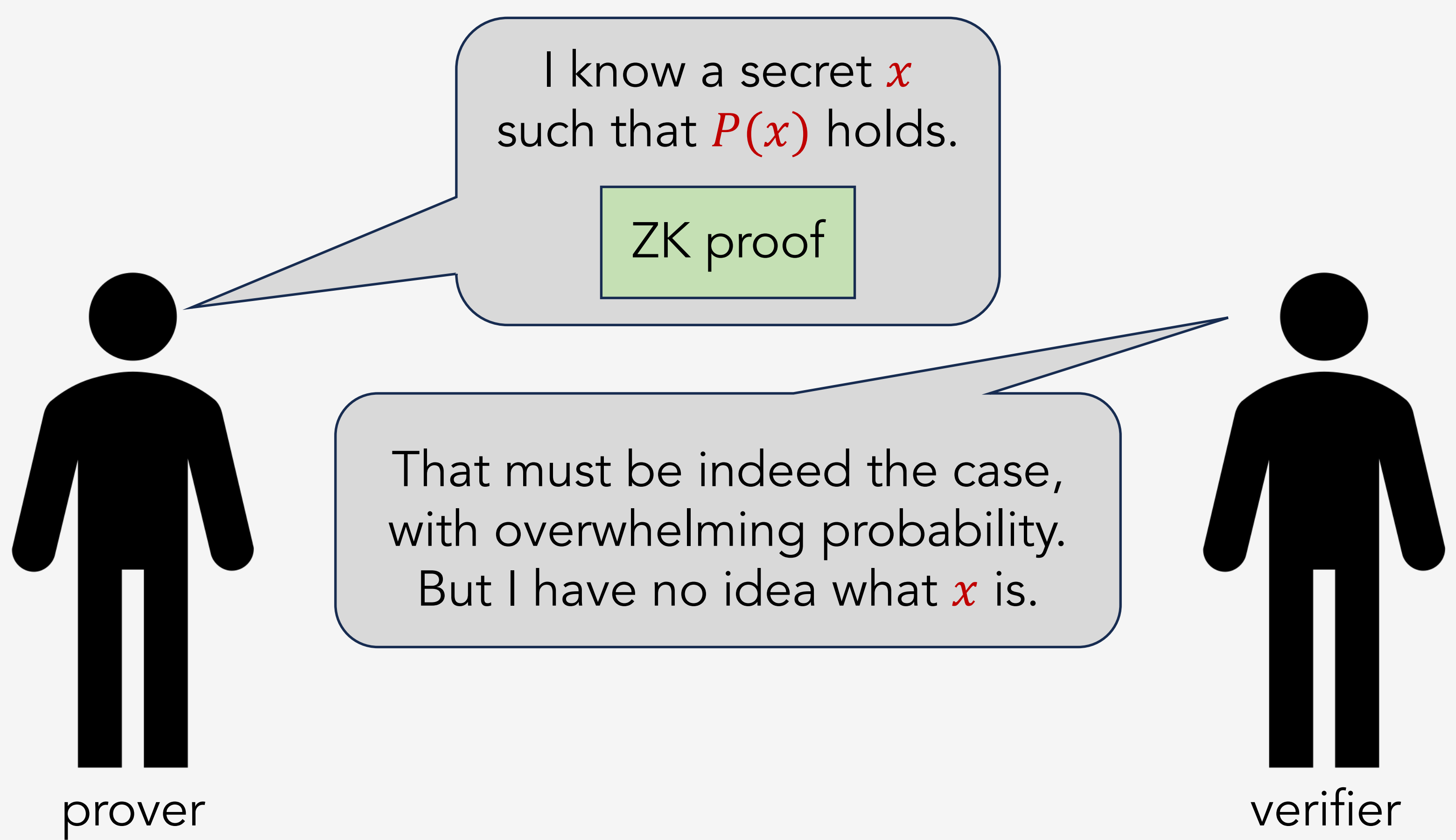
Formal Verification of Zero-Knowledge Circuits

