ with an opening o_Γ and pass them to the proof nodes (explained in Section 3.2). This way, we ensure that all the proof nodes have access to the same theory Γ.

We also need a commitment scheme for the Γ-proof $\langle \varphi_1, \ldots, \varphi_n \rangle$, which allows to be opened only w.r.t. a specific position. This is known as *vector commitments*. The following definition is from `https://shorturl.at/wzE79`, adapted and simplified to fit our setting and notation.

Definition 10. A *vector commitment scheme* is a tuple of procedures $(\texttt{VC.Setup}, \texttt{VC.Commit}, \texttt{VC.VerCommit})$, where

1. $\texttt{VC.Setup}(1^\lambda, n) \to ck$ takes a security parameter $\lambda \in \mathbb{N}$ and a size $n \in \mathbb{N}$, and outputs a commitment key ck. This key includes descriptions of the input space $\mathcal{I}$, commitment space $\mathcal{C}$, and opening space $\mathcal{O}$.

2. $\texttt{VC.Commit}(ck, \vec{u}) \to (c, o)$ takes ck and an n-length vector $\vec{u} = \langle u_1, \ldots, u_n \rangle$ with $u_1, \ldots, u_n \in \mathcal{I}$, and outputs a commitment c and an opening o.

3. $\texttt{VC.VerCommit}(ck, c, u, i, o) \to b$ takes ck as well as a commitment c, a value u, a position i, and an opening o, and accepts ($b = \texttt{accept}$) or rejects ($b = \texttt{reject}$).

A vector commitment should satisfy the following properties (where $\vec{u} = \langle u_1, \ldots, u_n \rangle$):

1. (Correctness). For every $\lambda \in \mathbb{N}$ and $\vec{u}$, we have $\Pr[\texttt{CS.VerCommit}(ck, c, u_i, i, o) = \texttt{accept} \mid ck = \texttt{CS.Setup}(1^\lambda, n), (c, o) = \texttt{CS.Commit}(ck, \vec{u})] = 1$.

2. (Binding). For every poly-time adversary $\mathcal{A}$, we have

$$\Pr[\texttt{CS.VerCommit}(ck, c, u_i, i, o) = \texttt{accept} \wedge \texttt{CS.VerCommit}(ck, c, u', i, o') = \texttt{accept} \wedge u' \neq u \mid ck = \texttt{CS.Setup}(1^\lambda, n), (c, \vec{u}, i, o, u', o') = \mathcal{A}(ck)] < \epsilon$$

3. (Hiding). For $ck = \texttt{CS.Setup}(1^\lambda, n)$ and $\vec{u}, \vec{u}' \in \mathcal{I}$, the following two probabilistic distributions are statistically close: $\texttt{CS.Commit}(ck, \vec{u}) \approx \texttt{CS.Commit}(ck, \vec{u}')$.

Given a Hilbert Γ-proof $\Pi = \langle \varphi_1, \ldots, \varphi_n \rangle$, we can compute its vector commitment c_Π with an opening o_Π and pass them to the proof nodes. This way, we ensure that all the proof nodes have access to the same Hilbert Γ-proof, without needing to send the entire Hilbert Γ-proof to each proof node.

3.2 Proof Network Architecture

We introduce a *proof network* for checking the correctness of any given Hilbert Γ-proof $\langle \varphi_1, \ldots, \varphi_n \rangle$ in parallel. The proof network $\mathcal{N}$ consists of $n + 2$ nodes, as follows:

1. An *initial node IN* that stores Γ and $\varphi_1, \ldots, \varphi_n$, as well as a proof annotation α. The proof annotation includes additional information that simplifies the proof checking procedures. The initial node is in charge of dispatching the entire proof checking task into n sub-tasks, which will be carried out by n proof nodes in parallel.

Algorithm 1: Initial Node

Data: A theory Γ, a Hilbert Γ-proof $\Pi = \langle \varphi_1, \ldots, \varphi_n \rangle$ of length n, and a proof annotation α
Parameters: Two security parameters $\lambda_\Gamma, \lambda_\Pi \in \mathbb{N}$
Description: Dispatch the proof checking task into sub-tasks and send them to n proof nodes.

