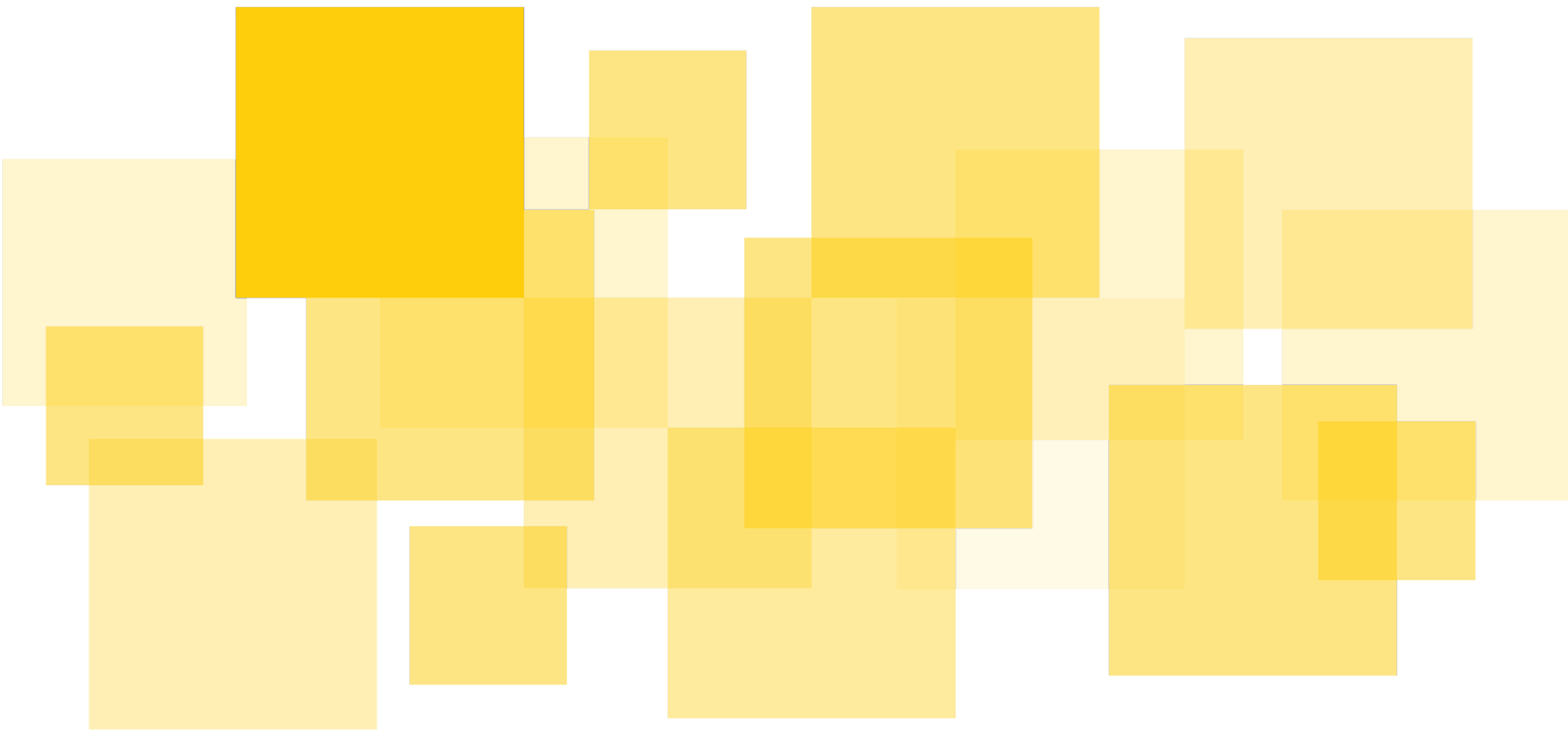
Universal Truth Framework & The Proof Chain

May 2023



runtime verification

Universal Truth Framework & The Proof Chain

— Vision —

Grigore Rosu — RV & UIUC

This is a shortened version of our RV Vision (2023-2025), focusing exclusively on the vision underlying the Universal Truth Framework and its blockchain application, the Proof Chain. Please consult the RV Vision document for a broader picture of Runtime Verification, Inc., its mission, vision and products. This document is based on our Proof Chain Vision video.

1. Introduction and Motivation

Bitcoin is the most secure decentralized network ever created, with a relatively simple consensus algorithm and little to no need for upgrades. However, Bitcoin's incapacity to execute custom programs, or smart contracts, has led to the creation and proliferation of many blockchains as World Computers, starting with Ethereum. The current programmable blockchains, however, have a series of limitations that inherently prohibit them from achieving the full potential that the blockchain technology can have on our society. Limitations such as:

- *Duplication of computation*. Currently, blockchain nodes re-execute the smart contracts. Regardless of whether the blockchain is based on proof of work, or on proof of stake, or on any other consensus mechanism, duplication of computation is bad. At a minimum, it wastes energy. In an ideal world, computation should be done only once and then the result of the computation should be propagated across the entire blockchain network using a stable consensus algorithm that is completely separated from computation, that is, from smart contracts, their programming languages and their execution.

- *Hardwired virtual machines or languages*. Currently, blockchains come equipped with predefined virtual machines or languages, like EVM, WASM, Plutus, Cairo, Move VM, etc., for all their smart contracts. Languages naturally evolve, with new versions every few months, which result in frequent blockchain upgrades; this is not only inconvenient and risky, but also likely inconsistent with blockchains being commodities. It also limits adoption among developers, because they need to learn new languages. Also, it exposes blockchain applications to bugs in language implementations, not to mention that many of these VMs or languages are new, so new compilers and/or interpreters need to be written for them under tremendous time pressure, which asks for bugs. In an ideal world, blockchains should allow smart contracts to be written in any VMs or programming languages and their execution, either directly using the language or indirectly by translation to other languages or VMs, should be off-chain choices/activities. Bugs in such off-chain tools or implementations should not propagate to the blockchain.