Alessandro Coglio Eric McCarthy Eric Smith

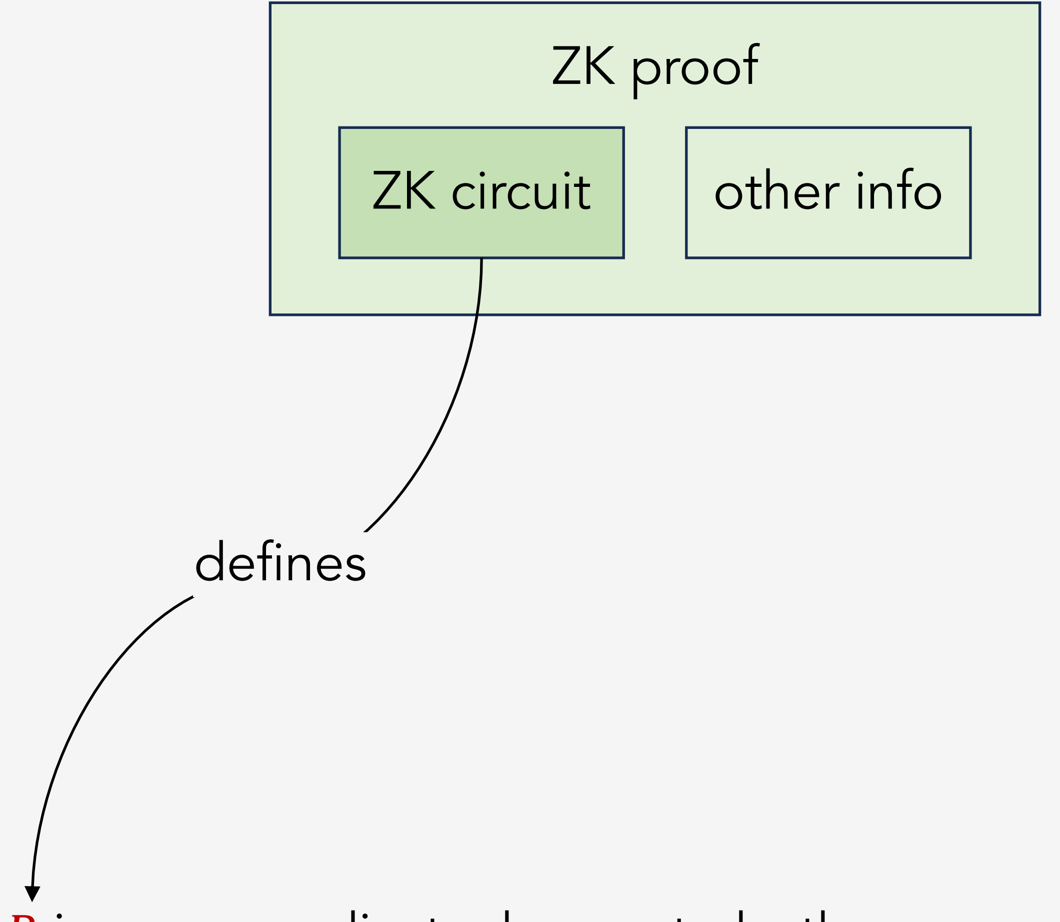
Aleo

Zero-knowledge circuits are used in zero-knowledge proofs.



P is some predicate, known to both prover and verifier.

Zero-knowledge circuits are used in zero-knowledge proofs.



P is some predicate, known to both prover and verifier.

The input parameter x of P is represented by one or more variables in the constraints.

The constraints are essentially a program to calculate $P(x)$ from x:
the solutions to the constraints yield the values of x that satisfy P.

equality constraints
over arithmetic expressions
interpreted in a prime field
(i.e. integers modulo a prime),
in R1CS or other form

