Pi Squared

A Logic-Programming Approach to Arithmetic Circuit Design

Deriving Zero-Knowledge Certificates from Mathematical Proofs


Research:
Brandon Moore


Presentation:
Traian Şerbănuţă


SuperVision:
Xiaohong Chen
CTO, Pi Squared


Vision:
Grigore Roşu
CEO, Pi Squared

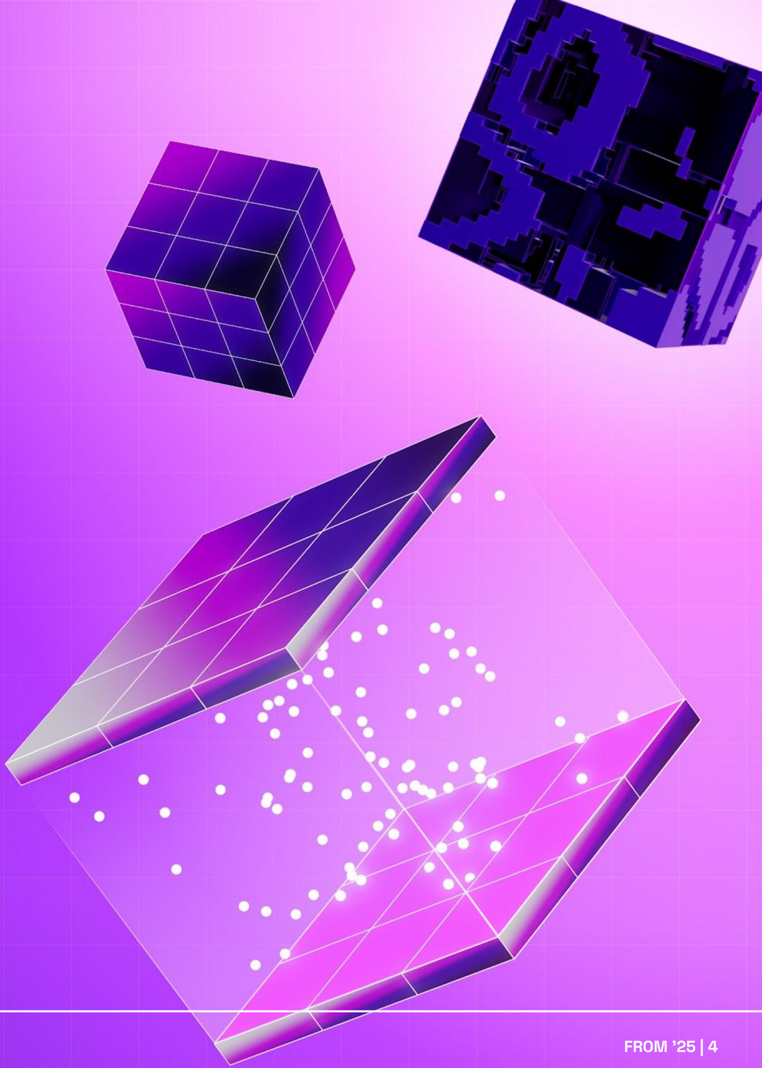
Contributors

- ZK team
  - Brandon Moore (the Block Model, overall design)
  - Mihai Calancea (original prototype)
  - B. Bailey, T. Şerbănuţă, N. Watson, P. Raduletu (R&D)
- Math Proof Generation team
  - D. Lucanu (pinning the ASCII syntax for the Block language)
- Xiaohong Chen (making sure we stay on task and deliver)

Plan of the talk

1. Vision
   - Certified Execution: mathematical proofs of program execution
2. Research
   - Background: Proofs and (zk)SNARKs
   - Adapting proofs for SNARKs: the BLOCK model
3. Example
   - Propositional logic using the BLOCK model
4. Implementation
   - Compiling blocks into circuits

