# PLINK: Verified Generation of Constraints for PLONK

Pablo Castellanos, Ignacio Cascudo, Dario Fiore, Niki Vazou
IMDEA Software Institute
25th March, 2025

# Arithmetization is hard

- General-purpose ZK protocols, as PLONK, allow me to prove I know *w* such that

  $$R(x,w,y) = \textbf{true}$$

  for any NP relation *R* and known (public) values *x* and *y* …

# Arithmetization is hard

- General-purpose ZK protocols, as PLONK, allow me to prove I know *w* such that

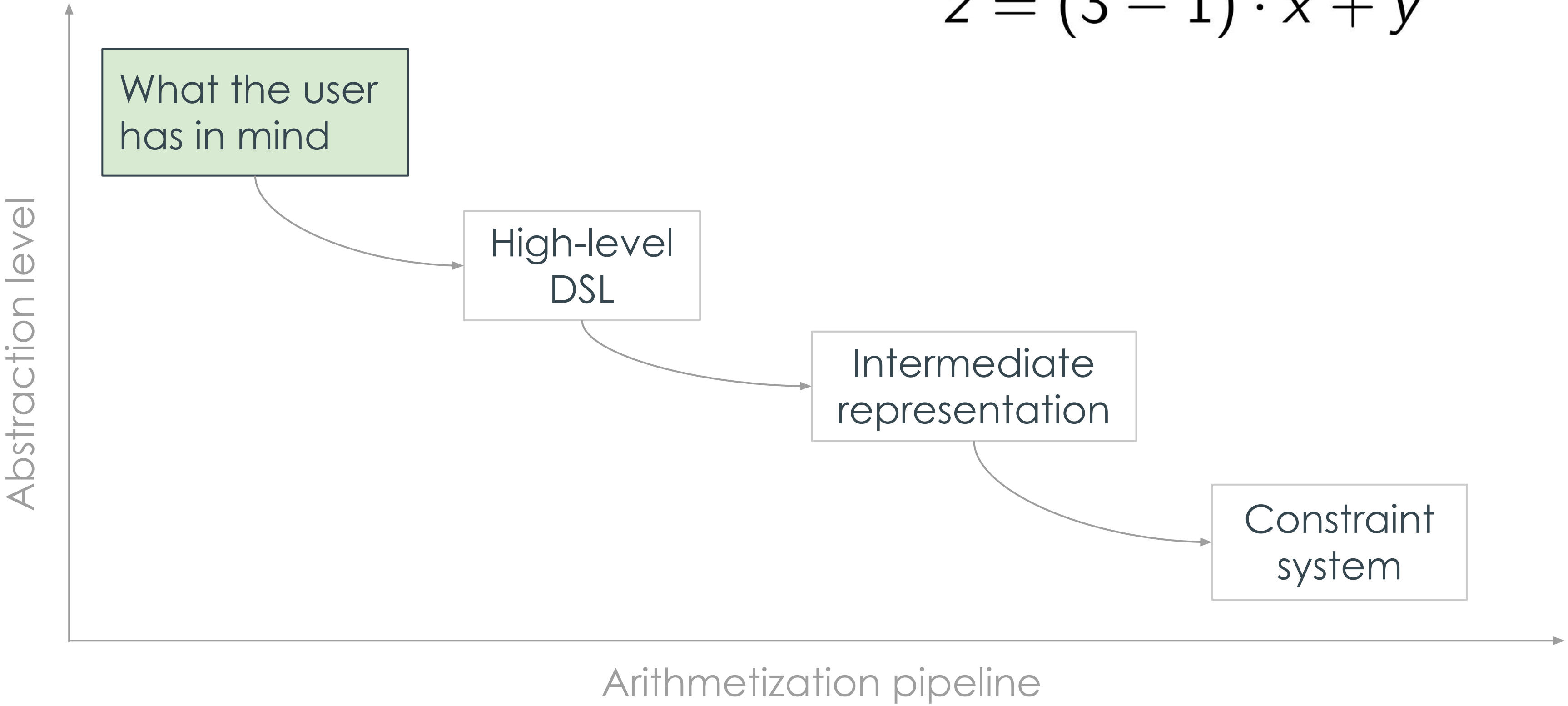
  *R(x,w,y)* = **true**

  for any NP relation *R* and known (public) values x and *y* …
- … but that relation needs to be encoded as a system of equations.

$$\begin{cases} 1 \cdot x_4 + 1 \cdot x_5 - 1 \cdot x_6 + 0 \cdot x_4 \cdot x_5 + 0 = 0 \\ 0 \cdot x_2 + 0 \cdot x_3 - 1 \cdot x_4 + 1 \cdot x_2 \cdot x_3 + 0 = 0 \\ 1 \cdot x_2 + 1 \cdot x_1 - 1 \cdot x_0 + 0 \cdot x_2 \cdot x_1 + 0 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_0 + 0 \cdot x_0 \cdot x_0 + 3 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_1 + 0 \cdot x_0 \cdot x_0 + 1 = 0 \end{cases}$$