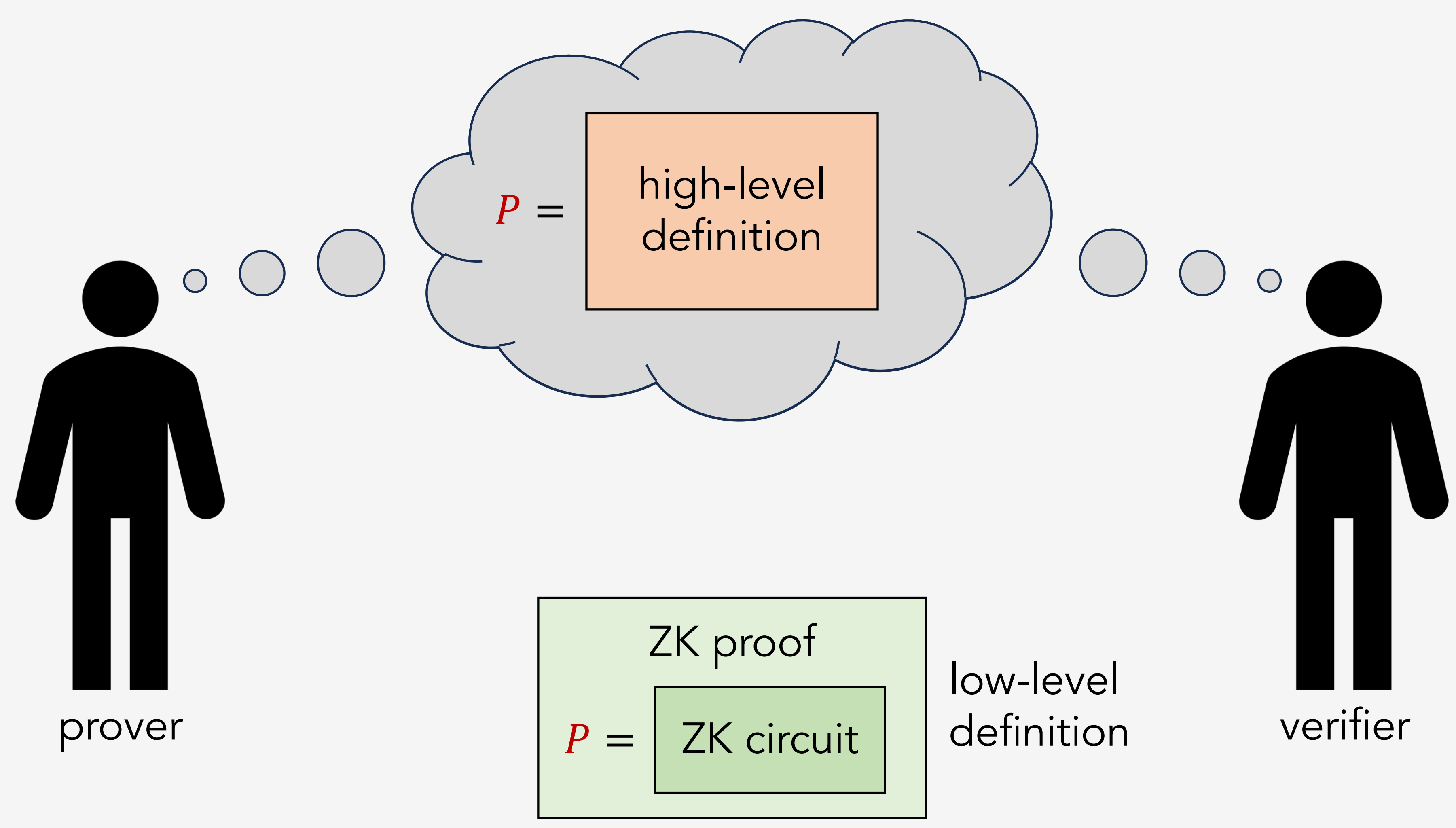
ZK circuit

$\ldots$
$(c)(1-c) = (0)$
$(2a)(b) = (a+b-y)$
$(c)(u-w) = (v-w)$
$(z)(1) = (x_0 + 2x_1 + \cdots + 2^{n-1}x_{n-1})$
$\ldots$

low-level
definition

defines

P is some predicate, known to both prover and verifier.



P is some predicate, known to both prover and verifier.

$P =$

high-level definition, e.g. in a programming language

```
...
// This function calculates the interest accrued
// over a variable number of iterations (max 50) for some `capital` and `rate`.
transition bounded_iteration_interest(capital: u32, public rate: u32, iterations: u8) -> u32 {
    assert(iterations <= 50u8);
    let amount: u32 = capital;

    // Accrue for up to 50 iterations.
    for i:u8 in 0u8..50u8 {
        if i < iterations {
            // Note that the added amount is rounded down.
            amount += (amount * rate) / 100u32;

        } // Skip the remaining iterations.
        if i == 40u8 {
            return amount;
        }
    }
    return amount;
}
...
```

How do we know that they define the same P?

If they do not, the ZK proof may not quite prove what is expected.

We use formal verification, of course.

P = high-level definition

$\stackrel{?}{=}$

P = ZK circuit (low-level)

This is how we use formal verification to ensure that a ZK circuit is correct.

specification

The high-level definition is a specification.

=

The ZK circuit can be for P, or for some part of it.

ZK circuit (low-level)

This is how we use formal verification to ensure that a ZK circuit is correct.

specification

=

ZK circuit

② Formal characterization of the specification.

③ Formal proof ($\neq$ ZK proof) of equivalence.

① Formal characterization of the ZK circuit.

specification example

$$z = \begin{cases} x/y & \text{if } y \neq 0 \\ \mathcal{E} & \text{if } y = 0 \end{cases}$$

Divide x by y if $y \neq 0$, otherwise return $\mathcal{E}$ (error).

=

ZK circuit example

$$(y)(w) = (1)$$
$$(x)(w) = (z)$$

The 1st constraint sets $w = 1/y$, if $y \neq 0$.

The 2nd constraint sets $z = x/y$, if $y \neq 0$.

If $y = 0$, the constraints have no solution.

specification example

$$f(x,y) = \begin{cases} x/y & \text{if } y \neq 0 \\ \mathcal{E} & \text{if } y = 0 \end{cases}$$

$$S(x,y,z) = [z = f(x,y) \neq \mathcal{E}\,]$$

Formally, a specification is a functional computation f from inputs to outputs or error.

This determines a relation S over the input/output variables.

based on

$$Q(x,y,z) = S(x,y,z)$$

Formally, the ZK circuit correctness is expressed as $Q(x,y,z) \Leftrightarrow S(x,y,z)$.

input variables

output variables

auxiliary variables

ZK circuit example

$$R(x,y,z,w) = \begin{Bmatrix} (y)(w) = (1) \\ (x)(w) = (z) \end{Bmatrix}$$

$$Q(x,y,z) = \exists w.\, R(x,y,z,w)$$

Formally, a ZK circuit is a relation R over the variables.

This determines a relation Q over the input/output variables, by existentially quantifying over the auxiliary variables.

specification

$$f(\vec{\imath}) = \begin{cases} \dots & \text{if } \dots \\ \mathcal{E} & \text{if } \dots \end{cases}$$

$$S(\vec{\imath}, \vec{\omega}) = [\vec{\omega} = f(\vec{\imath}) \neq \mathcal{E}\,]$$

Formally, a specification is a functional computation f from inputs to outputs or error.

This determines a relation S over the input/output variables.

based on

$$Q(\vec{\imath}, \vec{\omega}) \iff S(\vec{\imath}, \vec{\omega})$$

Formally, the ZK circuit correctness is expressed as $Q(\vec{\imath}, \vec{\omega}) \iff S(\vec{\imath}, \vec{\omega})$.

input variables

output variables

auxiliary variables

ZK circuit

$$R(\vec{\imath}, \vec{\omega}, \vec{\alpha}) = \{\text{constraints}\}$$

$$Q(\vec{\imath}, \vec{\omega}) = \exists \vec{\alpha}.\, R(\vec{\imath}, \vec{\omega}, \vec{\alpha})$$

Formally, a ZK circuit is a relation R over the variables.