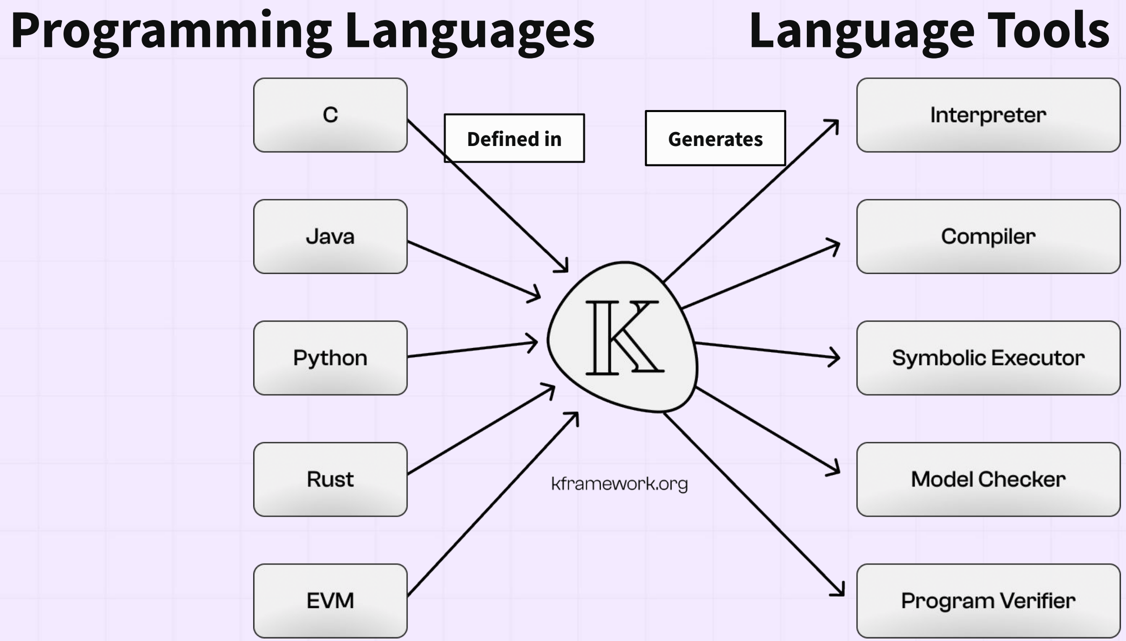
Vision



Pi Squared Web3 Vision

1. Program in any language
2. Settle any (zero knowledge) proof
3. Reach lightning fast (weak/generalized) consensus

Breaking Programming Language Barriers Using Formal Semantics



Separation of Concern

- Language design
- Tool implementation

Plug & Play your language

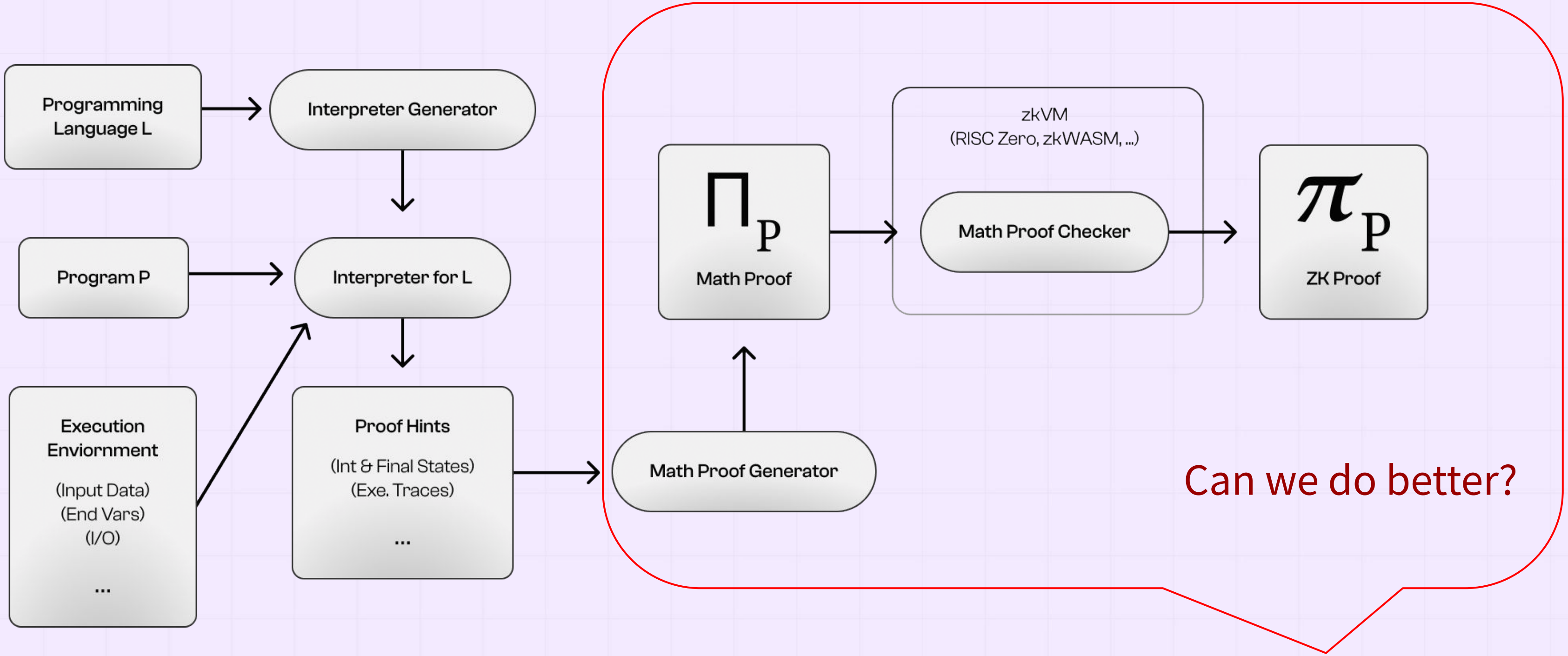
Correct by Construction

Pi^2 = Proof of Proof

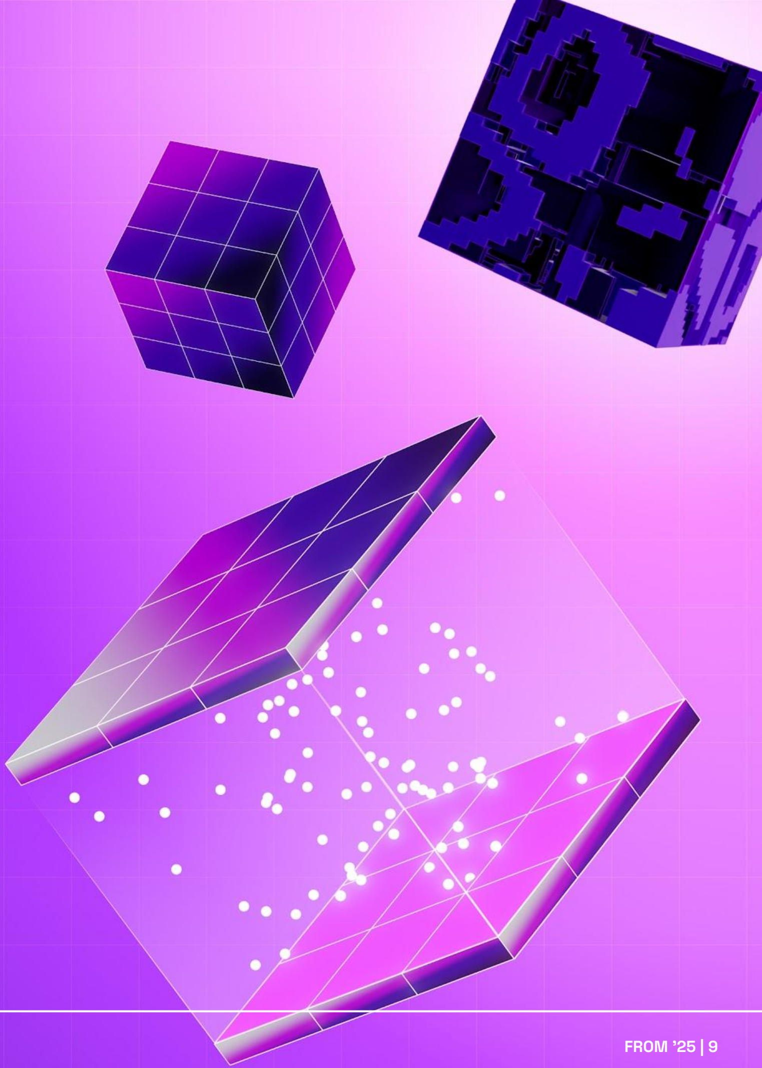
Zero Knowledge Proof

Mathematical Proof

Pi² (Proof of Proof) Workflow



Research



Background: Proofs and (zk)Snarks

SNARK for Mathematical Proofs

- Want efficient (zk)SNARK proof for validity of a mathematical proof
- A SNARK is a system for cryptographic “proofs” (aka “receipts”) of claims
  - About a relation R between “instances” and “witnesses”
  - Public input of a claim is the instance x. Claim is “I know a w with (x,w) in R”
  - Succinct: receipt small, efficiently checked
  - Noninteractive: receipt is a string checkable by anyone
  - ARgument: computational rather than absolute security
  - of Knowledge
- We call SNARK proofs “receipts” to distinguish from mathematical proofs

Proof and Circuit codesign

Plan:

- review the structure of mathematical proofs
- review the features of zkSNARKs
- restrict the allowed form of mathematical proof rules
  - to be efficiently checkable with zk circuits.

Pi Squared

Review Proof Structure

$$\dfrac{}{\vdash (A\to((B\to A)\to A))\to((A\to(B\to A))\to(A\to A))}\ \text{Ax2} \qquad \dfrac{}{\vdash A\to((B\to A)\to A)}\ \text{Ax1}$$

$$\dfrac{\vdash (A\to((B\to A)\to A))\to((A\to(B\to A))\to(A\to A)) \qquad \vdash A\to((B\to A)\to A)}{\vdash (A\to(B\to A))\to(A\to A)}\ \text{MP}$$

$$\dfrac{}{\vdash A\to(B\to A)}\ \text{Ax1}$$