- *Extrinsic correctness and security*. Because of the decentralized, public and irreversible nature of blockchains, applications require correctness and security guarantees like never before. Even a minor bug in a smart contract, such as an arithmetic overflow, can result in millions lost and the end of a business. Current blockchains have no intrinsic support for attesting the correctness or security of smart contracts, or even to link them to their specifications. Currently, these are external efforts/activities, done off-chain via uncheckable documentation (PDF or readme files) or complex, adhoc formal verification or auditing tools, and auditors who provide no guarantee other than their reputation. In an ideal world, smart contract stakeholders should be allowed to make as many security or correctness claims as they find fit, and those should be verified and stored on the blockchain as well. This not only increases users' confidence in those smart contracts, but also enables a new level of compositionality and integration of smart contracts; for example, a bridge contract can check on the chain if a token is a proper ERC20. Speaking of bridges, in an ideal world there should be none. Currently, they exist as an artifact of fragmentation and lack of an ideal solution. If Bitcoin, for example, offered all the ideal features above, then we would have no other blockchains and thus no bridges.

Proof Chain is our proposal for a new kind of blockchain, built specifically to avoid the current blockchain limitations. A key insight of the Proof Chain is that *computation is proof*: computation done using any program written in any language is a special case of a mathematical proof in a universal logic formalism, as is, without any encoding or translation to another language. The Proof Chain has one task, and only one task: to achieve consensus by verifying such proofs.

While Proof Chain may seem very general in scope, in fact it is an instance of a more general framework which transcends blockchains, one aimed at asserting and attesting the truth of claims. We call it the Universal Truth Framework (UTF). In the rest of this article we first discuss the UTF, then the Proof Chain as a blockchain instance of the UTF, and then we give details on the scientific innovations that made the UTF and the Proof Chain possible now.

2. Universal Truth Framework

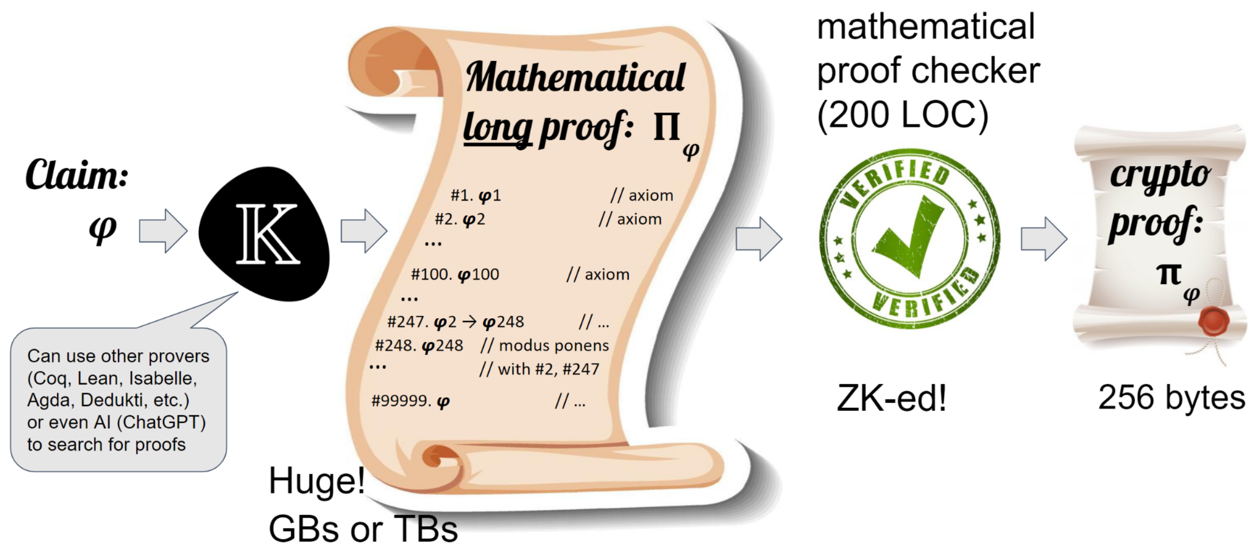
Recent advances in proof generation and verification have led to fragmentation. We propose the *Universal Truth Framework (UTF)* as a unifying framework, purposely general in scope, yet achievable, encompassing everything that can be expressed as a formal statement that admits a proof in a mathematical theory. This includes the entire field of computing, as well as that of formal semantics and verification.

In the UTF, every claim that is made is *verifiably true*. Specifically, claims come with independent, succinct, third party checkable *proof certificates*. A claim is anything that is provable, in particular computable: the execution of a program, a particular work that has been done or an action that has been performed, the formal correctness or security of some code, or any mathematical result. In the UTF, all these are formalized as mathematical *theorems*. Any true claim φ will have a succinct cryptographic proof π_φ. We can check π_φ virtually instantaneously and thus know that φ is true. UTF will facilitate both claim producers to also produce proofs for claims, and claim consumers to attest the validity of claims by checking the cryptographic proofs that come with those claims.

UTF captures several major existing approaches, but will have many applications beyond those:

- Verifiable Computing would now work for all programming languages. Basically, we can execute our code securely in untrusted environments, say in the cloud. Indeed, the execution environment can give us back a proof certificate for the claimed execution. We verify the proof certificate and thus we know that the computation was correct.
- Zero Knowledge capability available for all languages, similar to zkEVM, Cairo, zkVM (RiskZero), zkLLVM (=nil; Foundation), etc. All correct by construction, because semantically correct program execution is a claim in the UTF, in its language's theory.
- Formal verification / correctness claims, security audits, program analyses, etc., become checkable certificates (vs PDFs). We don't have to trust the developers of the smart contracts, or their auditors, or anybody else. We simply check the claim certificates.
- Critical procedures or devices in hospitals, aviation, cars, robots, etc., yield checkable correctness certificates, increasing confidence in complex systems and processes, in machines, even in AI. We don't have to trust them, we check their claim certificates.