# Arithmetization: overview

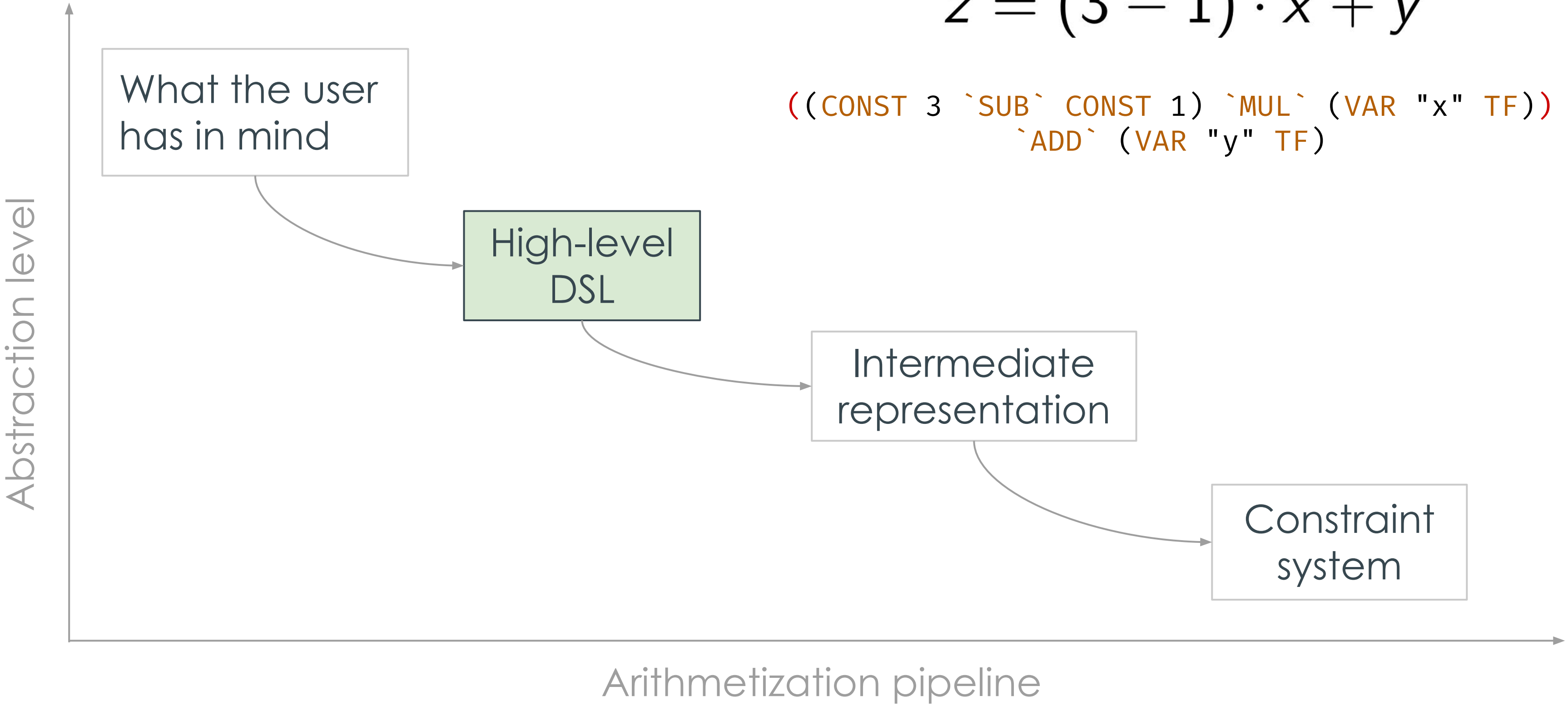
$$z = (3 - 1) \cdot x + y$$



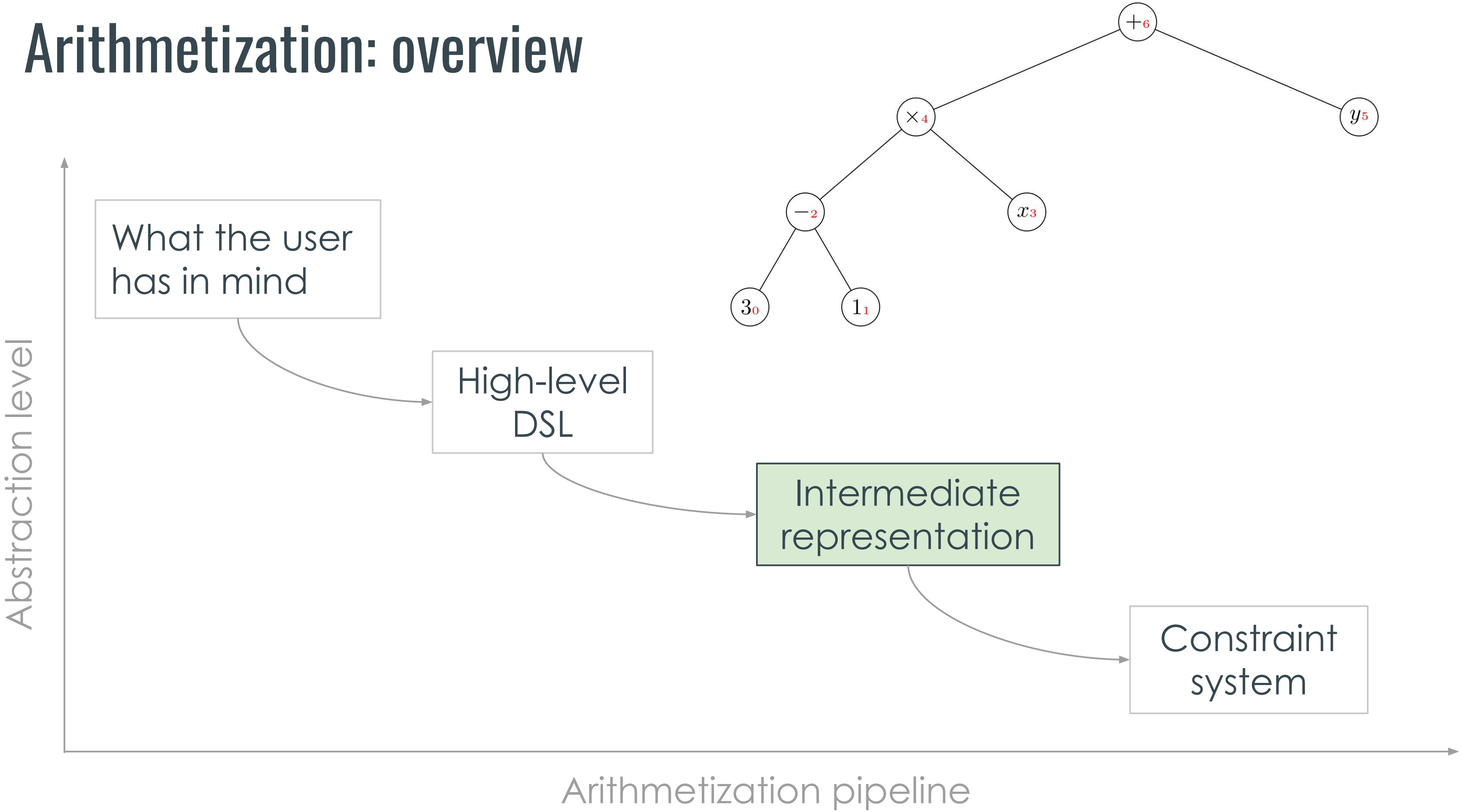
# Arithmetization: overview

$$z = (3 - 1) \cdot x + y$$

```
((CONST 3 `SUB` CONST 1) `MUL` (VAR "x" TF))
    `ADD` (VAR "y" TF)
```