$$\dfrac{\vdash (A\to(B\to A))\to(A\to A) \qquad \vdash A\to(B\to A)}{\vdash A\to A}\ \text{MP}$$

All hypotheses must equal a conclusion of a predecessor

Review Proof Rule Structure

- Rule are parameterized
- Lists of hypotheses and conclusions written using the parameters
- We call each hypothesis or conclusion a statement / claim
- Claims could be in different relations, e.g.,
  - φ is well-formed
  - x is free in φ
  - …

$$\frac{\vdash A \to B \qquad \vdash A}{\vdash B}\ \mathrm{MP}(A,B)$$

$$\frac{}{\vdash A \to (B \to A)}\ \mathrm{Ax1}(A,B)$$

$$\frac{}{\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))}\ \mathrm{Ax2}(A,B,C)$$

Review zkSNARK

- Primitive data elements of a finite field, usually $\mathbb{F}_p$ (some schemes $\mathbb{F}_{2^n}$)
- Native form of the instance and relation are vectors of field elements
- The relation is defined with arithmetic circuits or with polynomial constraints.
  - R1CS special case of degree 2 polynomials, also expresses circuits.
    - Constraints described by matrices A,B,C over the field.
    - Vector z formed from instance and witness (and a constant 1)
    - Check equation $(Az)\circ(Bz) = (Cz)$, where $\circ$ is element-wise product.

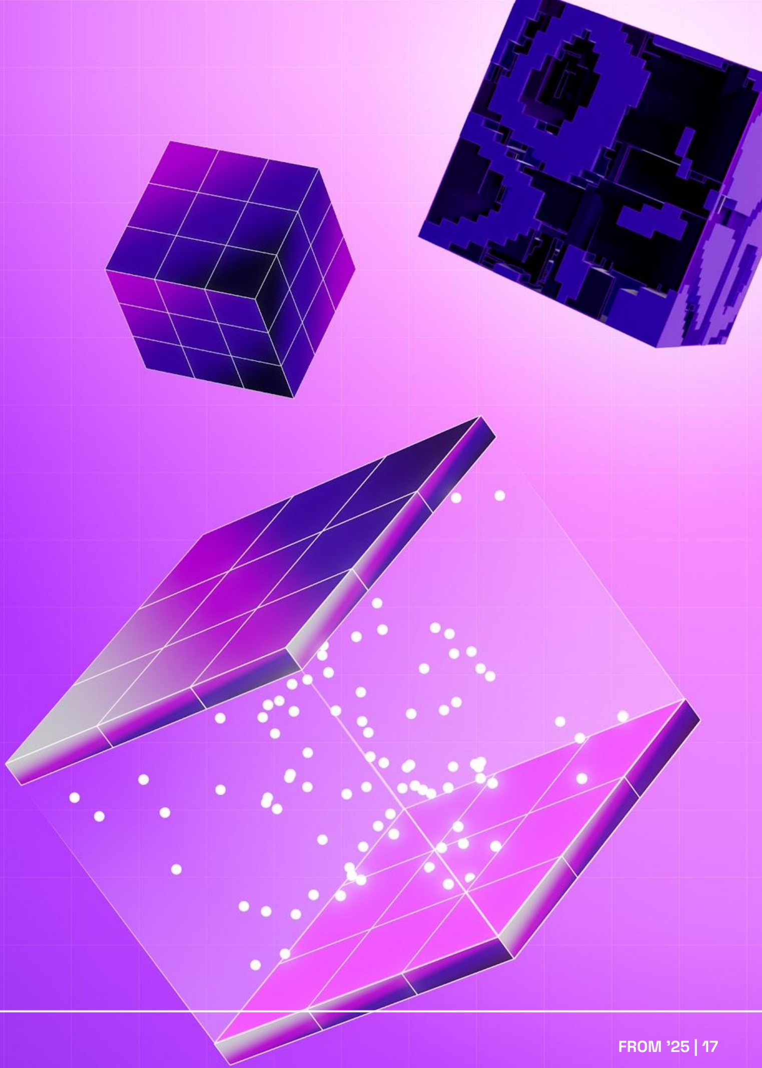
Review zkSNARK Randomization

- Access to “random” inputs through “Fiat-Shamir heuristic”
  - from public-coin interactive protocol to a non-interactive proof.
- With randomness we have permutation and lookup arguments
- Two lists of field elements $a_1,\ldots,a_n$ and $b_1,\ldots,b_m$
- Permutation argument enforces that lists are permutations
- Lookup argument enforces a subset relationship $\{a_i : i \in 1..n\} \subseteq \{b_j : j \in 1\ldots m\}$
- List elements are field elements, or easy generalization to fixed-size tuples