Our proposal for the UTF is to combine proof generators, like those provided by the tools in the K Framework, with Zero Knowledge (ZK) technology:


Claim:
φ
K
Can use other provers (Coq, Lean, Isabelle, Agda, Dedukti, etc.) or even AI (ChatGPT) to search for proofs
Mathematical long proof: Π_φ
#1. φ1 // axiom
#2. φ2 // axiom
...
#100. φ100 // axiom
...
#247. $\varphi2 \rightarrow \varphi248$ // ...
#248. φ248 // modus ponens // with #2, #247
...
#99999. φ // ...
Huge! GBs or TBs
mathematical proof checker (200 LOC)
VERIFIED
ZK-ed!
crypto proof: π_φ
256 bytes

We start with the claim φ and pass it to a proof generator, like the K framework. Think of K as a searcher in a huge space of possibilities for a mathematical proof of our claim. Our claim can be that this program execution is correct, or that this protocol is correct, or anything that can be formally stated. Then K, with its suite of tools, will search for a mathematical proof for our claim. K can use many other tools as helpers. All these help us to search for a mathematical proof for our claim φ. We can even use AI, like GPT, to find helping lemmas and invariants. We can truly think of K as a mighty searcher of a mathematically rigorous proof Π_φ of our claim φ.

The problem with mathematically rigorous proofs, or *proof objects*, is that they can be very long. When we go down to the axioms of mathematics and logic reasoning, mathematical proofs can be huge. Nevertheless, they are the ultimate correctness certificates for claims. Unfortunately, their size is a major problem for us, both in terms of space and checking time, because we want to ship them with the claims they certify and the consumers of the said claims to check them.

Can we reduce the size of mathematical proofs significantly, maybe even to constant size, in a way that makes their checking, or verification, very fast, maybe even constant? It turns out, with the latest developments in mathematical proof checking and zero knowledge, this is becoming increasingly possible. The advantage of having a very detailed mathematical proof $\boldsymbol{\Pi_\varphi}$ is that we can rigorously check it with a very small and simple proof checker. We were able to recently implement a surprisingly small proof checker of only 200 lines of code. Therefore, with this 200 LOC proof checker, we can check any claim … made by any program … in any programming language (PL). Furthermore, we compiled this proof checker as a zero knowledge (ZK) circuit, using RiscZero's zkVM. This ZK circuit yields a cryptographic certificate $\boldsymbol{\pi_\varphi}$ that a mathematical proof ($\boldsymbol{\Pi_\varphi}$) for the public claim $\boldsymbol{\varphi}$ has been presented to it and checked and passed.

Therefore, our proposal to implement the UTF is to use language frameworks, like K, to produce (long) mathematical proofs ($\boldsymbol{\Pi_\varphi}$), followed by (small) ZK-ed proof checkers that yield succinct cryptographic proofs ($\boldsymbol{\pi_\varphi}$). We call it *Proof of Proof*: ZK proof of mathematical proof. The two components can be piped, for efficiency. Note that we don't have to trust complex language frameworks or proof generators, like K. We only have to trust the ZK-ed proof checker, which is public, small and the same for all languages and all claims (execution, formal verification, etc.).

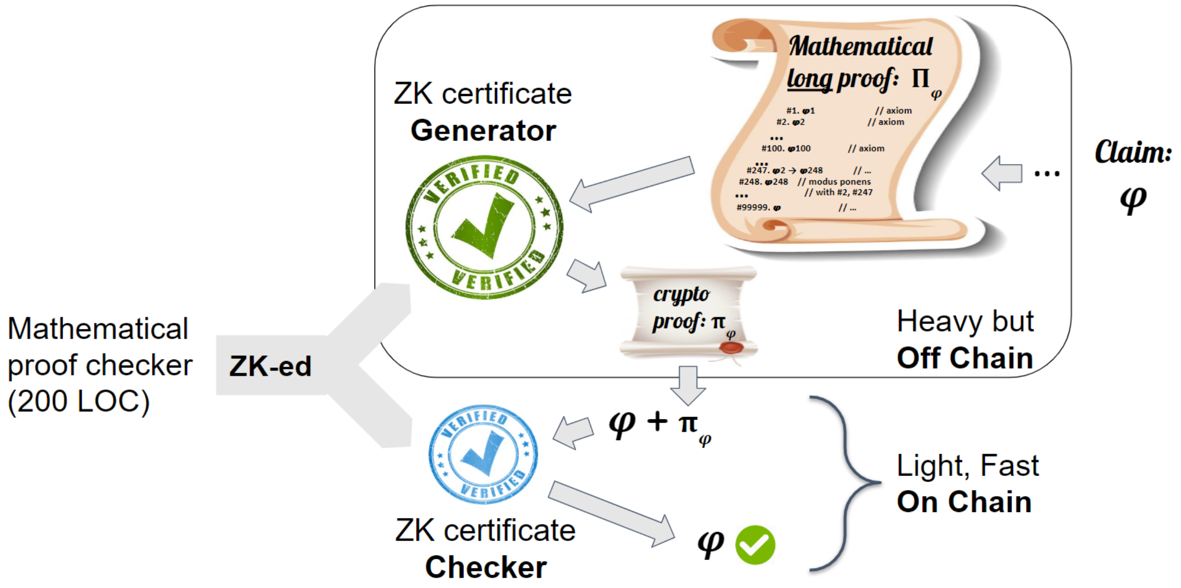
3. Proof Chain — The Ultimate Layer Zero

The UTF does not require a blockchain. It is a general framework for attesting the truth of claims. Producers of claims produce succinct proofs for their claims and consumers of claims check the proofs and are free to do whatever they want with the true claims. However, in many cases, claims need to be stored somewhere; for example, to use them to build other claims. There is no better place than a blockchain. We can use any blockchain for that, we can even incorporate the UTF as a Layer 2 ZK Rollup on any blockchain. But the big question is: what would the *minimal* requirements be for a blockchain to support the UTF? The Proof Chain is our answer. The main design principle was simple: if something can go off-chain, then it *must*.

The result took us by surprise. Such a minimal blockchain infrastructure only needs to do consensus, and as part of that, to only execute one light program, a ZK Certificate Checker. It has no hardwired or predefined VM or PL. Its validators need not (re)execute smart contracts, as these are executed off-chain, once, and produce ZK certificates; validators only check the ZK certificates and reach consensus. Yet, it will allow users to add any PLs and VMs, and to write smart contracts in any of them. Compilers (and their bugs) are not needed anymore, as one can directly interpret the source code; but they can be used, too, it is simply an off-chain choice. All claims, including all program executions, are correct-by-construction: they all come with proofs of proof, generated off-chain. There are no bridges as we know them anymore, all transactions take place on-chain, no tokens need to leave the chain. And finally, it transcends programs and their correctness: it is a blockchain of knowledge, where any provable claim can be stored and reused. From Proof Chain's angle, there is no difference between Pythagoras theorem and the fact that Alice transferred Bob 10 ETH. They are both provable claims.