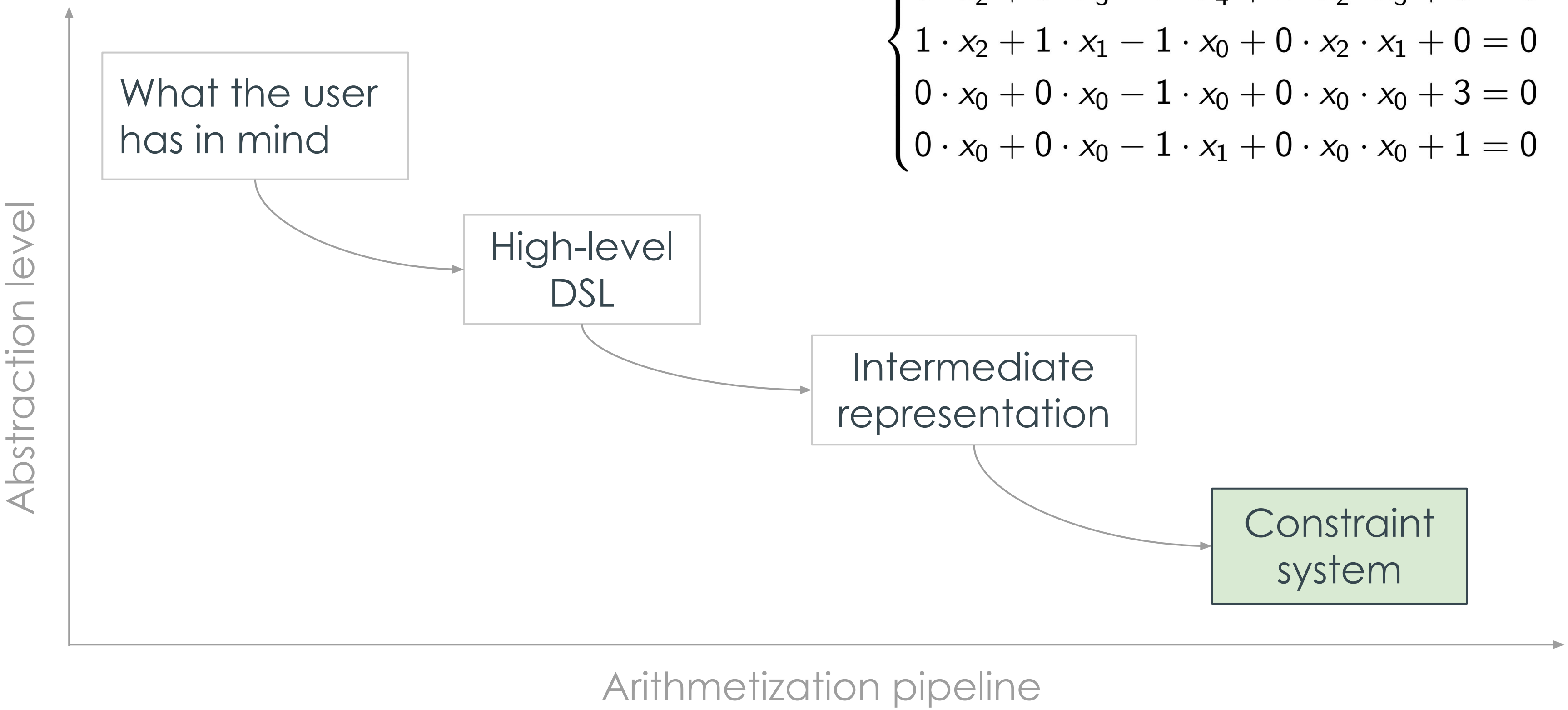


# Arithmetization: overview



# Arithmetization: overview

$$
\begin{cases}
1 \cdot x_4 + 1 \cdot x_5 - 1 \cdot x_6 + 0 \cdot x_4 \cdot x_5 + 0 = 0 \\
0 \cdot x_2 + 0 \cdot x_3 - 1 \cdot x_4 + 1 \cdot x_2 \cdot x_3 + 0 = 0 \\
1 \cdot x_2 + 1 \cdot x_1 - 1 \cdot x_0 + 0 \cdot x_2 \cdot x_1 + 0 = 0 \\
0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_0 + 0 \cdot x_0 \cdot x_0 + 3 = 0 \\
0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_1 + 0 \cdot x_0 \cdot x_0 + 1 = 0
\end{cases}
$$