// set up and commit Γ
$ck_\Gamma \leftarrow \texttt{CS.Setup}(1^{\lambda_\Gamma})$ and $(c_\Gamma, o_\Gamma) \leftarrow \texttt{CS.Commit}(ck_\Gamma, [\![\Gamma]\!])$;
// set up and commit $\Pi = \langle \varphi_1, \ldots, \varphi_n \rangle$
$ck_\Pi \leftarrow \texttt{CS.Setup}(1^{\lambda_\Pi})$ and $(c_\Pi, o_\Pi) \leftarrow \texttt{CS.Commit}(ck_\Pi, \langle [\![\varphi_1]\!], \ldots, [\![\varphi_n]\!] \rangle)$;
// dispatch the task; note that this can also be done in parallel
for $i = 1$ **to** n **do**
  // prepare the data to be sent to PN_i
  prepare the proof annotation α_i for φ_i, which, usually, can simply be extracted from α;
  switch *how* φ_i *is derived in* Π **do**
    case φ *is an assumption, i.e.,* $\varphi_i \in \Gamma$ **do**
      send $\langle \texttt{assump}, ck_\Gamma, ck_\Pi, c_\Gamma, o_\Gamma, c_\Pi, o_\Pi, i \rangle$ to PN_i via a trusted channel;
      send $\langle [\![\Gamma]\!], [\![\varphi_i]\!], \alpha_i \rangle$ to PN_i via an untrusted channel;
    case φ *is an instance of one of 7 logical axiom schemas* **do**
      send $\langle \texttt{ax}, ck_\Pi, c_\Pi, o_\Pi, i \rangle$ to PN_i via a trusted channel;
      send $\langle [\![\varphi_i]\!], \alpha_i \rangle$ to PN_i via an untrusted channel;
    case φ *is derived by* (Modus Ponens) **do**
      $j_1 \leftarrow$ index of the first premise of φ_i in Π;
      $j_2 \leftarrow$ index of the second premise of φ_i in Π;
      send $\langle \texttt{mp}, ck_\Pi, c_\Pi, o_\Pi, j_1, j_2, i \rangle$ to PN_i via a trusted channel;
      send $\langle [\![\varphi_{j_1}]\!], [\![\varphi_{j_2}]\!], [\![\varphi_i]\!], \alpha_i \rangle$ to PN_i via an untrusted channel;
    case φ *is derived by any other rule of inference (which has one premise)* **do**
      $j \leftarrow$ index of the premise of φ_i in Π;
      send $\langle \kappa, ck_\Pi, c_\Pi, o_\Pi, j, i \rangle$ to PN_i via a trusted channel, where κ indicates the rule of inference;
      send $\langle [\![\varphi_j]\!], [\![\varphi_i]\!], \alpha_i \rangle$ to PN_i via an untrusted channel;
  end
end

2. *Proof nodes* $PN_1, \ldots, PN_n$, which are called by the initial node. Each proof node checks only one step of the Hilbert proof and produces a cryptographic certificate. All the cryptographic certificates will then be passed to the aggregation node.

3. An *aggregation node* AN that aggregates the cryptographic certificates produced by the proof nodes into the final cryptographic certificate.

3.2.1 Initial Node

The initial node IN runs Algorithm 1. In the algorithm, we only send the commitments of Γ and/or Π, the index of the pattern being proved, and a type code that indicates how it is proved, to the proof nodes via a trusted channel. All the other information such as the pattern itself and proof annotations can be sent via an untrusted channel to reduce cost.

3.2.2 Proof Nodes

Each proof node is equipped with a *basic procedure*, denoted `basic_prove`, which checks the correctness of one Hilbert proof step. A proof node calls *basic procedure* for proof checking and verifies the integrity of the

Algorithm 2: Proof Node: Main Entry Point

Description: Receive a task from the Initial Node IN, perform proof checking, verify integrity, and send the certificate to the Aggregation Node.

```
// receive a task
receive data = ⟨id, ...⟩ from IN via the trusted channel;
switch id do
    case id = assump do
        execute proof_node_assump in Algorithm 3;
    case id = ax do
        execute proof_node_ax in Algorithm 4;
    case id = mp do
        execute proof_node_mp in Algorithm 5;
    otherwise do
        // set κ to be the rule of inference
        κ ← id;
        execute proof_node_other in Algorithm 6;
    end
end
```

data received from the untrusted channel using the commitments received from the trusted channel. The algorithm that runs on a proof node is listed in Algorithm 2, which calls sub-algorithms listed in Algorithms 3 to 6.

3.2.3 Aggregation Node

The aggregation node AN simply collects all the cryptographic certificates from the proof nodes and aggregate them into a final certificate, using techniques such as recursive SNARKs, SnarkPack[7], or incrementally verifiable computation (IVC).