The UTF does not require any specific underlying logical foundation. All it requires is that the claims, their mathematical proofs, and the proof checker use the exact same logical formalism. To instantiate the UTF, however, as required by the Proof Chain, we must choose one. Moreover, we want it to be expressive enough to capture any PL or VM as a theory, and any claim as a theorem. To have confidence that we make the right choice and can capture the entire field of mathematics, we want a logical formalism that can capture any other logical formalism as a *shallow embedding* theory, that is, as notations: if logic L1 shallowly embeds logic L2, then everything that L2 can express and do can also be directly expressed and done with L1, that is, L1 is more expressive than L2. Moreover, we want a *minimal* such logic L1, because we want its proof checker, which is the only component of the UTF that needs to be trusted, to be minimal. Finding such a logical formalism is orthogonal to the UTF and has been the target of our research for more than 20 years. The result of our quest for the ideal logical foundation is Matching Logic. We have also implemented a proof checker for Matching Logic and instrumented some of the K Framework tools to produce Matching Logic proof objects. Consequently, we chose Matching Logic as an underlying logical formalism of the Proof Chain.

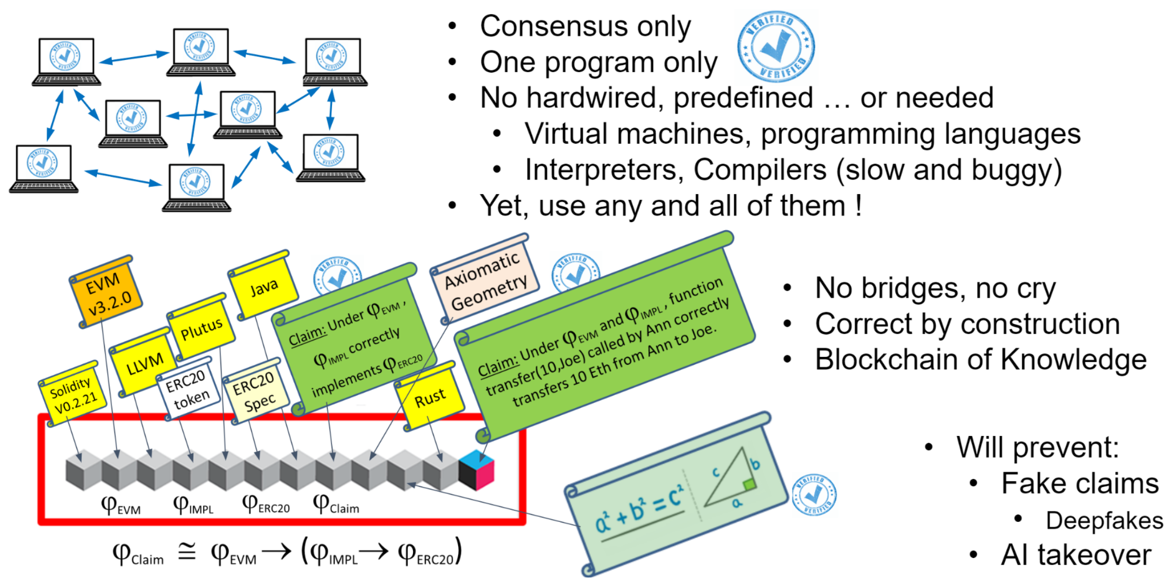
We next discuss the high-level architecture of the Proof Chain, essentially what goes on-chain and what goes off-chain. At the core of our UTF instance in the Proof Chain, we have the 200 LOC Matching Logic proof checker, implemented as a ZK circuit. This results into two different ZK components, one off-chain and the other on-chain:



The *ZK Certificate Generator* does the heavy lifting: it takes the (long) mathematical proof objects Π_φ of the claims φ as input, and produces the (succinct) cryptographic proof certificates π_φ as output. The ZK certificate generator will be part of the off-chain stack and will likely be executed on fast machines in clouds and warehouses and will be offered as part of larger services by various for-profit companies, like RV, whose main role is to search for proofs of claims. Searching for proofs is hard and open-ended in terms of techniques used. We believe this is a positive, because it will stimulate competition, creativity, and new technologies to be used for a noble purpose (eg AI/GPT, automated reasoning, specialized hardware, etc).

The second component is the *ZK Certificate Checker*. Its role is to check the ZK certificates π_φ and thus confirm the validity of the claims φ. The ZK Certificate Checker is fast and it will be incorporated in the validators' code. In fact, validators execute only this program as part of their overall consensus. They will run no virtual machines and know nothing about any specific programming languages. These will be part of the claims they check and will likely be pulled from their respective accounts on the Proof Chain, for checking (not execution) purposes.

The picture below illustrates how the Proof Chain operates and shows its benefits:


- Consensus only
- One program only
- No hardwired, predefined … or needed
  - Virtual machines, programming languages
  - Interpreters, Compilers (slow and buggy)
- Yet, use any and all of them !

- No bridges, no cry
- Correct by construction
- Blockchain of Knowledge

- Will prevent:
  - Fake claims
    - Deepfakes
  - AI takeover

EVM v3.2.0
Java
Plutus
LLVM
Solidity v0.2.21
ERC20 token
ERC20 Spec
Rust
Axiomatic Geometry
Claim: Under φ_{EVM}, φ_{IMPL} correctly implements φ_{ERC20}
Claim: Under φ_{EVM} and φ_{IMPL}, function transfer(10,Joe) called by Ann correctly transfers 10 Eth from Ann to Joe.
φ_{EVM} φ_{IMPL} φ_{ERC20} φ_{Claim}
$a^2+b^2=c^2$
VERIFIED

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

We next elaborate on some of the advantages of the Proof Chain compared with the state-of-the-art, and in the process of doing so we give more details on how it works:

Simplicity. Proof Chain will essentially consist of only a lightweight consensus layer and only one program to be executed by validators, the ZK Certificate Checker. There will be no hardwired or predetermined VM or PL. The importance of this cannot be overstated. Successful VMs and PLs evolve by design, with a new release every few months, to keep up with advances in technology, programming abstractions, and education of new generations. If they are hardwired in blockchains' validators, then they need to be regularly upgraded. This is inconvenient, risky, and can raise concerns with regulators, who may rightfully assume an entity in charge. Once the Proof Chain is launched, there will be no need for upgrades. Unlike PLs and VMs, mathematical logic does not change. So it is a better choice for blockchains. Proof Chain will be as simple as Bitcoin, but with all the additional features and benefits.