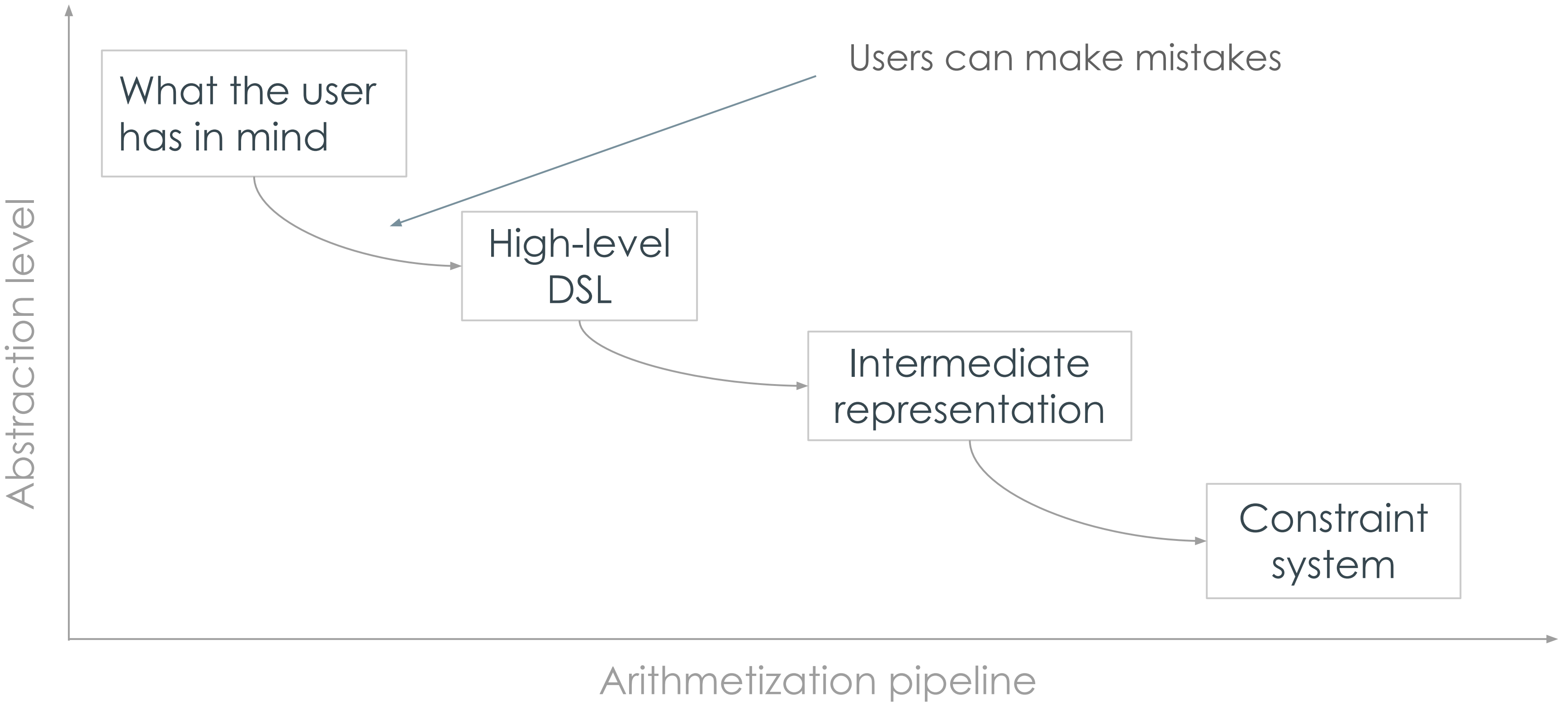
# Arithmetization is hard

- DSLs give a higher-level description of the low-level constraint system.
- Still, that may not be enough:
    - DSLs often are still quite **low-level** and lack common “safety” features like **type systems**.
    - Compilers can be **buggy**, generating incorrect circuits even from correct programs.

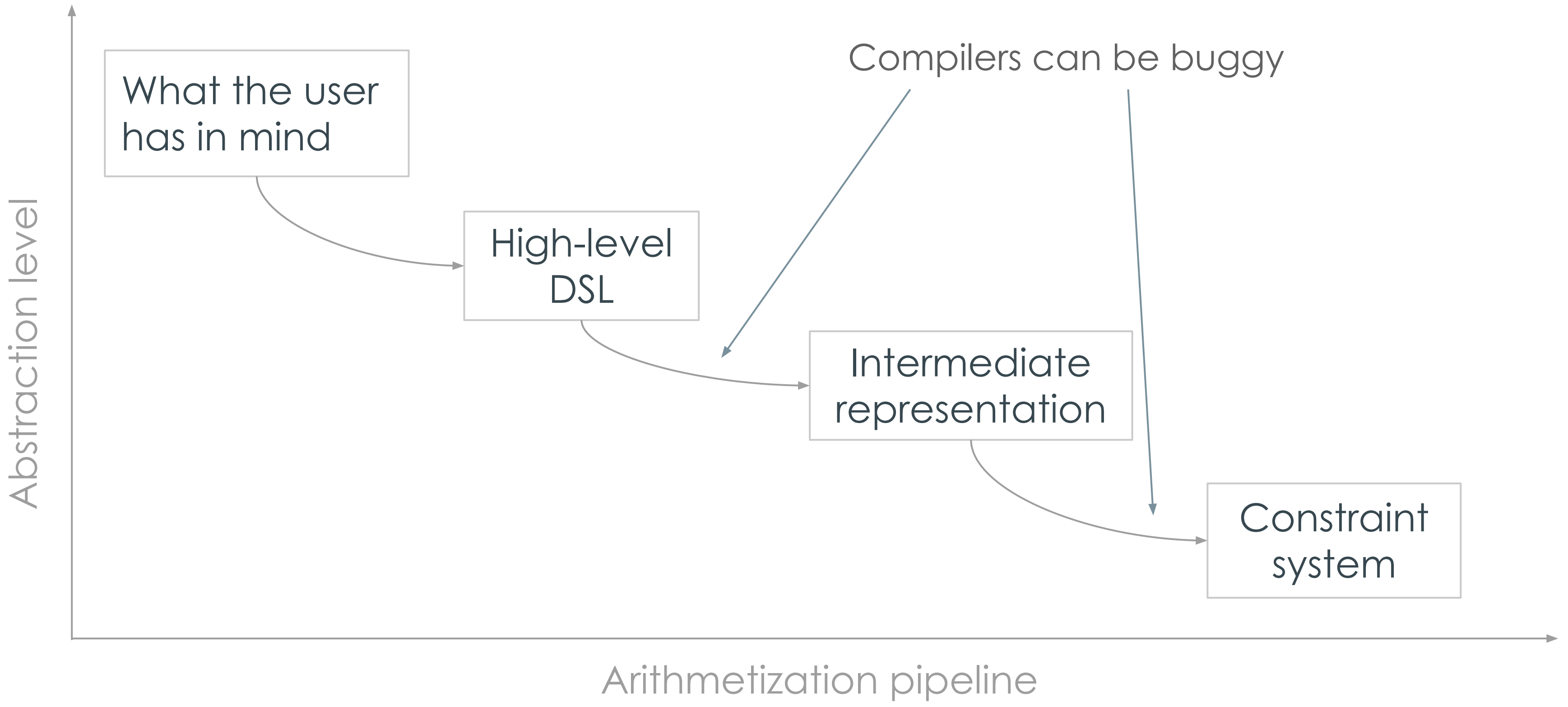
**SoK: What Don’t We Know? Understanding Security Vulnerabilities in SNARKs**

Stefanos Chaliasos *Imperial College London*
Jens Ernstberger
David Theodore *Ethereum Foundation*
David Wong *zkSecurity*
Mohammad Jahanara *Scroll Foundation*
Benjamin Livshits *Imperial College London & Matter Labs*