Permutation from Polynomials

- Permutation and lookup argument use polynomials, permutation is simple
- $\prod(a_i-x) - \prod(b_i-x)$ is a degree $O(n+m)$ polynomial in x
  - Uniformly 0 if the lists are permutations
  - Otherwise at most $O(n+m)$ roots, while usually $|\mathbb{F}|$ is very large
  - Just evaluate at a random value $\boldsymbol{\alpha}$ and require the result is zero
- Lookup uses similar ideas, more complicated expressions
- Both generalize to lists of fixed-size tuples of field elements
  - code tuple $(a_0,\ldots,a_k)$ as polynomial $a_0+a_1x+\ldots+a_kx^k$ evaluated at random $\boldsymbol{\beta}$

Research



Adapting proofs for SNARKs
The Blocks model

Adapting Proofs for SNARKs

- Translate instances of a proof rule into small section of witness or circuit.
- **Only** interaction between different proof steps is checking hypotheses are satisfied by other rule's conclusions. Adapt to use lookup arguments

- Need to flatten claims to tuples of atomic values / field elements
  - Handling terms: Must translate syntax of formulas to additional claims
- Problem: Lookup does not enforce DAG structure.
  - Solution: add "depth" to claims and extra hypotheses to proof rules

Breaking Cycles

- Add an additional depth argument to claims: $\vdash_k \varphi$ instead of $\vdash \varphi$
  - Can read $\vdash_k \varphi$ as "φ has a proof tree of depth at most k"

$$\frac{\vdash A \to B \qquad \vdash A}{\vdash B}\ \mathrm{MP}(A,B) \quad \Longrightarrow \quad \frac{\vdash_{k1} A \to B \qquad \vdash_{k2} A \qquad k1 < k,\ k2 < k}{\vdash_k B}\ \mathrm{MP}(A,B,k,k1,k2)$$

- Not all relations need a depth parameter
  - Proof rules might simply never depend on hypothesis of the same kind
  - Or rules emitting claims of that kind may only allow "structural recursion" so ensuring certain other things are acyclic is sufficient

Flattening Syntax

Eliminate explicit syntax in terms by

- Introducing extra relations about relating terms to immediate subterms
  - e.g. is_impl(T,A,B) means T represents term A→B
- Give proof rules extra arguments naming all terms and subterms, extra hypothesis using new relations. Now claim arguments are just variables
- (This is an independent transformation from adding depths, will do both)

$$\frac{\vdash A \rightarrow B \qquad \vdash A}{\vdash B}\ \mathrm{MP}(A,B) \quad \Rightarrow \quad \frac{\vdash T \qquad \vdash A \qquad \mathrm{is\_impl}(T,A,B)}{\vdash B}\ \mathrm{MP}(T,A,B)$$

Pi Squared

Flattening Syntax - Terms

- To use flattened rules, need syntax claims like is_impl(T,A,B)
- Flattened proof rules similar to use of Datalog for program analysis
  - there the syntax facts would be supplied as a preloaded table
- To fit the overall design, let rules emit these facts
- Attempt to define a rule

$$\frac{}{\texttt{is\_impl}(T,A,B)}\ \mathrm{DefImpl}(T,A,B)$$

Flattening Syntax - Terms

- Want to demand A,B to be terms; Need depths to prevent cyclic terms

$$\frac{\texttt{term}(A,ka) \quad \texttt{term}(B,kb) \quad ka<k \quad kb<k}{\texttt{term}(T,k) \quad \texttt{is\_impl}(T,A,B)}\ \mathrm{DefImpl}(T,A,B,k,ka,kb)$$

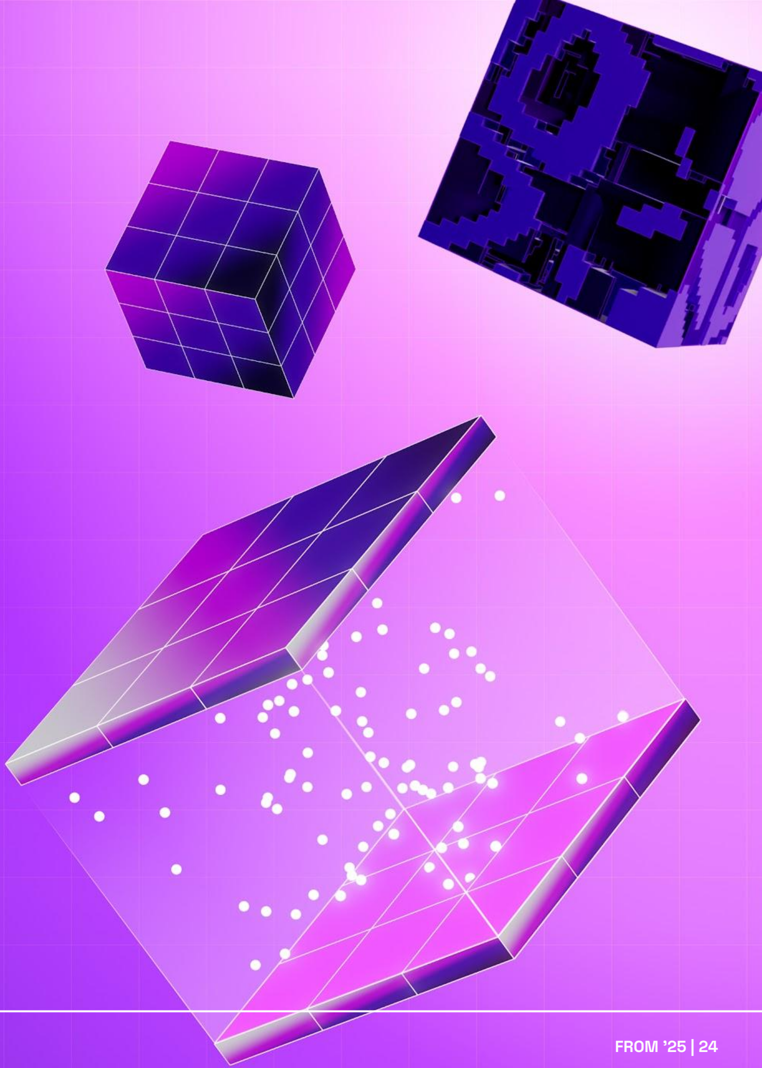
- But also need to prevent conflicting definitions.
  - Can’t allow both is_impl(T,A,A) and is_impl(T,C,D)