Generality. In spite of having no predetermined VMs or PLs, Proof Chain will allow programs to be written in virtually all PLs and to be executed either directly, by interpreting the PL, or via compilation to any VM if one chooses so. This will be possible by simply adding the desired PL or VM on the chain, as normal data. Programs in said PL or VM are data as well. Execution or correctness of programs in such PLs become normal claims, which come with proofs that will be checked by the validators. Blockchains as we know them, or subnets or sidechains, become clusters in Proof Chain, all related through a PL, or a VM, or a topic of interest, or geopolitical regulations, etc. Proof Chain's mission is to provide the infrastructure for such clusters to form.

***Correctness / Security*.** All claims are provably correct. For example, the claim that an ERC20 token was correctly transferred from Ann to Joe is a theorem, proved from the theory corresponding to the EVM semantics and the ERC20 token implementation; the claim that the ERC20 token implementation is correct is a theorem proved from the theory corresponding to the EVM semantics and the ERC20 specification. Bridges as we know them will not be needed. Transfers will be on the chain, with normal transactions, secure as anything else; eg., to transfer 10 ETH from an account in the "Ethereum cluster" to an account in the "Cardano cluster", one would need to sign two transactions, one burning/freezing the 10 ETH in Ethereum cluster and another minting/unfreezing 10 ETH on Cardano cluster. All on Proof Chain. Nothing off-chain.

***Efficiency*.** All the heavy lifting, both searching for proofs and generating ZK certificates, takes place off-chain. Only checking the ZK certificates happens on-chain, which is fast and cheap. That means that once the transactions are fully proved off-chain, using any mechanisms there ranging from instrumenting interpreters to complex theorem provers, or even AI, they can be efficiently deployed on the Proof Chain. Both the off-chain and the on-chain components can take advantage of parallelism and transaction independence. As usual, the off-chain computations will dictate the latency, while the on-chain component will dictate the throughput.

We discuss two other Proof Chain advantages when compared to existing blockchains:

***Separation of Concerns*.** Proof Chain massively separates concerns. It separates the PLs, VMs and other specifications and protocols from the blockchain implementation. It separates the PLs and VMs from their own implementations, too. It separates correct functionality from correctness of tools, using Translation Validation (validate each use of the tool, not the tool). Finally, it separates responsibilities and, ultimately, blame when things go wrong.

***Foster Innovation*.** It will stimulate and inspire participants at all levels, even more than the combined blockchains today. Proof Chain will be a store of value, similar to Bitcoin, where "value" is any claim. Like Bitcoin, the basic capability of Proof Chain is to achieve consensus, using one stable program. Nobody will be in charge of the Proof Chain, and it should not need to be upgraded. Adding a new language is equivalent to starting a new blockchain, or sidechain or subnet, and will be achieved with one transaction. Dependence on particular languages or VMs will gradually disappear. Businesses can invent their own DSLs targeted to their application domain, then have their clients use it right away, as if they had their own blockchain built on top of their DSL. Languages will evolve more systematically and less painfully for developers, from one formal specification to another, everything transparently on the Proof Chain

Off-chain businesses focused on searching for proofs will flourish. Anybody and anything can produce a proof to a claim; the problem is well-defined and completely transparent. RV uses the K framework for that, but other companies may use different approaches. RV will have a competitive advantage initially, because K was designed in the same spirit of separation of concerns as the Proof Chain, but we believe that in 2-3 years we will see many other similar service providers, specialized on various types of claims: execution, formal verification, fake claims like deepfakes, NFTs, etc. The Proof Chain will encourage good practices, like formal semantics, formal verification and reasoning, and mathematics education. For example, a formally verified program is not only more trustworthy, but it can also be … faster! With intelligent proof engineering, we can extract a proof of a program's execution which is much smaller than the naive proof obtained by executing the program blindly. Developers of smart contracts will be incentivised to formally verify their code not only because that is the right thing to do, but also because their users will see faster end-to-end execution and pay less gas.

Last but not least, Proof Chain will lead to applications and innovations beyond the reach of current blockchains. Scientific theories can be formalized for educational purposes. From Proof Chain's perspective, there is no difference between Euclidean Geometry and the formal semantics of Rust. Classic results in these domains, such as Pytagoras theorem, become claims on the chain, same as any program execution, or any transaction, or any formally verified token. We see a future where much of our scientific knowledge will be stored on the Proof Chain. Importantly, this knowledge will be immediately available to everybody and every application, to be used to construct other claims and results. We are not concerned that AI will take over the world. AI is good at searching solutions for us, creative or complex solutions to hard problems. Those solutions will result in proofs, which will be checked by the Proof Chain. We are the ones who decide what to do with the proven claims, not the AI.

4. Why Now

What makes the UTF possible now is the convergence of three major scientific advances, none of which available a few years ago but all together truly available only now, starting with 2023:

- *Scalable Zero Knowledge*. While the concept of ZK proofs was proposed in 1985, it was only recently that ZK technology became usable at scale. It was only very recently, in 2022 and 2023, that ZK variants of languages, like zkEVM, Cairo, zkVM, zkLLVM, etc., have demonstrated the feasibility of the ZK technology to general-purpose computation.

- *Matching Logic*. Although first published in 2009, it was only in 2019 that we figured out its full expressiveness: it plays in the world of logic the same role K plays in the world of PLs: any other logic is captured as a *matching logic theory*. This gives us the green signal that it is the right logic for the UTF and the Proof Chain, not only for computational and program correctness claims, but also for scientific ones. The first Matching Logic proof checker was done in 2021, but the final 200 LOC one was only finalized in 2023.