# Arithmetization: what can go wrong?



# Arithmetization: what can go wrong?



# Formal methods to the rescue

- Formal techniques give us higher confidence in the correctness of software.
- Promising topic in ZK:

| DSL | Target CS | Written in | Verification | Verified using |
|---|---|---|---|---|
| Leo [arXiv'23] | R1CS | Rust | Compiler + R1CS | ACL2 |
| Coda [S&P'24] | R1CS | OCaml (eDSL) | HL program | Coq |
| Clap [ZKProof'24] | PLONKish | Rust (eDSL) | Compiler | Agda |

# Formal methods to the rescue

- Formal techniques give us higher confidence in the correctness of software.
- Promising topic in ZK:

| DSL | Target CS | Written in | Verification | Verified using |
|---|---|---|---|---|
| Leo [arXiv'23] | R1CS | Rust | Compiler + R1CS | ACL2 |
| Coda [S&P'24] | R1CS | OCaml (eDSL) | HL program | Coq |
| Clap [ZKProof'24] | PLONKish | Rust (eDSL) | Compiler | Agda |
| **PLINK (This work)** | **PLONK** | **Liquid Haskell (eDSL)** | **Compiler + HL program** | **Liquid Haskell** |

# Formal methods: Liquid Haskell

```haskell
{-@ fib :: {n:Int | n ≥ 0} → {f:Int | f ≥ n} @-}
fib :: Int → Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- *Refinement type* checker for Haskell.
- A stronger type system means it can catch more errors *at compile time*.
- With appropriate type signatures, we can use it to prove theorems:
    - Proofs are aided by an SMT solver.

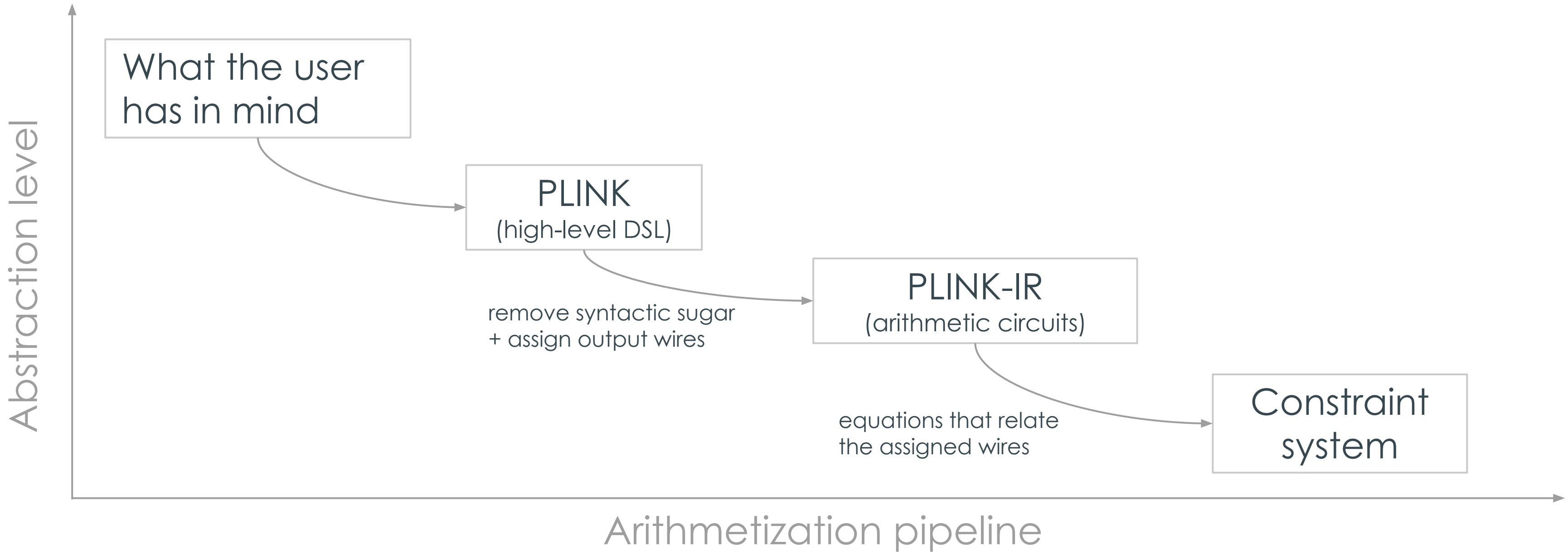
```haskell
{-@ fibMonotonic :: x:Nat → y:{Nat | x < y}
                 → { fib x ≤ fib y } @-}
```

# PLINK

## Programming Language for INtegrity and Knowledge

# PLINK

- Embedded in (Liquid) Haskell
- Declarative language with types (bool, field, vectors)
- Witness generator (calculation of intermediate values)



# Example: modular addition

Given $x$ and $y$ , how to encode $z = x + y \pmod{2^e}$?

- New boolean variable $b$ that represents the "overflow".
- Add equality constraint $x + y = z + b \cdot 2^e$ .
- Add *in*equality constraint enforcing $0 \leq z < 2^e$ .

# Example: modular addition

Given $x$ and $y$ , how to encode $z = x + y \pmod{2^e}$?

- New boolean variable $b$ that represents the "overflow".
- Add equality constraint $x + y = z + b \cdot 2^e$ .
- Add *in*equality constraint enforcing $0 \leq z < 2^e$ .
  - Equivalently, $z$ can be encoded using $e$ bits.

# Example: modular addition

```
{-@ addMod :: {n:Nat | n ≥ 1}
           → x:FieldDSL p → y:FieldDSL p
           → GlobalStore p (z:FieldDSL p) @-}
addMod :: Field p ⇒ Int → DSL p → DSL p → GlobalStore p (DSL p)
addMod e x y = do
  let modulus = 2^e

  let b = VAR "overflow" TF
  let z = VAR "sum" TF

  witnessGenHint b (\x y → if x + y < modulus then 0 else 1) x y
  witnessGenHint z (\x y → (x + y) `mod` modulus) x y

  assert $ BOOL b
  assert $ (x `ADD` y) `EQA` (z `ADD` (b `MUL` CONST modulus))
  toBinary e z -- z can be encoded using ‘e’ bits
  return z
```

Add constraints

# Example: modular addition