Unique Outputs

$$\frac{\texttt{term}(A,ka) \quad \mathrm{term}(B,kb) \quad ka<k \quad kb<k}{\texttt{UNIQUE termdef}(T) \quad \texttt{term}(T,k) \quad \texttt{is\_impl}(T,A,B)}\ \mathrm{DefImpl}(T,A,B,k,ka,kb)$$

- The `UNIQUE termdef`(T) is the unique output constraint
  - Will enforce that no other step in the proof has same unique output
- Now if we try to have both is_impl(T,A,A) and is_impl(T,C,D) with two instances of the DefImpl rule, the unique tags conflict
- Rules defining all other sorts of terms, such as conjunction will also have a `UNIQUE termdef`(T) unique output, with the same relation <`termdef`>

Example



Propositional Logic in the BLOCK model

Pi Squared

Example: ASCII Blocks definition

```
block def_term_bot(B):
    is_bot(B),
    UNIQUE wf_term(B), wf_term2(B, 0) -: .

block def_term_mvar(T, V):
    is_mvar(T, V),
    UNIQUE wf_term(T), wf_term2(T, 0) -: .

block def_term_impl(T, TA, TB, d, d_A, d_B):
    is_impl(T, TA, TB),
    UNIQUE wf_term(T), wf_term2(T, d)
    -: wf_term2(TA, d_A), wf_term2(TB, d_B),
       inc_max(d, d_A, d_B).
```

```
block axiom1(T; TA,TB,TI): // (TA -> (TB -> TA))
    proved2(T, 0)   -:        is_impl(TI,TB,TA), is_impl(T,TA,TI).

block axiom2(T; TA, TB, TC, THB, THC, TI, THI, TIH):
    proved2(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).

block modus_ponens(T; TA, TB, d, d_A, d_B):
    proved2(TB, d) -:
      is_impl(T, TA, TB), proved2(T, d_A), proved2(TA, d_B),
      inc_max(d, d_A, d_B).
```

Pi Squared

Example: $A \to A$ proof transcript

- Syntax construction of all used formulas.
- Last arguments of def_term_impl – depths

```
def_term_mvar(1, 0)            // v0 or A

def_term_impl(2, 1, 1,   1, 0, 0)  // A→A
def_term_impl(3, 1, 2,   2, 0, 1)  // A→(A→A)
def_term_impl(4, 1, 7,   3, 0, 2)  // A→((A→A)→A)
def_term_impl(5, 4, 6,   4, 3, 3)  // (A→((A→A)→A))→((A→(A→A))→(A→A))
def_term_impl(6, 3, 2,   3, 2, 1)  // (A→(A→A))→(A→A)
def_term_impl(7, 2, 1,   2, 1, 0)  // (A→A)→A
```

Instantiated blocks

```
block def_term_mvar(T, V):
    is_mvar(T, V),
    UNIQUE wf_term(T), wf_term2(T, 0) -: .

block def_term_impl(T, TA, TB, d, d_A, d_B):
    is_impl(T, TA, TB),
    UNIQUE wf_term(T), wf_term2(T, d)
    -: wf_term2(TA, d_A), wf_term2(TB, d_B),
       inc_max(d, d_A, d_B).
```

Example: $A \to A$ proof transcript

- Logical proof itself
- Last arguments of modus_ponens – depths

```
axiom1(3, 1, 1, 2)                 //  A→(A→A)
axiom1(4, 1, 2, 7)                 //  A→((A→A)→A)
axiom2(5, 1, 2, 1, 3, 2, 7, 4, 6)  // (A→((A→A)→A))→((A→(A→A))→(A→A))

modus_ponens(5, 4, 6,   1, 0, 0) // (A→(A→A))→(A→A)
modus_ponens(6, 3, 2,   2, 1, 0) // A→A
```

- Instantiated block:

```
block modus_ponens(T; TA, TB, d, d_A, d_B):
   proved2(TB, d) -:
      is_impl(T, TA, TB),   proved2(T, d_A),
      proved2(TA, d_B),   inc_max(d, d_A, d_B).
```

- Terms:

2: A→A   3: A→(A→A)   4: A→((A→A)→A)