This determines a relation Q over the input/output variables, by existentially quantifying over the auxiliary variables.

specification

$$f(\vec{\imath}) = \begin{cases} \dots & \text{if } \dots \\ \mathcal{E} & \text{if } \dots \end{cases}$$

$$S(\vec{\imath},\vec{\omega}) = [\vec{\omega} = f(\vec{\imath}) \neq \mathcal{E}\,]$$

$Q(\vec{\imath},\vec{\omega}) \iff S(\vec{\imath},\vec{\omega})$

correctness

$Q(\vec{\imath},\vec{\omega}) \implies S(\vec{\imath},\vec{\omega})$

soundness

(≠ their meaning in ZK proofs)

$Q(\vec{\imath},\vec{\omega}) \impliedby S(\vec{\imath},\vec{\omega})$

completeness

ZK circuit

$$R(\vec{\imath},\vec{\omega},\vec{\alpha}) = \{\text{constraints}\}$$

$$Q(\vec{\imath},\vec{\omega}) = \exists\vec{\alpha}.\, R(\vec{\imath},\vec{\omega},\vec{\alpha})$$

To prove soundness, we expand Q and turn ∃ over the antecedent into ∀ over the implication.

$$\forall\vec{\imath},\vec{\omega},\vec{\alpha}.\ R(\vec{\imath},\vec{\omega},\vec{\alpha}) \implies S(\vec{\imath},\vec{\omega})$$

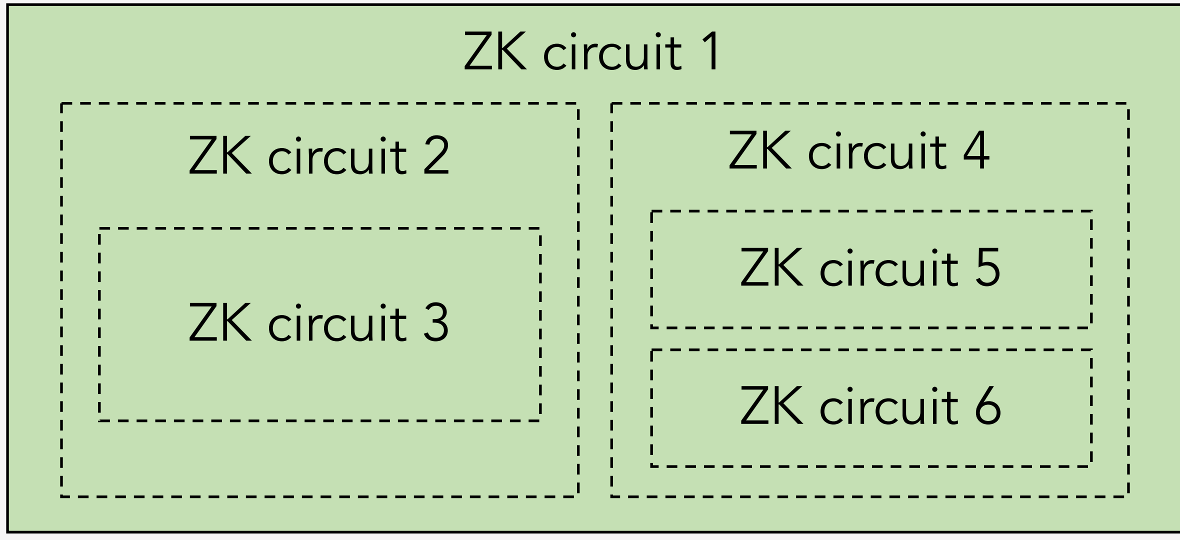
Every solution to the constraints is a non-erroneous computation.

Every non-erroneous computation is a solution to the constraints.

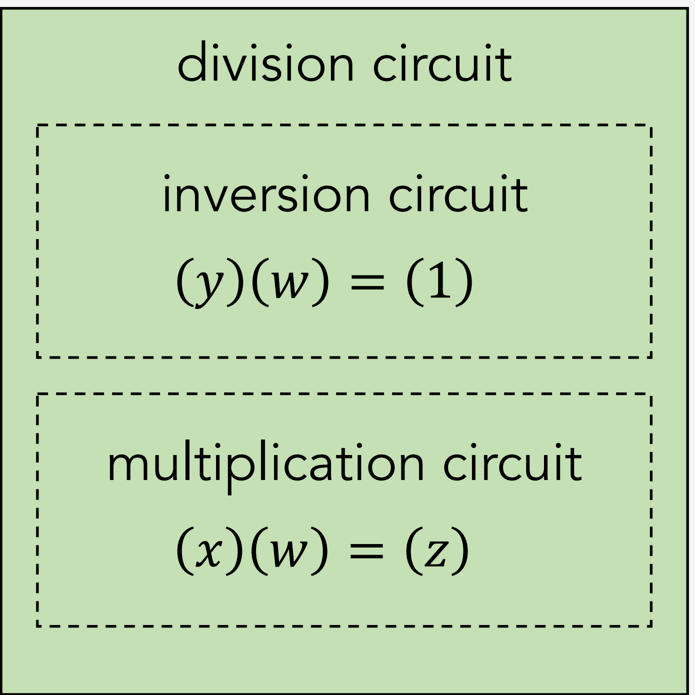
To prove completeness, we expand Q but we cannot turn ∃ into ∀ here, and we must exhibit $\vec{\alpha}$ witnesses.

$$\forall\vec{\imath},\vec{\omega}.\ S(\vec{\imath},\vec{\omega}) \implies \exists\vec{\alpha}.\, R(\vec{\imath},\vec{\omega},\vec{\alpha})$$