```
{-@ addMod :: {n:Nat | n ≥ 1}
           → x:FieldDSL p → y:FieldDSL p
           → GlobalStore p (z:FieldDSL p) @-}
addMod :: Field p ⇒ Int → DSL p → DSL p → GlobalStore p (DSL p)
addMod e x y = do
  let modulus = 2^e

  let b = VAR "overflow" TF
  let z = VAR "sum" TF

  witnessGenHint b (\x y → if x + y < modulus then 0 else 1) x y
  witnessGenHint z (\x y → (x + y) `mod` modulus) x y

  assert $ BOOL b
  assert $ (x `ADD` y) `EQA` (z `ADD` (b `MUL` CONST modulus))
  toBinary e z -- z can be encoded using ‘e’ bits
  return z
```

Declare variables

Add constraints

# Example: modular addition

```haskell
{-@ addMod :: {n:Nat | n ≥ 1}
           → x:FieldDSL p → y:FieldDSL p
           → GlobalStore p (z:FieldDSL p) @-}
addMod :: Field p ⇒ Int → DSL p → DSL p → GlobalStore p (DSL p)
addMod e x y = do
  let modulus = 2^e

  let b = VAR "overflow" TF
  let z = VAR "sum" TF

  witnessGenHint b (\x y → if x + y < modulus then 0 else 1) x y
  witnessGenHint z (\x y → (x + y) `mod` modulus) x y

  assert $ BOOL b
  assert $ (x `ADD` y) `EQA` (z `ADD` (b `MUL` CONST modulus))
  toBinary e z -- z can be encoded using ‘e’ bits
  return z
```

Declare variables

Give hints to the witness generator

Add constraints

# Example: modular addition

```
{-@ type FieldDSL p =
      {v:DSL p | typed v TF} @-}
```

Add types

```
{-@ addMod :: {n:Nat | n ≥ 1}
           → x:FieldDSL p → y:FieldDSL p
           → GlobalStore p (z:FieldDSL p) @-}
addMod :: Field p ⇒ Int → DSL p → DSL p → GlobalStore p (DSL p)
addMod e x y = do
  let modulus = 2^e

  let b = VAR "overflow" TF
  let z = VAR "sum" TF

  witnessGenHint b (\x y → if x + y < modulus then 0 else 1) x y
  witnessGenHint z (\x y → (x + y) `mod` modulus) x y