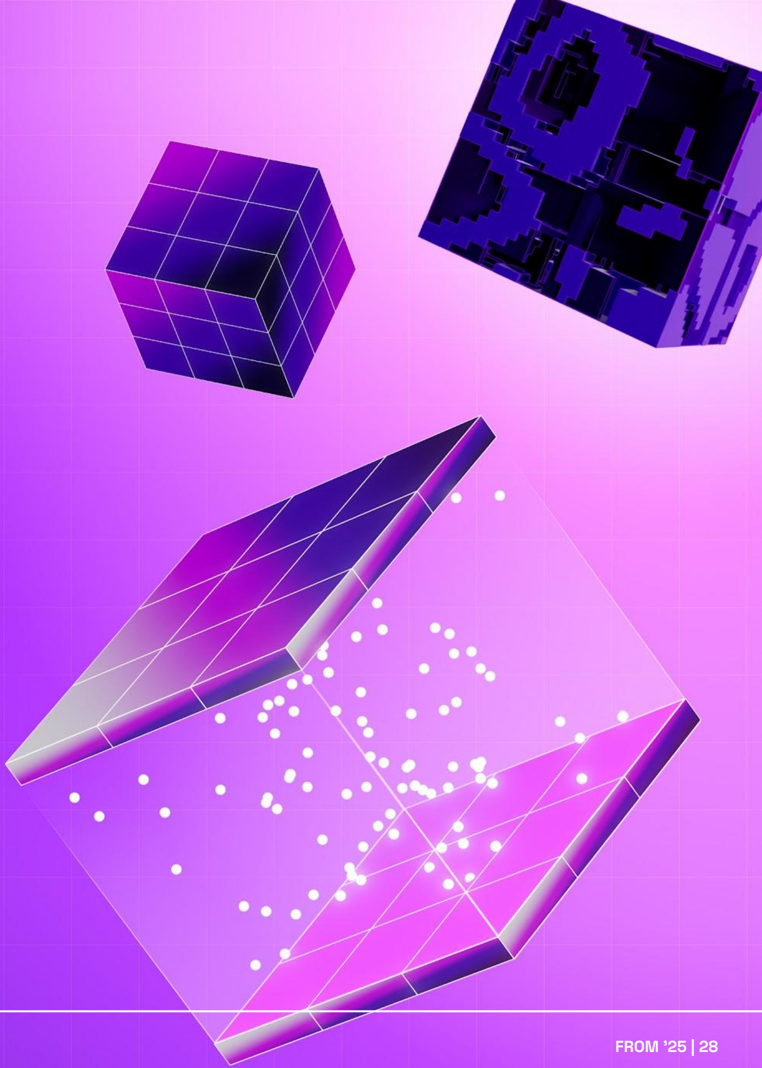
5: (A→((A→A)→A))→((A→(A→A))→(A→A))

6: (A→(A→A))→(A→A)

Pi Squared

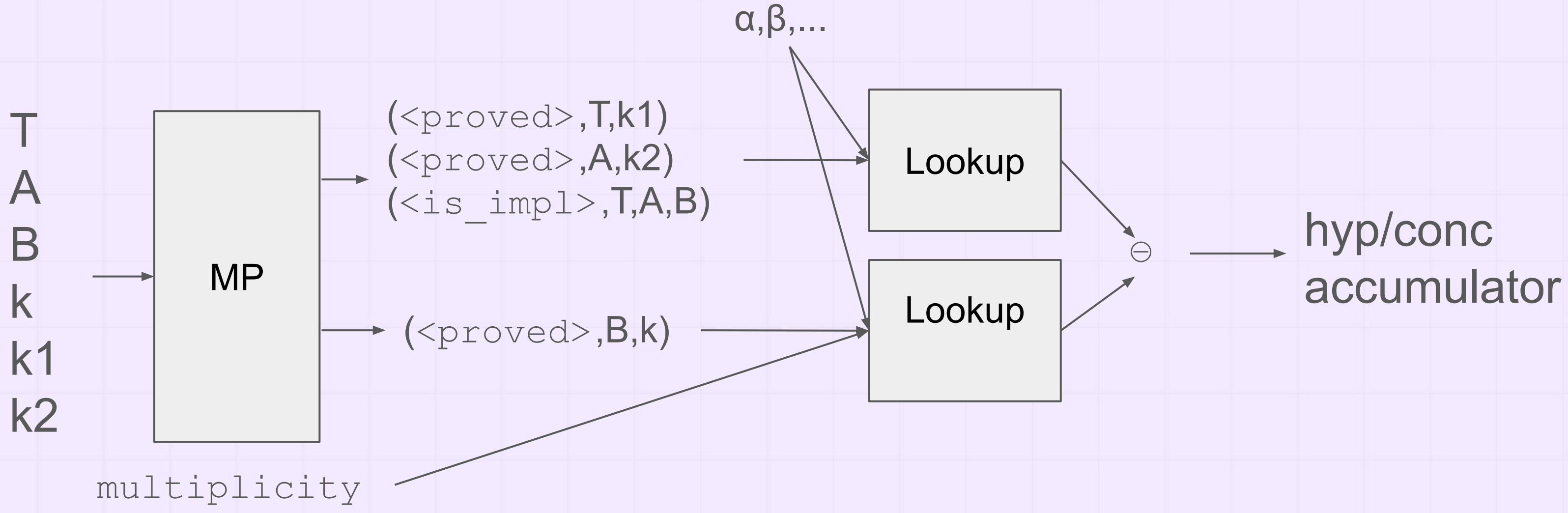
Implementation

Compiling blocks into circuits



Rules to Circuits

$$\frac{\texttt{proved}(T,k1)\quad \texttt{proved}(A,k2)\quad \texttt{is\_impl}(T,A,B)\quad k>k1\quad k>k2}{\texttt{proved}(B,k)}\ MP(T,A,B,k,k1,k2)$$