- *Proof Objects*. In theory, any formal system backed by a logical formalism can and should produce proof objects for every claim it proves. In practice, very few such systems are capable of doing it. It is an unbelievably hard engineering challenge. The first implementation that demonstrated the capability to generate proof objects from program executions was published in 2021 for concrete execution and in 2023 for symbolic executions. Consequently, there was no satisfactory solution to generating mathematical proof objects as needed for the UTF and the Proof Chain until 2023.

Therefore, it is the timely convergence of advances in ZK technology, matching logic, and proof objects engineering that makes the UTF and the Proof Chain possible only now, in 2023.

We authored two of the three converging technologies: Matching Logic and Proof Objects. We were also the creators and main developers of the K Framework, which was the main source of inspiration for the UTF and which will play a key role in advancing and commercializing these initiatives. Regarding cryptography and zero knowledge, we have access to talent through our top CS research university (UIUC), and closely collaborate with our adviser Prof. Andrew Miller and his students. They connected us and initiated our collaboration with RiscZero, and helped with our Metamath implementation on their zkVM. We are carrying out discussions with teams behind other major ZK languages (zkEVM, zkLLVM, Cairo), to see which one is best for us. Our long term solution, however, is to eventually craft our own ZK circuit for Matching Logic.

Risking to brag a little, we believe that the idea underlying the UTF, namely the separation between uniformly producing mathematical proofs as justifications for claims followed by their checking using a ZK-ed proof checker to produce succinct certificates, is not obvious. We were the first to propose it, though, back in 2020. It was too early to pull it then, the basic ingredients were not ripe. The idea may look natural in hindsight, but to conceive it and work out all details, we need a foundational understanding of computation, languages, formal methods, semantics, verification, logic, and cryptography, paired with a strong belief that blockchain is the future.

As lifelong academics specialized in the above fields, with corporate experience in mission and safety critical systems, and as blockchain security auditors, we believe we've seen everything. We are qualified and ready to disrupt the world of blockchain with the UTF and the Proof Chain.

5. Conclusion

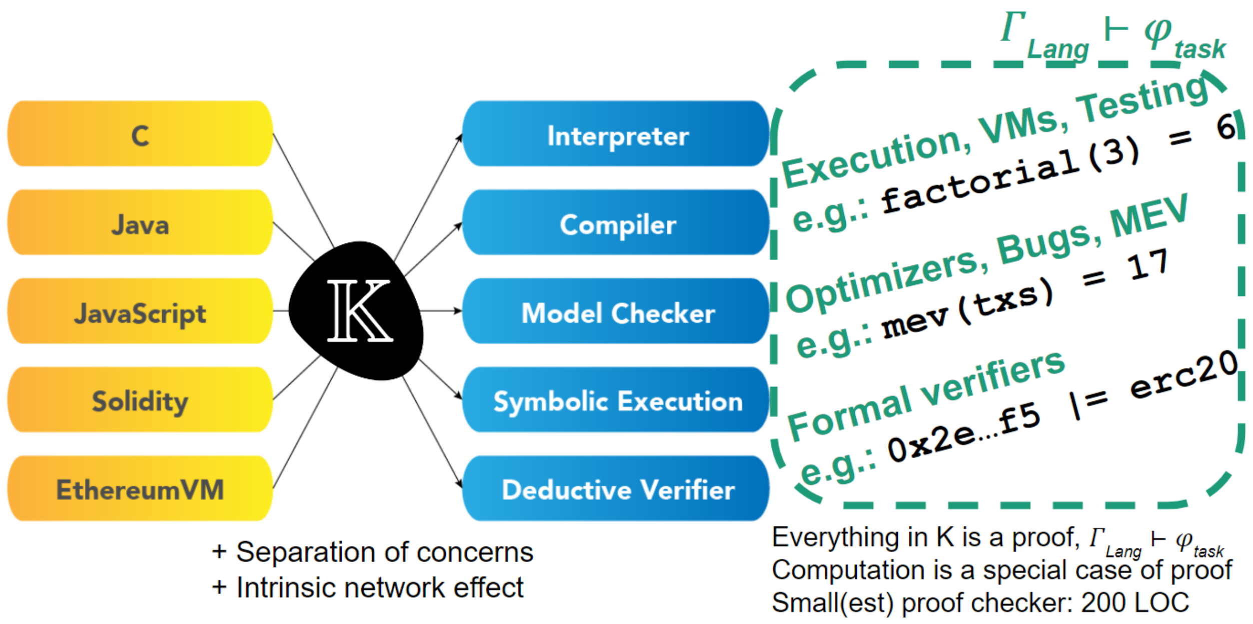
Proof Chain will be at least as fundamental as Bitcoin and Ethereum. It will be as simple, stable and secure as Bitcoin because, like Bitcoin, Proof Chain has one job: to achieve consensus. It will be as versatile as Ethereum, because, like Ethereum, Proof Chain allows programs. Proof Chain completely separates computation from consensus. Computation is done off-chain, using arbitrary languages, virtual machines, powerful tools, and fast hardware. Computation, like any other proved claim, produces a ZK certificate off-chain. Proof Chain's validators check the ZK certificate and admit the claim. This separation of concerns will allow users to write contracts in any programming languages, to port existing smart contracts from other blockchains, and to interoperate like never before. No bridges will be needed. With no language barriers, innovation and creativity will proliferate. So will education, because mathematical proofs will also go on the Proof Chain. There is no upper limit for how transformative the Proof Chain can be.

Appendix

Here we discuss the technical innovations that make the UTF and Proof Chain possible.

K Framework

K is a programming language semantic framework. We can think of it as a meta-language for languages, which provides us with virtually all the tools that we need for virtually all languages. First, we formally specify, or implement in a mathematical language, our target PL. Or take it from somewhere else, like from our open-source PL library. Then we pass it to one of the language-parametric tools in the K toolkit, or implement our own on top of the existing ones, or combine them, or from scratch using the provided K API. Now K generates our PL-specific tool.