  assert $ BOOL b
  assert $ (x `ADD` y) `EQA` (z `ADD` (b `MUL` CONST modulus))
  toBinary e z -- z can be encoded using 'e' bits
  return z
```

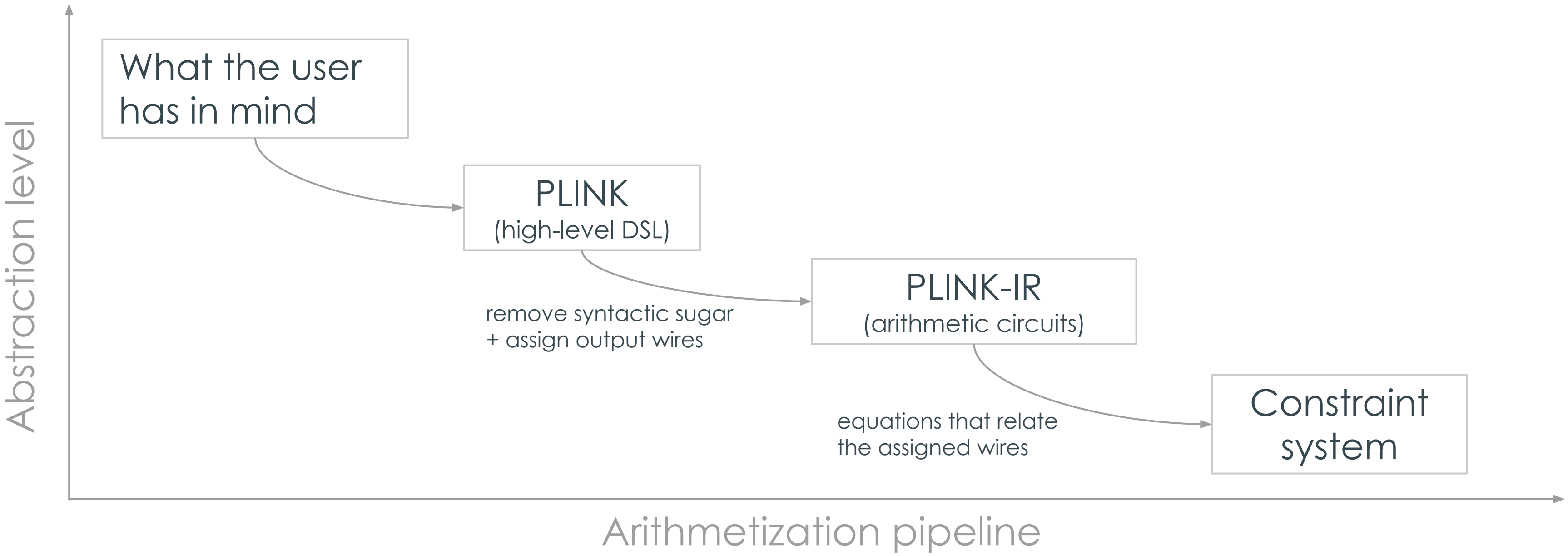
Declare variables

Give hints to the witness generator

Add constraints

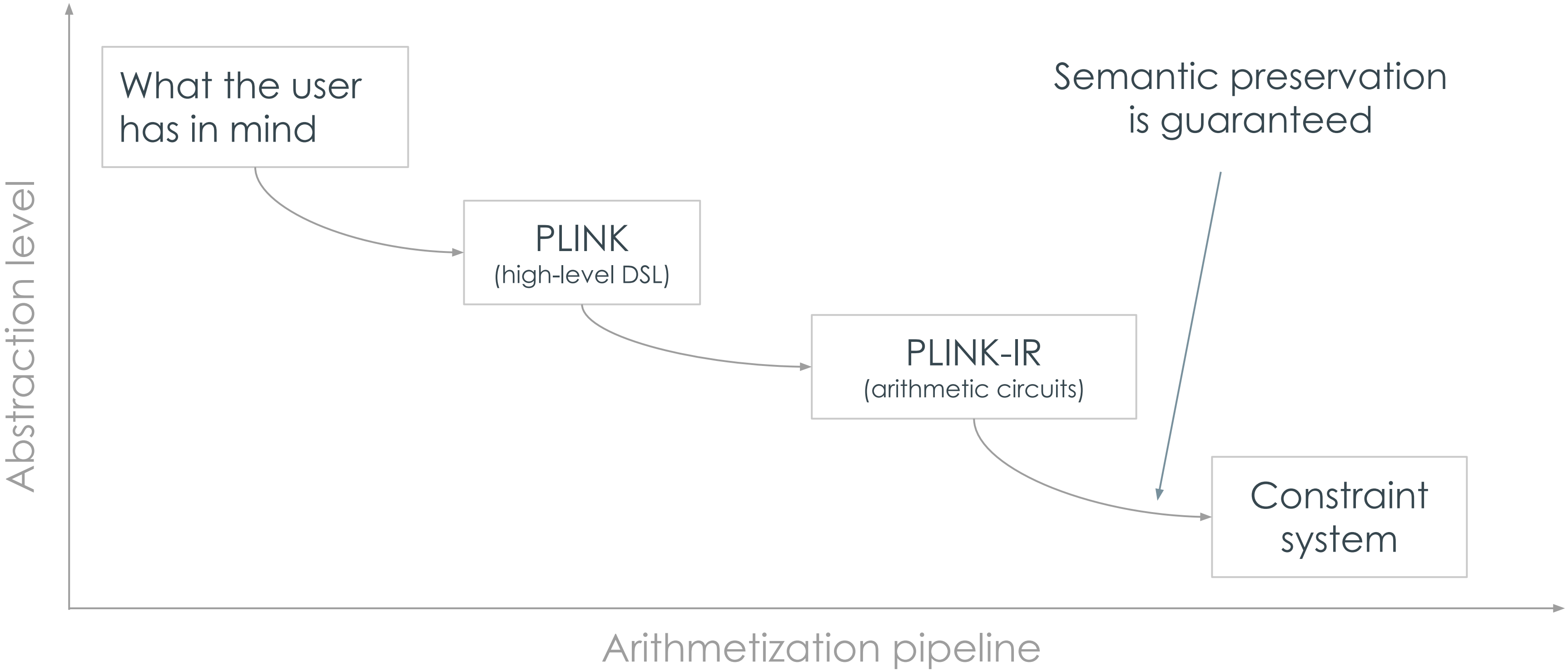
# PLINK

- Embedded in (Liquid) Haskell
- Declarative language with types (bool, field, vectors)
- Witness generator (calculation of intermediate values)



# Security guarantees

# Semantics is preserved



# Semantics is preserved

- **Theorem**: Let *CS* be the set of PLONK constraint systems. Then, for each program $P \in$ *PLINK-IR* and valuation σ of its variables, we have

  $$\sigma \text{ satisfies } P \iff \sigma \text{ is a solution to } C(P),$$

  where $C : \mathit{PLINK\text{-}IR} \to \mathit{CS}$ is the compilation function.

  - Needs the **IR** to refer to the **intermediate values** (through their wires).
  - **Proven** about the Haskell **implementation** *itself* using Liquid Haskell.
  - Completely **modular** (e.g. in case we want to add a new instruction).

# Additional guarantees

- With LH, we can prove some properties about programs implemented in PLINK.
- Example: **padding** function in SHA-256 pre-processing returns a bit-vector with a "valid" length (multiple of 512):

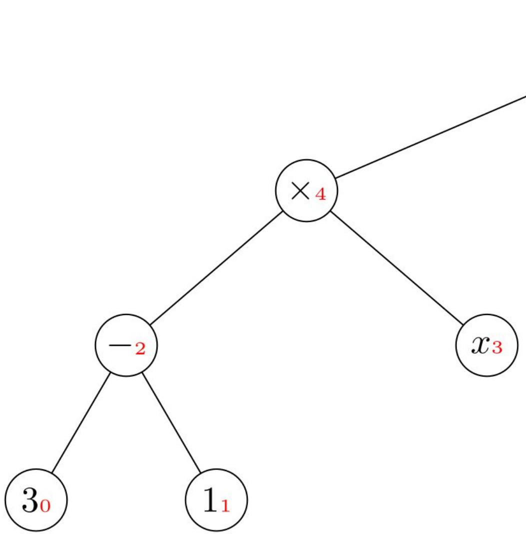
```
{-@ padding :: msg:{DSL p | typed msg (TVec TBool)
                            && vlength msg < pow 2 64}
            -> {res:DSL p | typed res (TVec TBool)
                            && (vlength res) mod 512 = 0} @-}
padding :: Num p => DSL p -> DSL p
padding msg = msg ++ (fromList TBool [BOOLEAN True])
                  ++ (vReplicate TBool k (BOOLEAN False))
                  ++ len
  where ...
```

# Optimizations

# Optimizations

```
((CONST 3 `SUB` CONST 1) `MUL` (VAR "x" TF)) `ADD` (VAR "y" TF)
```

$P$

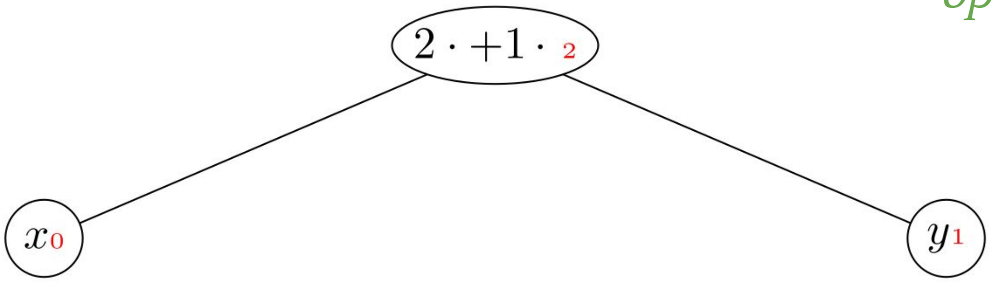



$$\begin{cases} 1 \cdot x_4 + 1 \cdot x_5 - 1 \cdot x_6 + 0 \cdot x_4 \cdot x_5 + 0 = 0 \\ 0 \cdot x_2 + 0 \cdot x_3 - 1 \cdot x_4 + 1 \cdot x_2 \cdot x_3 + 0 = 0 \\ 1 \cdot x_2 + 1 \cdot x_1 - 1 \cdot x_0 + 0 \cdot x_2 \cdot x_1 + 0 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_0 + 0 \cdot x_0 \cdot x_0 + 3 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_1 + 0 \cdot x_0 \cdot x_0 + 1 = 0 \end{cases}$$

# Optimizations