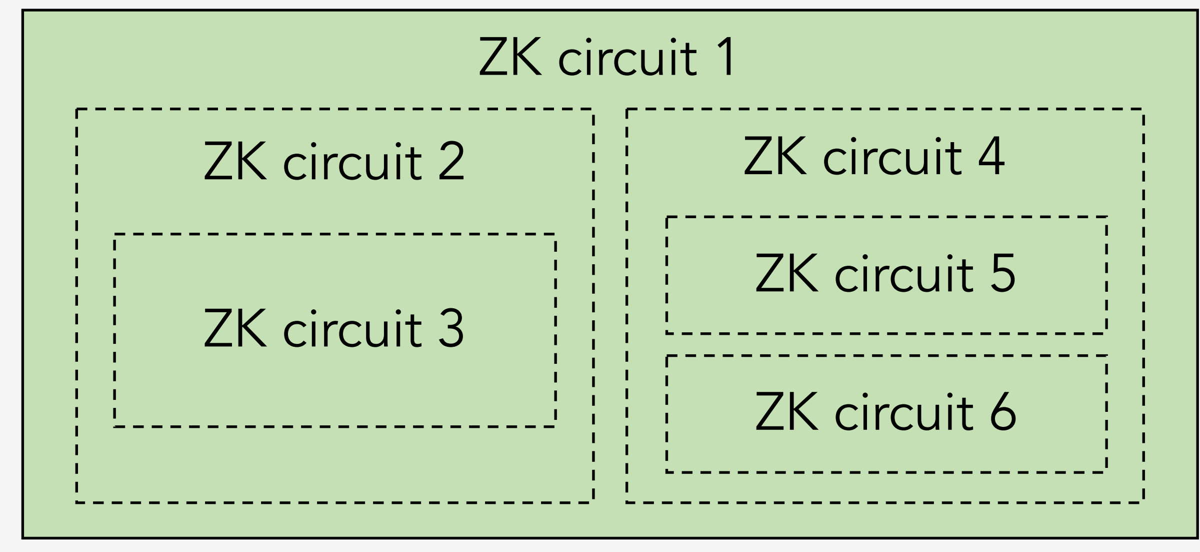
ZK circuits have an implicit hierarchical structure.



ZK circuits have an implicit hierarchical structure.

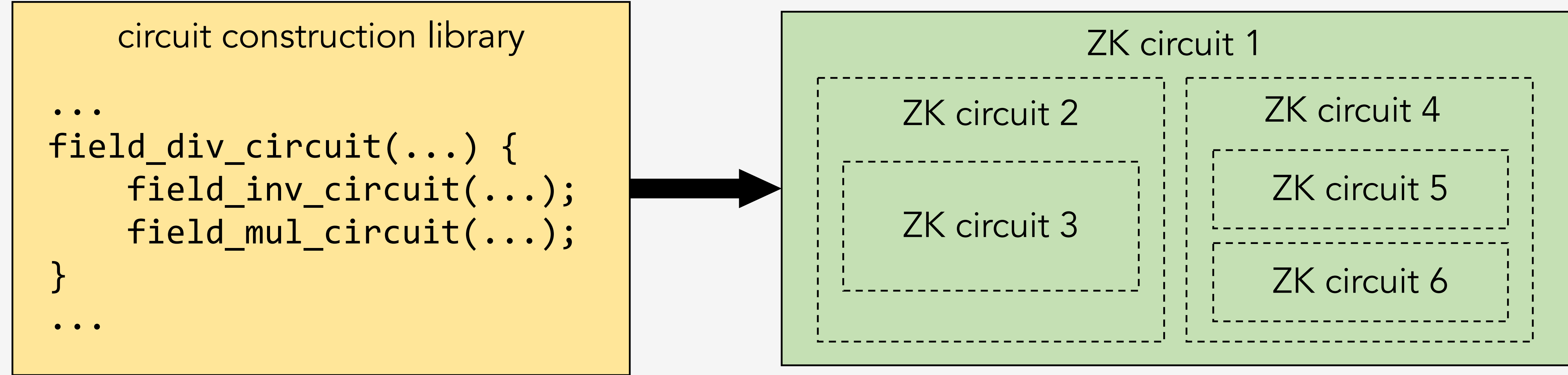


(very simple example)



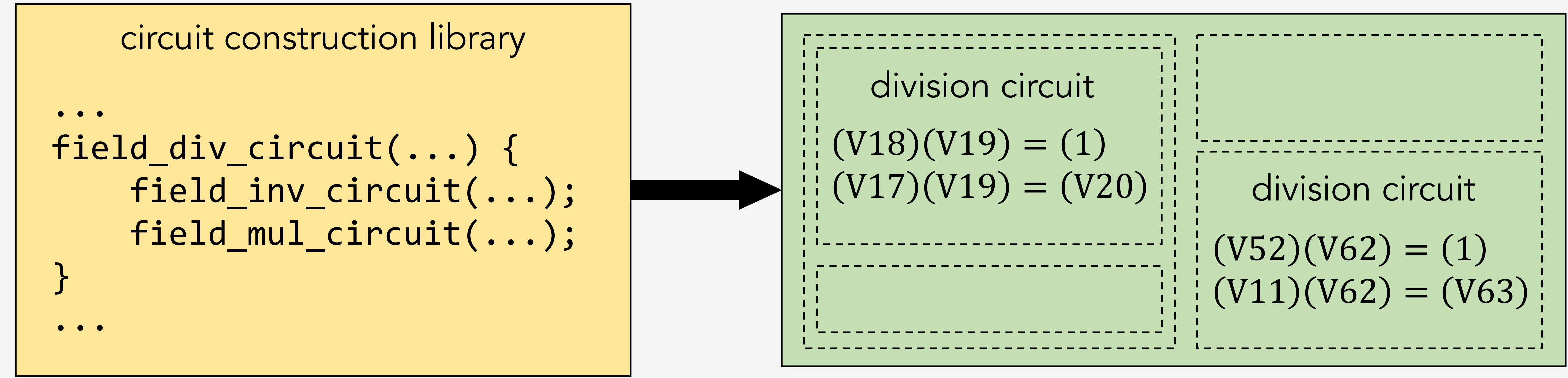
This implicit hierarchical structure derives from the way the circuits are constructed.