Not unsurprisingly, initially the generic K tools were much slower than tools that were specifically crafted for a given language. But after 20 years of improvements, some of the tools reached a point where it is not easy to beat them with a manually written tool. For example, our KEVM interpreter outperforms most of the existing EVM interpreters. Same phenomenon was seen in the early times, where programs written in C were slower than those in Assembly; but now it is the other way around, it is hard to manually beat the compilers, as they incorporate an immense body of knowledge (loop fusion, levels of caching, threading, speculation, pre-fetching,etc). Same happened with K. Its tools were continuously improved incorporating the state-of-the-art in pattern matching, rule indexing, compilation, automatic reasoning, and so on, to a point where it takes a significant amount of manual effort to outperform K's automatically generated tools.

Besides offering an environment for language designers to properly design and publish their languages, instead of implementing them in some adhoc way, and for the community to scrutinize language designs in an abstract, human readable notation, K solves two major pain points: one for developers of languages and tools, and another for users of these.

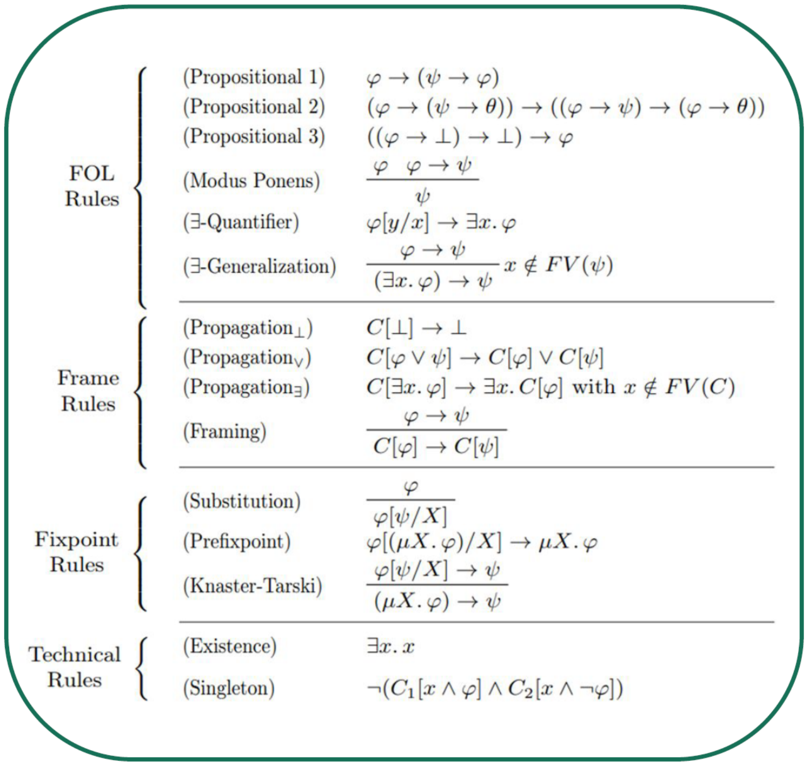
On the developers' side, it is painful and wasteful to implement different tools for the same language, tools which likely duplicate a lot of the code (like parsing, CFG-extraction, etc). Such tools become quickly obsolete, because their authors do not upgrade them as the languages evolve: there are 20 versions of Java now. Also, it is painful to re-implement the same tool, say a symbolic execution engine, for each language, using the same principles for symbolic transitions. K's separation of concerns has major implications in terms of increasing network effect, because we implement a language once and for all and then we get all the tools for that language. We improve the language, say go from Java 1.4 to Java 5, and now we've got all the tools already upgraded automatically for our new version of the language. Similarly, we are very motivated now to fix bugs or add improvements to the tools, because those affect all the languages. For example, if we make the symbolic execution engine faster in K, then we get a faster symbolic execution engine for all languages. Importantly, we only need to maintain the languages themselves and the tools themselves, separately, not their combinations.

While developers' pain may seem serious enough, in fact it pales in comparison with its consequences for the rest of us, the users of these tools. Why should we trust that a program executed correctly; for example, that the factorial(3) correctly evaluated to 6, in some given implementation of Java, or C, or Solidity? After all, there are hundreds of thousands of lines of code implementing the compilers and interpreters of these languages. The situation is even worse for tools meant to ensure correctness, such as model checkers or deductive verifiers. Why should we trust a tool claiming that the contract at address 0x2e…f5 correctly implements the ERC20 token specification? All the tools for all the languages, in the end, make claims that we are forced to accept. Claims of execution, claims of optimality, claims of correctness. We take risks on a daily basis by trusting all these claims. Risks that sometimes result in loss of money, or of expensive machinery, or even of human life (Wormhole, Ariane-V88, Therac-25).

This is how K addresses this problem. Any claim made by any tool is a mathematical theorem. A theorem which is valid in a mathematical theory, the formal semantics of the instance PL. The K tools can search for and produce mathematical proofs for claims. We want to emphasize that, in K, *computation is proof*: when we execute a program with the interpreter tool of K, we get a mathematical proof which demonstrates step by step that our program executes to the claimed result. For example, that factorial(3) executes to 6. We have a small proof checker, of only 200 lines of code, which can check any of the proofs done by any of these K tools, or any other tool that can produce mathematical proofs. We will discuss it shortly. But the point here is that everything that any of the K tools do for any of the PLs, can be checked for correctness using a 200 LOC proof checker. This proof checker is all we need to trust, not the complex K tools.

Matching Logic — Foundation of K (and Coq, Lean, etc)

Matching Logic is the foundation of K (and of other provers, too). It is the smallest logical foundation known for language semantics and formal verification, that has this expressiveness (monadic second-order logic). Its most general form, which includes support for least fixed points, was published in a LICS'19 paper and it is so small that you can write it on a napkin. Matching Logic has 7 syntactic constructs for building *formulae* and 15 *proof rules*. A semantics of a PL is defined as a *theory*, that is, a set of formulae called *axioms*. A proof rule allows us to derive a *conclusion* once we derive its *premises*. Premises are written above a horizontal line, and the conclusion is written underneath the line. See the *modus ponens* rule, for example: you can derive ψ once you derived φ and $\varphi \rightarrow \psi$. With these 15 proof rules we can derive, starting with the axioms in a target language semantics, theorems of interest. And such theorems are *correct by construction*, because we construct actual proofs for them, using the proof system:



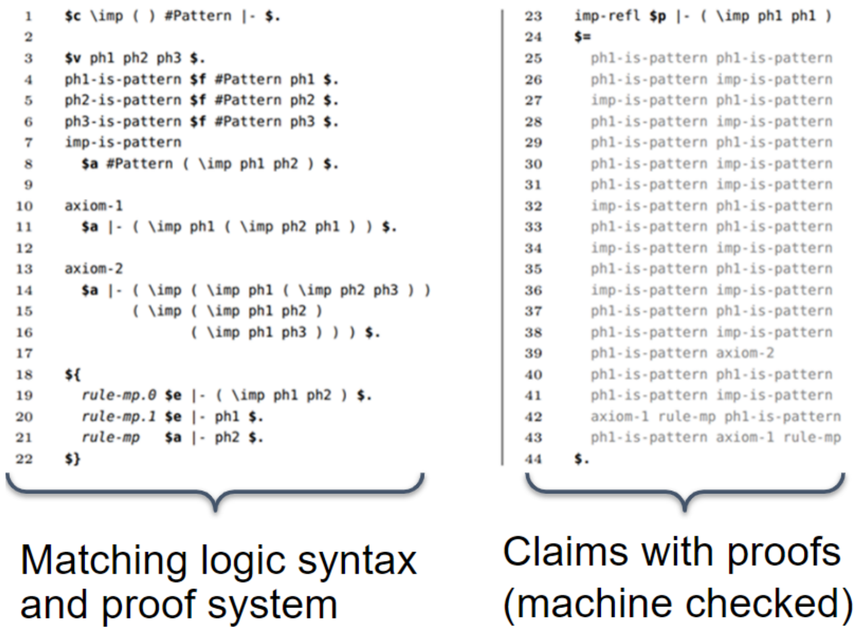
FOL Rules	(Propositional 1)	$\varphi \to (\psi \to \varphi)$
	(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
	(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
	(Modus Ponens)	$\dfrac{\varphi \quad \varphi \to \psi}{\psi}$
	(∃-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
	(∃-Generalization)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\ x \notin FV(\psi)$
Frame Rules	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
	(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ with $x \notin FV(C)$
	(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
Fixpoint Rules	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Prefixpoint)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
Technical Rules	(Existence)	$\exists x.\, x$
	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Despite its simplicity, Matching Logic is general and expressive. We refer the reader interested in foundational aspects to Matching Logic's website for details, history and papers. The bottom line is that this very small logic allows us to define any programming language as a theory, and to prove any claim about that language, including a correct program execution, as a theorem.

Everything that K or other provers do is a provable Matching Logic theorem φ from a language semantics Γ, written $\Gamma \vdash \varphi$. So any and all of these systems can be used to build Matching Logic proofs. See our ICFP'20 paper for details. However, we prefer K. We understand its code best, but that's not the only reason. As explained, in K, *computation is proof*: when we execute a program using the K generated interpreter, an actual direct proof object of that claim is being produced by that particular execution. Another reason is that K is very fast: its interpreters automatically generated from formal language semantics, for example, compete at performance with existing interpreters specifically hand-crafted for those languages. Finally, we already have many programming languages formalized in K: C, Java, EVM, AVM, WASM, and so on. It would be a huge effort to translate all those into other formalisms or provers.

200 LOC Proof Checker for Matching Logic

We chose Metamath as an implementation language for our proof checker. Metamath is a *very simple* language, *very low level and expressive* at the same time. There are 20+ Metamath verifier implementations already: in C, Rust, Haskell, OCaml, and so on. Most of them have only a few hundred lines of code; the largest, in C, has 2500 lines of C code. What we did was to implement Matching Logic in Metamath, in 200 lines of code (199, in fact):



```
$c \imp ( ) #Pattern |- $.

$v ph1 ph2 ph3 $.
ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $.
ph3-is-pattern $f #Pattern ph3 $.
imp-is-pattern
  $a #Pattern ( \imp ph1 ph2 ) $.

axiom-1
  $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.

axiom-2
  $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
        ( \imp ( \imp ph1 ph2 )
               ( \imp ph1 ph3 ) ) ) $.

${
  rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
  rule-mp.1 $e |- ph1 $.
  rule-mp   $a |- ph2 $.
$}
```

```
imp-refl $p |- ( \imp ph1 ph1 )
$=
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern axiom-2
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  axiom-1 rule-mp ph1-is-pattern
  ph1-is-pattern axiom-1 rule-mp
$.
```

Matching logic syntax and proof system
Claims with proofs (machine checked)

The left column shows about 10% of it, including modus ponens. Very mathematical, almost identical to the proof system. We can also encode claims and their proof objects in Metamath, like in the right column above (the proof object for $\varphi \rightarrow \varphi$ is unusually small). We can similarly prove any claim. Anything that K does yields such a proof object, possibly having millions of steps. Very long and boring proofs. But very precise and low level. As we want them to be!

Basically, any true claim derived with K or any other formal systems or interactive theorem provers, in the end reduces to verifying a formal claim $\Gamma \vdash \varphi$ in Metamath: under given theory Γ, a given theorem φ is true because a proof object has been provided and has been checked.

Next we want to implement the Matching Logic proof checker as a ZK circuit, to yield succinct cryptographic proofs. Unfortunately, none of the Metamath implementations use PLs with ZK support. So we re-implemented Metamath in a Rust version supported by RiscZero's compiler. Executing this Metamath implementation on RiscZero's zkVM, we can now generate a succinct SNARK proof certificate that a mathematical proof object for the original claim exists.

This is work in progress, a collaboration with Tim Carstens and his RiscZero colleagues; also with my UIUC colleague Andrew Miller and his research group. Note that Andrew Miller is also a cryptography advisor in our company. The results so far are encouraging, with performance of SNARK proof generation currently the main concern. This is consistent with other ZK projects, which appear to all suffer from the same fundamental problem – slow ZK proof generation. We are now experimenting with RiscZero zkVM's nascent support for recursive STARKs.

We plan to do the same with all languages that produce ZK proofs: zkEVM, Cairo, zkLLVM. If any of these gives us an efficient Matching Logic proof checker, then we can leverage that and provide efficient ZK proofs for *all* languages! Ultimately, we plan to implement a ZK circuit directly for our 200 LOC checker, optimized for this one program. One circuit to rule them all! Such an efficient circuit will give us efficient ZK variants for all languages, from their semantics.