```
LINCOMB 2 (VAR “x” TF) 1 (VAR “y” TF)
```

*opt(P)*




$$\left\{ 2 \cdot x_0 + 1 \cdot x_1 - 1 \cdot x_2 + 0 \cdot x_0 \cdot x_1 + 0 = 0 \right.$$

# Optimizations are *proven* correct

- Optimizations happen at the DSL level.
  - They *change* what the user writes.
- **Theorem**: For each program $P \in PLINK$ and valuation σ of its variables

$$P \cong_{\sigma} opt(P)$$

where $opt : PLINK \rightarrow PLINK$ is the optimization function. Concretely, if $P$ has value $v$ under σ, then $opt(P)$ also has the same value.

  - Completely modular (e.g. in case we want to add new optimizations).

# Benchmark: SHA-256

# SHA-256 implementation

- Built using library functions (e.g. modular addition, bitwise xor, vector rotate…).
  - Some of these (e.g. bitwise xor, vector rotate…) can be implemented just as in Haskell.
  - ~220 lines of Haskell + Liquid Haskell annotations
- Standard functional implementation that combines these components.
  - ~210 lines of Haskell + Liquid Haskell annotations

| #Blocks | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| #Constraints | 79518 | 158208 | 237168 | 316132 | 395088 | 474057 | 553013 |
| Delta | — | 78690 | 78960 | 78964 | 78956 | 78969 | 78956 |

After optimizations, we managed ~79k constraints / block.

# Future work

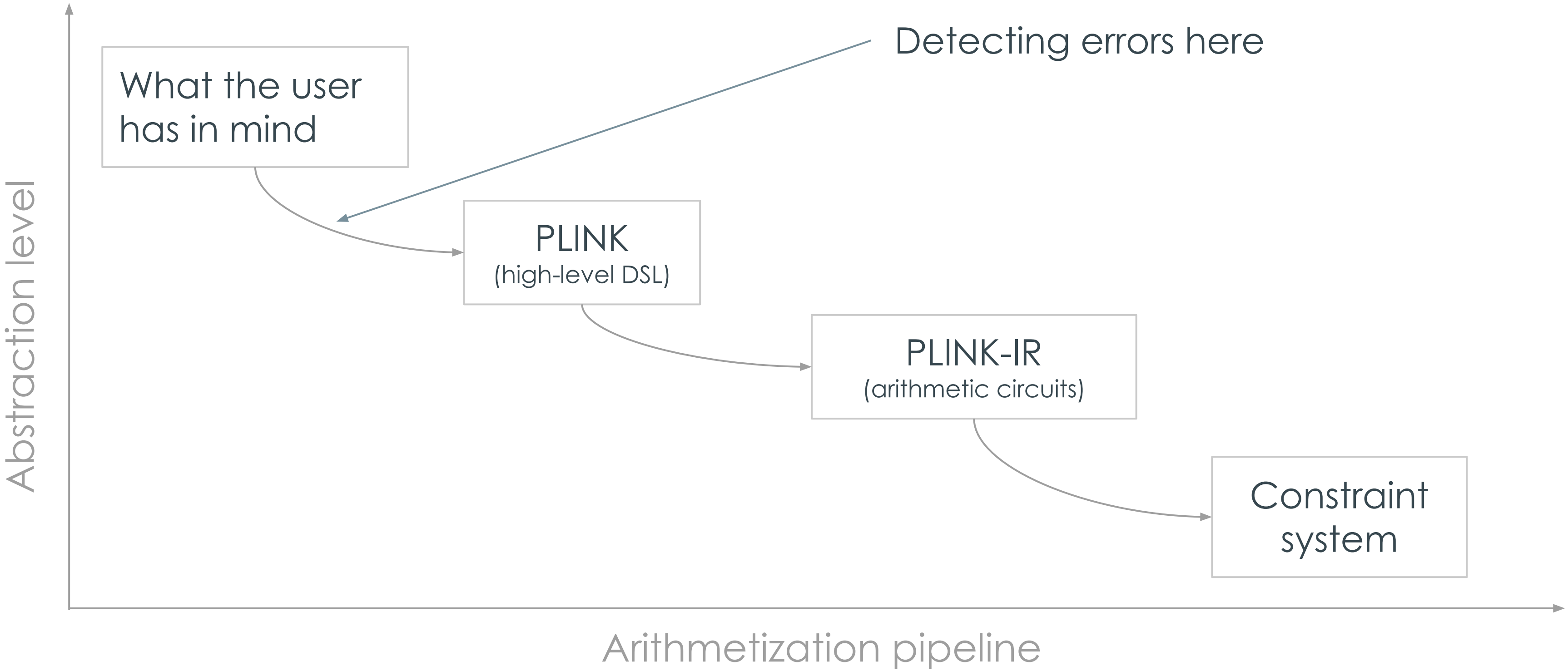
# Circuit verification

- Prove correctness of circuit implementations.
- Example: modular addition

```
assert $ BOOL b
assert $ (x `ADD` y) `EQA` (z `ADD` (b `MUL` CONST modulus))
toBinary e z -- z can be encoded using ‘e’ bits
```

  - Are these constraints really encoding “$z = x + y \pmod{2^e}$”?
- Liquid Haskell could be used to prove circuit correctness *on the part of the user.*

# Circuit verification



# Extend the language

with other PLONKish constructs. In particular,

- support for custom gates
- support for lookup tables

# PLINK

1. Embedded **DSL** in Haskell
2. Declarative, with support for **types** (bool, field, vectors)
3. **Semantic preservation** is proven using Liquid Haskell
4. **Optimizations** are also proven correct
5. We tested it by implementing SHA-256

Thank you! Questions?