Circuits are hierarchically constructed via libraries like snarkVM, bellman, etc.



This implicit hierarchical structure derives from the way the circuits are constructed.

Circuits are hierarchically constructed via libraries like snarkVM, bellman, etc.



This implicit hierarchical structure derives from the way the circuits are constructed.

Circuits are hierarchically constructed via libraries like snarkVM, bellman, etc.

But the generated constraints are a flat list; the hierarchy is not explicit.

Every circuit instance has different variables, with increasing indices.

Reconstructing the hierarchy from the flat list is difficult in general.

circuit construction library

```
...
field_div_circuit(...) {
    field_inv_circuit(...);
    field_mul_circuit(...);
}
...
```

…	…
…	…
$(V18)(V19) = (1)$	…
$(V17)(V19) = (V20)$	…
…	$(V52)(V62) = (1)$
…	$(V11)(V62) = (V63)$
…	…

Our initial proofs did not take advantage of the hierarchical structure, naturally.

circuit construction library

```
...
field_div_circuit(...) {
    field_inv_circuit(...);
    field_mul_circuit(...);
}
...
```

...
...
(V18)(V19) = (1)
(V17)(V19) = (V20)
...
...
...

...
...
...
...
(V52)(V62) = (1)
(V11)(V62) = (V63)
...

extract convert

representation of the circuit in ACL2

```
(defconst *circuit* <constraints>)
```

proofs in ACL2 and Axe

```
(defthm circuit-correct
  ... (satp *circuit* ...) ...)
```

model of R1CS in ACL2

```
...
(defaggregate constraint
  ((a (sparse-vectorp a))
   (b (sparse-vectorp b))
   (c (sparse-vectorp c))))
...
(defund satp (constrs asg prime) ...)
...
```

Our initial proofs did not take advantage of the hierarchical structure, naturally.

circuit construction library

```
...
field_div_circuit(...) {
    field_inv_circuit(...);
    field_mul_circuit(...);
}
...
```

$$
\begin{array}{ll}
\ldots & \ldots \\
\ldots & \ldots \\
(V18)(V19) = (1) & \ldots \\
(V17)(V19) = (V20) & \ldots \\
\ldots & (V52)(V62) = (1) \\
\ldots & (V11)(V62) = (V63) \\
\ldots & \ldots
\end{array}
$$

extract convert

representation of the circuit in ACL2

```
(defconst *circuit* <constraints>)
```

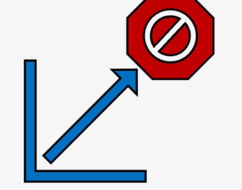
proofs in ACL2 and Axe

```
(defthm circuit-correct
  ... (satp *circuit* ...) ...)
```

We verified several circuits, for Ethereum, Zcash, & Aleo.

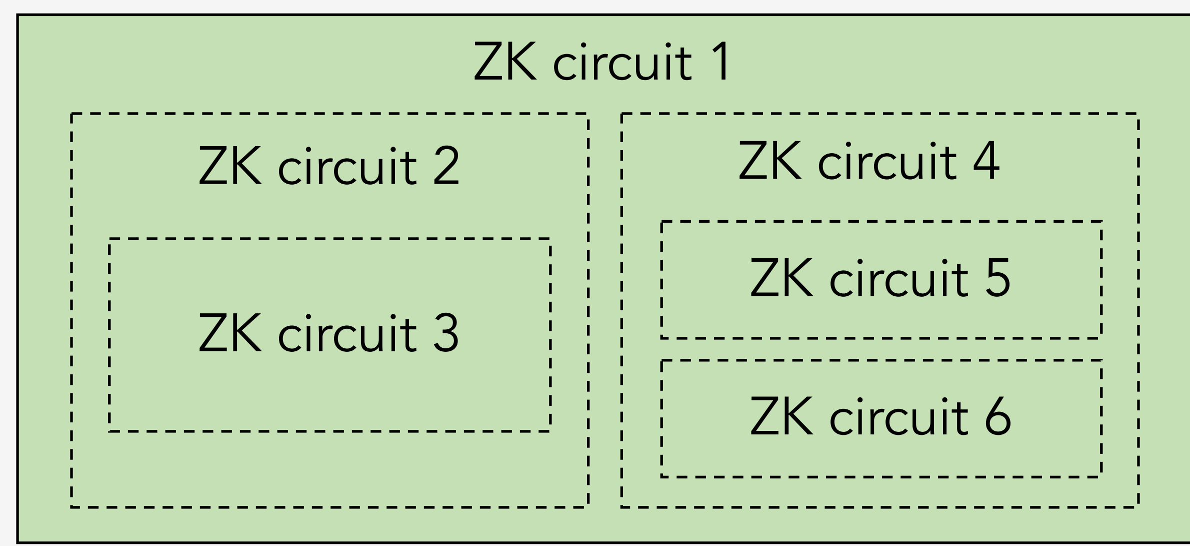
We ran into scalability problems:

- Size of circuits grows quickly.
- Hard to reuse sub-circuit proofs.



Axe helps with the first problem, but the second problem remains.

The next step was to take advantage of the hierarchical structure.