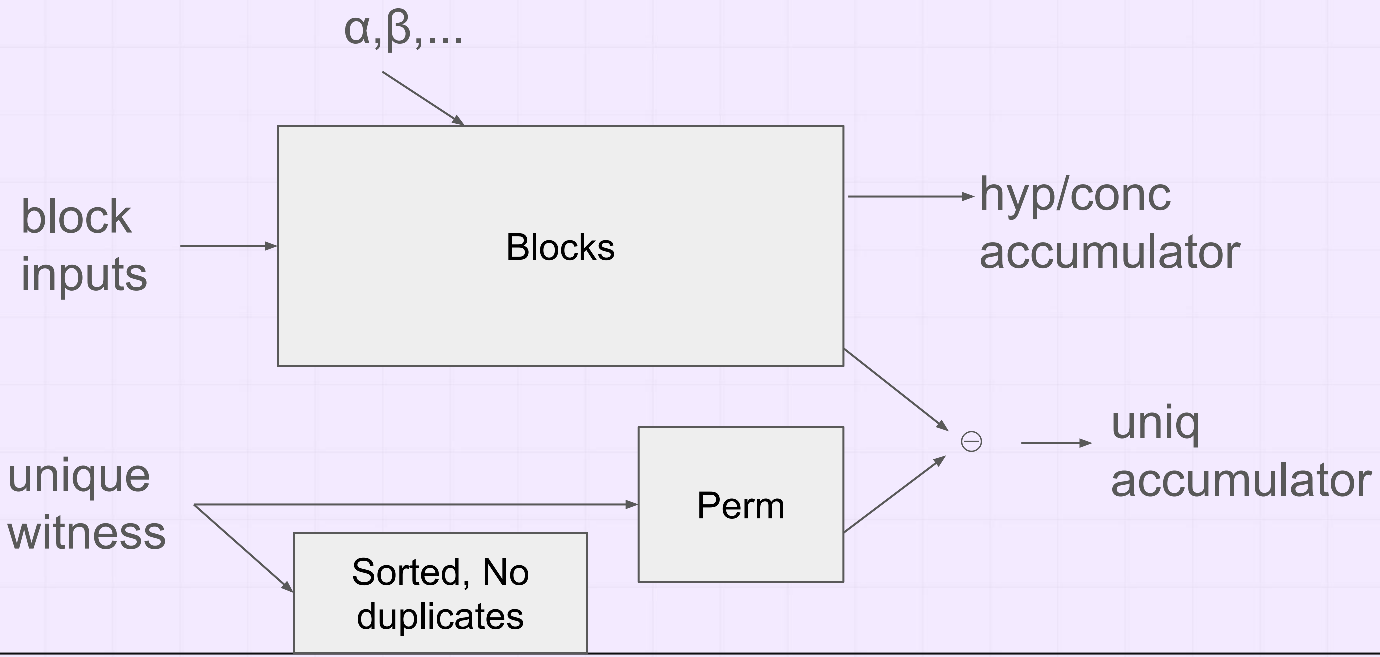


Uniqueness in circuits

- Unique outputs handled with a permutation argument
- Rule circuits output tuples as one side of a permutation argument
- Overall circuit has second witness input which is constrained to be a permutation of those outputs, and locally constrained to be sorted
- Then it is easy to check there are no duplicates
  - (except a specially allowed dummy element, if needed)

Segment Circuit

Many blocks can also be aggregated with similar small output



Folding (Nova style)

- Recursively aggregate multiple R1CS instances while preserving the structure
- Standard R1CS: $(Az)\circ(Bz) = (Cz)$, where $z = (1, x, w)$
- Relaxed R1CS: $(Az)\circ(Bz) = u(Cz) + E$
  - u scalar; E - *error vector* to absorb extra cross-terms when doing folding
- Given $(A,B,C),(E_1,u_1,x_1)$ with witness W_1, and $(A,B,C),(E_2,u_2,x_2)$ with witness W_2
  - With new random scalar r, and with $z_i=(1, x_i, w_i)$, compute:
    - $u=u_1 + r\,u_2$, $E=E_1+r((Az_1)\circ(Bz_2)+(Az_2)\circ(Bz_1) - u_1(Cz_2)-u_2(Cz_1)) + r^2E_2$
  - Then $z=(1,x_1+r\,x_2,w_1+r\,w_2)$ satisfies $(Az)\circ(Bz) = u(Cz) + E$

Optimization problem

- Each segment must have the exact same number of blocks of given type
  - Let r_i (to be determined) be the ratio of blocks of type i
  - It must be that $\sum_i r_i = 1$
- Public segments are separated from private segments
  - Let p_i (q_i) be the ratio of public (private) segments in a proof transcript
  - We have $\sum_i (p_i + q_i) = 1$
- We want to minimize the total number of segments, i.e., minimize
  - $\max_i (p_i / r_i) + \max_i (q_i / r_i)$

Conclusions

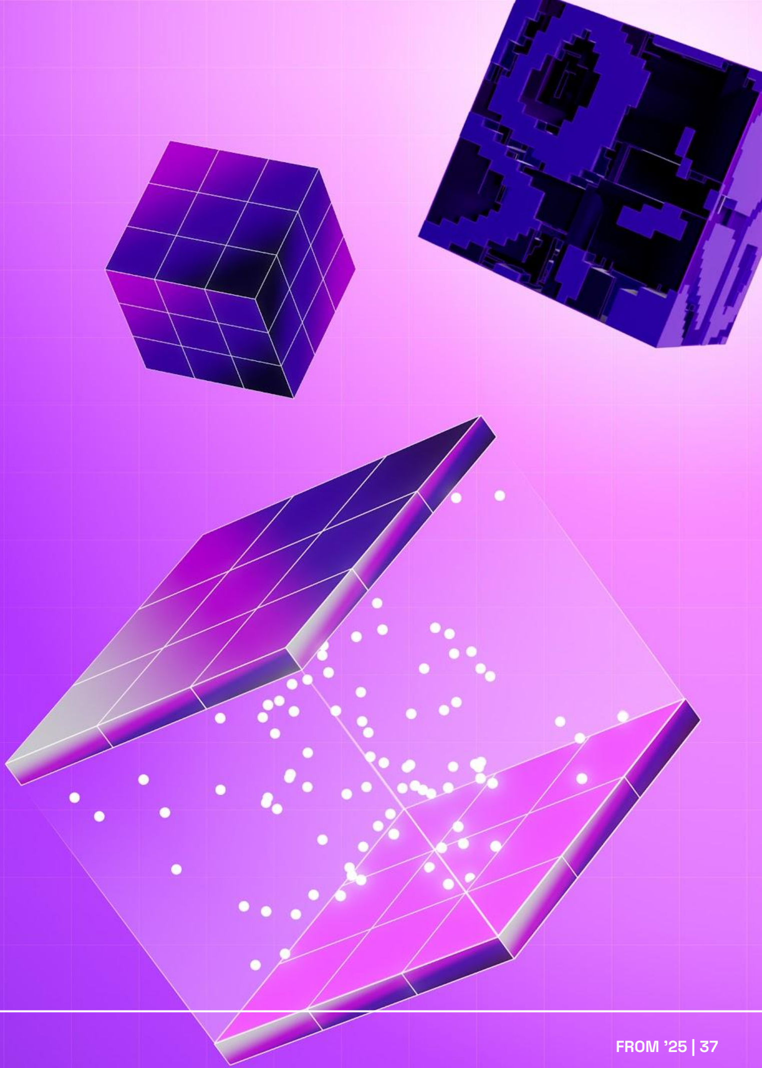
- We have defined and implemented a logic language for generating zkSNARKs
- Suitable for most logical inference-like problems
  - like math proofs, but not limited to that
- It is definitely a better solution than running proof verifiers on top of zkVMs
  - Its performance is close to handcrafted circuits for particular problems

Thank you!