[7] https://eprint.iacr.org/2021/529

Algorithm 3: Proof Note: `proof_node_assump`

Description: Called when a proof node receives `assump` data in Algorithm 2

receive $\langle \texttt{assump}, ck_\Gamma, ck_\Pi, c_\Gamma, o_\Gamma, c_\Pi, o_\Pi, i \rangle \leftarrow$ the trusted channel;
`// receive data from the untrusted channel`
`// the primes indicate that the data may be altered and is not trustworthy`
receive $\langle [\![\Gamma]\!]', [\![\varphi_i]\!]', \alpha_i' \rangle \leftarrow$ the untrusted channel;
if `basic_prove`$(\texttt{assump}, [\![\Gamma]\!]', [\![\varphi_i]\!]', \alpha_i') = \texttt{reject}$ **then**
  `// proof check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// proof check succeeds; need to check integrity`
if `CS.VerCommit`$(ck_\Gamma, c_\Gamma, [\![\Gamma]\!]', o_\Gamma) = \texttt{reject}$
  or `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_i]\!]', i, o_\Pi) = \texttt{reject}$ **then**
  `// integrity check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// integrity check succeeds; produce a certificate and send it to` AN
produce π_i for the i-th proof step;
send π_i to AN via an untrusted channel;

Algorithm 4: Proof Note: `proof_node_ax`

Description: Called when a proof node receives `ax` data in Algorithm 2

receive $\langle \texttt{ax}, ck_\Pi, c_\Pi, o_\Pi, i \rangle \leftarrow$ the trusted channel;
`// receive data from the untrusted channel`
`// the primes indicate that the data may be altered and is not trustworthy`
receive $\langle [\![\varphi_i]\!]', \alpha_i' \rangle \leftarrow$ the untrusted channel;
if `basic_prove`$(\texttt{ax}, [\![\varphi_i]\!]', \alpha_i') = \texttt{reject}$ **then**
  `// proof check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// proof check succeeds; need to check integrity`
if `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_i]\!]', i, o_\Pi) = \texttt{reject}$ **then**
  `// integrity check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// integrity check succeeds; produce a certificate and send it to` AN
produce π_i for the i-th proof step;
send π_i to AN via an untrusted channel;

Algorithm 5: Proof Note: `proof_node_mp`

Description: Called when a proof node receives `mp` data in Algorithm 2

receive $\langle \mathtt{ax}, ck_\Pi, c_\Pi, o_\Pi, j_1, j_2, i \rangle \leftarrow$ the trusted channel;
`// receive data from the untrusted channel`
`// the primes indicate that the data may be altered and is not trustworthy`
receive $\langle [\![\varphi_{j_1}]\!]', [\![\varphi_{j_2}]\!]', [\![\varphi_i]\!]', \alpha_i' \rangle \leftarrow$ the untrusted channel;
if `basic_prove`$(\mathtt{mp}, [\![\varphi_{j_1}]\!]', [\![\varphi_{j_2}]\!]', [\![\varphi_i]\!]', \alpha_i') =$ `reject` **then**
  `// proof check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// proof check succeeds; need to check integrity`
if `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_{j_1}]\!]', j_1, o_\Pi) =$ `reject`
  or `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_{j_2}]\!]', j_2, o_\Pi) =$ `reject`
  or `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_i]\!]', i, o_\Pi) =$ `reject` **then**
  `// integrity check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// integrity check succeeds; produce a certificate and send it to` AN
produce π_i for the i-th proof step;
send π_i to AN via an untrusted channel;

Algorithm 6: Proof Note: `proof_node_other`

Description: Called when a proof node receives `other` data in Algorithm 2

receive $\langle \kappa, ck_\Pi, c_\Pi, o_\Pi, j, i \rangle \leftarrow$ the trusted channel;
`// receive data from the untrusted channel`
`// the primes indicate that the data may be altered and is not trustworthy`
receive $\langle [\![\varphi_j]\!]', [\![\varphi_i]\!]', \alpha_i' \rangle \leftarrow$ the untrusted channel;
if `basic_prove`$(\kappa, [\![\varphi_j]\!]', [\![\varphi_i]\!]', \alpha_i') =$ `reject` **then**
  `// proof check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// proof check succeeds; need to check integrity`
if `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_j]\!]', j, o_\Pi) =$ `reject`
  or `VC.VerCommit`$(ck_\Pi, c_\Pi, [\![\varphi_i]\!]', i, o_\Pi) =$ `reject` **then**
  `// integrity check fails at` i`-th step; no certificate will be produced`
  error handling;
  return `reject`
end
`// integrity check succeeds; produce a certificate and send it to` AN
produce π_i for the i-th proof step;
send π_i to AN via an untrusted channel;