We replicate the hierarchical circuit constructions in ACL2.

circuit construction library

```
...
field_div_circuit(...) {
    field_inv_circuit(...);
    field_mul_circuit(...);
}
...
```

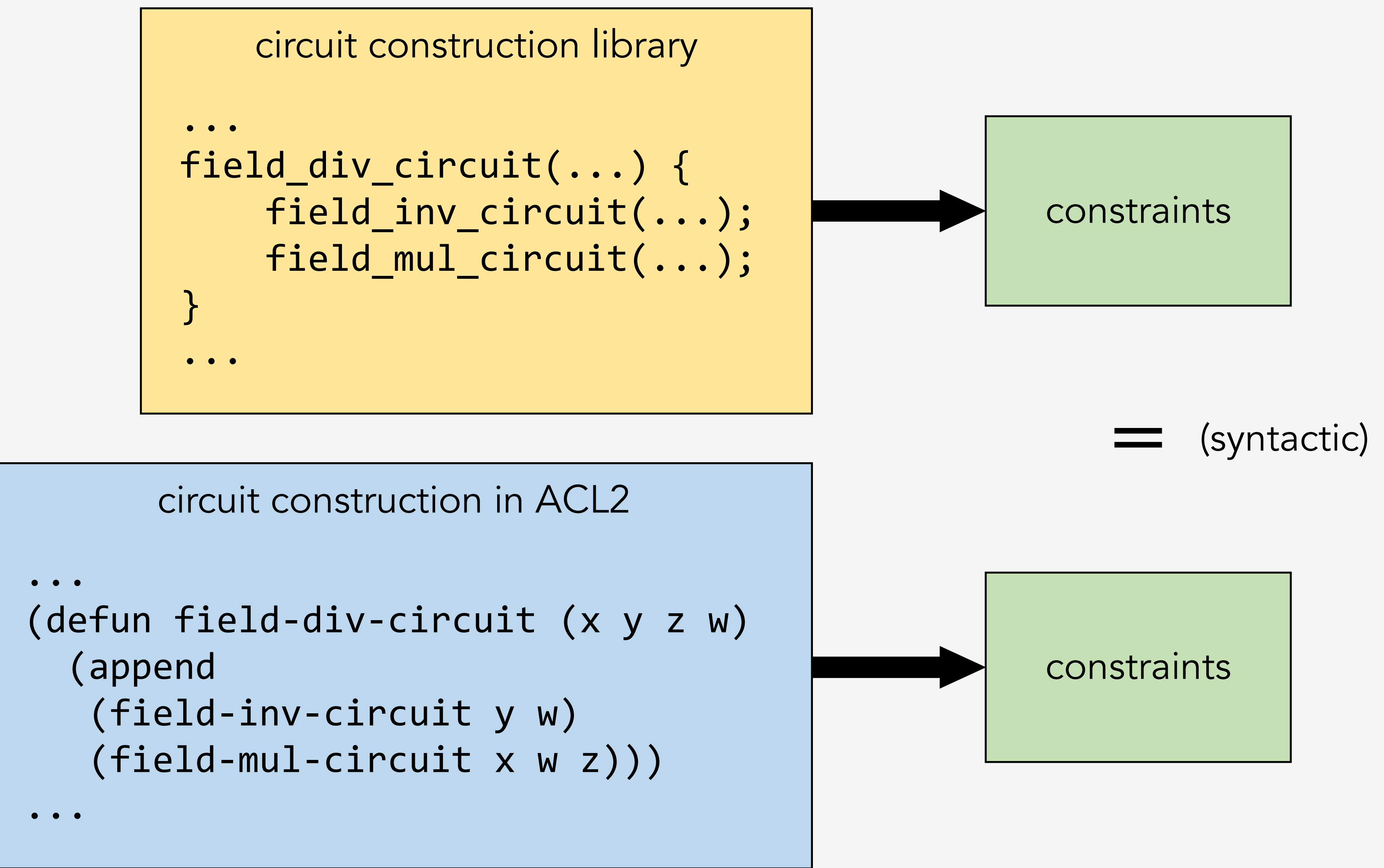
constraints

circuit construction in ACL2

```
...
(defun field-div-circuit (x y z w)
  (append
   (field-inv-circuit y w)
   (field-mul-circuit x w z)))
...
```

Circuits are constructed by ACL2 functions,
parameterized over the variables to use,
which call other functions to construct sub-circuits,
and/or construct constraints directly.

We validate the circuit constructions by comparing the generated constraints syntactically.