Questions?

References

- Pi Squared Inc. Whitepaper
- Justin Thaler, Proofs, Arguments, and Zero-Knowledge
- Ulrich Haböck, Multivariate lookups based on logarithmic derivatives
- Abhiram Kothapalli, Srinath Setty, Ioanna Tzialla, Nova: Recursive Zero-Knowledge Arguments from Folding Schemes



Case study

[zkUNSAT]() using the block model

Refutation through resolution

- Clauses: 1: $x1 \lor x2$; 2: $\neg x1 \lor x2$; 3: $x1 \lor \neg x2$; 4: $\neg x1 \lor \neg x2$
  - Encoding 1: 1 2; 2: -1 2; 3: 1 -2; 4: -1 -2

Refutation:

- resolution between 1 and 3 using x2 resulting in 5: x1
- resolution between 4 and 2 using ¬x2 resulting in 6: ¬x1
- resolution between 5 and 6 using x1 resulting in 7: $\perp$

Literals

```
block def_lit(X,NX):
    lit_negation(X,NX),
    lit_negation(NX,X),
    UNIQUE is_lit(X),
    UNIQUE is_lit(NX)
    -:
    .
```

Pi Squared

Clauses as lists of literals

```
block declare_clause(L;K):
   clause(L)
   -:
   ne_list(L,K)
   .

block def_list_empty(L):
   is_empty(L),    UNIQUE list(L)
   -:
   .

block def_list_singleton(L,X):
   is_singleton(L,X),    ne_list(L,1),    UNIQUE list(L)
   -:
   .

block def_list_app(L,L1,L2;K,K1,K2):
   is_ne_app(L,L1,L2),   ne_list(L,K),    UNIQUE list(L)
   -:
   ne_list(L1,K1),       ne_list(L2,K2),  add(K,K1,K2)
   .
```

Pi Squared

Resolution

```
block resolve(L,L1,X,L1a,L2,NX,L2a):
    clause(L)
    -:
    clause(L1),
    clause(L2),
    lit_negation(X,NX),
    remove_lit(L1a,L1,X),
    remove_lit(L2a,L2,NX),
    is_app(L,L1a,L2a)
    .
```

```
block goal(L):
  -:
  is_empty(L),
  clause(L)
  .
```

Removing a literal

```
block remove_singleton_eq(La, L,X):
    remove_lit(La, L,X)
    -:
    is_singleton(L,X),
    is_empty(La)
    .

block remove_keep(L, X):
    remove_lit(L, L, X)
    -:
    .
```

```
block remove_app(La, L,X,L1,L2,L1a,L2a):
    remove_lit(La,L,X)
    -:
    is_ne_app(L,L1,L2),
    remove_lit(L1a, L1,X),
    remove_lit(L2a, L2,X),
    is_app(La,L1a,L2a)
    .
```

List append as a predicate

```
block is_app_empty(L0):
    is_app(L0,L0,L0)
    -:
    is_empty(L0)
    .

block is_app_nonempty(L,L1,L2):
    is_app(L,L1,L2)
    -:
    is_ne_app(L,L1,L2)
    .
```

```
block is_app_empty_left(L,L0,K):
 is_app(L,L0,L)
 -:
 is_empty(L0),
 ne_list(L,K)
 .
block is_app_empty_right(L,L0,K):
 is_app(L,L,L0)
 -:
 is_empty(L0),
 ne_list(L,K)
 .
```

Example transcript

```
def_lit(1,2)
def_lit(3,4)
def_list_empty(1)
def_list_singleton(2,1)
def_list_singleton(3,3)
def_list_app(4, 2, 3, 2, 1, 1)
declare_clause(4,2)
def_list_singleton(5, 2)
def_list_app(6, 5, 3, 2, 1, 1)
declare_clause(6,2)
def_list_singleton(7, 4)
def_list_app(8, 2, 7, 2, 1, 1)
declare_clause(8,2)
```

```
def_list_app(9, 5, 7, 2, 1, 1)
declare_clause(9,2)

remove_singleton_eq(1, 3, 3)
remove_keep(2, 3)
is_app_empty_right(2, 1, 1)
remove_app(2, 4, 3, 2, 3, 2, 1)
remove_singleton_eq(1, 7, 4)
remove_keep(2, 4)
remove_app(2, 8, 4, 2, 7, 2, 1)
def_list_app(10, 2, 2, 2, 1, 1)
is_app_nonempty(10, 2, 2)
resolve(10, 4, 3, 2, 8, 4, 2)
```

```
remove_singleton_eq(1, 3, 3)
remove_keep(5, 3)
is_app_empty_right(5, 1, 1)
remove_app(5, 6, 3, 5, 3, 5, 1)
remove_singleton_eq(1, 7, 4)
remove_keep(5, 4)
remove_app(5, 9, 4, 5, 7, 5, 1)
def_list_app(11, 5, 5, 2, 1, 1)
is_app_nonempty(11, 5, 5)
resolve(11, 6, 3, 5, 9, 4, 5)
```

```
remove_singleton_eq(1, 2, 1)
remove_app(1, 10, 1, 2, 2, 1, 1)
remove_singleton_eq(1, 5, 2)
remove_app(1, 11, 2, 5, 5, 1, 1)
resolve(1, 10, 1, 1, 11, 2, 1)
```

Comparison with zkUNSAT

- Our solution is ~ 1.2 - 4.3 slower than zkUNSAT. However,
- Our solution is generic, generated from a particular block model for refutation
  - same solution could be applied to many other problems
- zkUNSAT uses an interactive algorithm
  - there are known to be faster than non-interactive ones
  - but are not suitable for generating zk receipt