division circuit

$$(y)(w) = (1)$$
$$(x)(w) = (z)$$

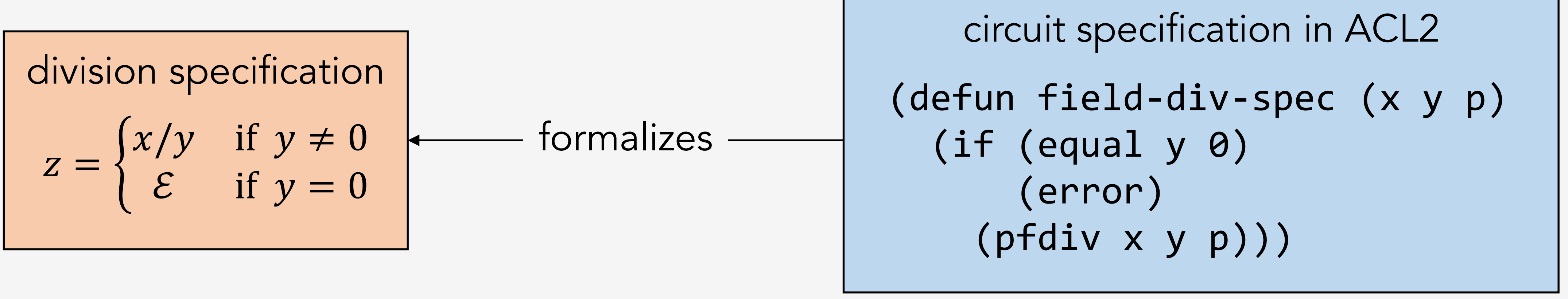
← formalizes —

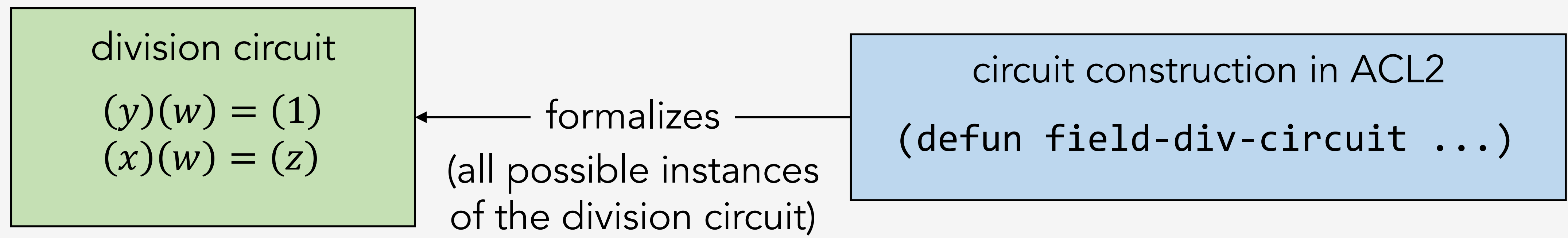
(all possible instances of the division circuit)

circuit construction in ACL2

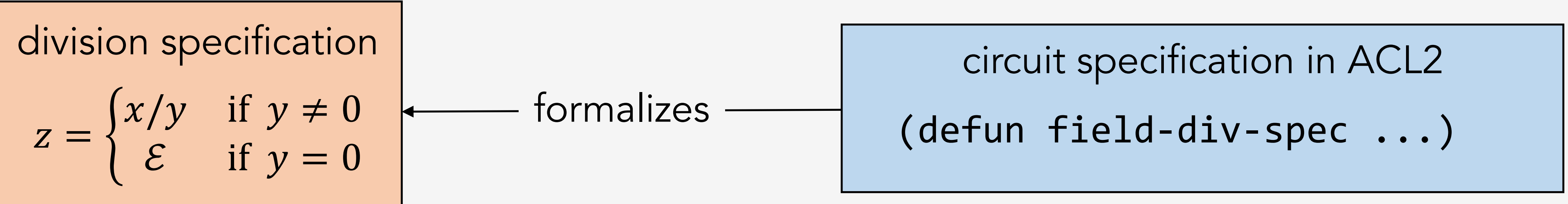
```
(defun field-div-circuit ...)
```

We write the circuit specifications in ACL2 as well.





We prove correctness in ACL2, for all possible circuit instances.



$\Updownarrow$ formally proves

circuit correctness theorem in ACL2

```
(defthm field-div-circuit-correctness
  (implies ... ; <- boilerplate hypotheses
    (equal (satp (field-div-circuit x y z w) asg p)
           (and (equal zval (field-div-spec xval yval p))
                ...)))) ; <- discussed later
```

division circuit

$$(y)(w) = (1)$$
$$(x)(w) = (z)$$

formalizes

circuit construction in ACL2

```
(defun field-div-circuit ...)
```

The formal proofs are compositional, according to the circuit hierarchy.

```
(defthm some-larger-circuit-correctness ...)
```

makes use of

```
(defthm field-div-circuit-correctness ...)
```

…

makes use of

```
(defthm field-inv-circuit-correctness ...)
```

```
(defthm field-mul-circuit-correctness ...)
```

The parameterization of the circuits goes beyond variable names: some circuits have varying numbers of variables and constraints.

Their correctness is proved using induction (directly or indirectly).

Their correctness holds for all possible sizes, beyond all possible variable names.

field-to-bits circuit

$$(y_0)(1 - y_0) = (0)$$
$$(y_1)(1 - y_1) = (0)$$
$$\dots$$
$$(y_{n-1})(1 - y_{n-1}) = (0)$$

$$(x)(1) = (y_0 + 2y_1 + \dots + 2^{n-1}y_{n-1})$$

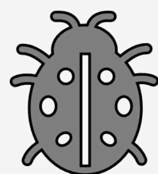
$$\left\langle \sum_{i=0}^{n-1} 2^i\, y_i < p \right\rangle$$ (details omitted)

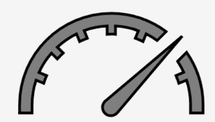
circuit construction in ACL2

list of variables

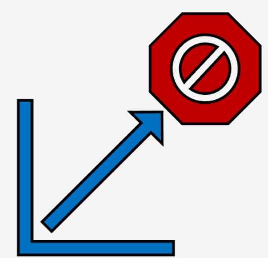
```
(defun field-to-bits-circuit (x ys ...)
  (append (boolean-check-circuit-list ys)
          (pow2sum-circuit x ys)
          (bits-lt-prime-circuit ys ...)))
```

We have used this approach to formalize and verify a large subset of Aleo's snarkVM circuits for boolean, field, and integer operations.
We are working on the remaining snarkVM circuits.





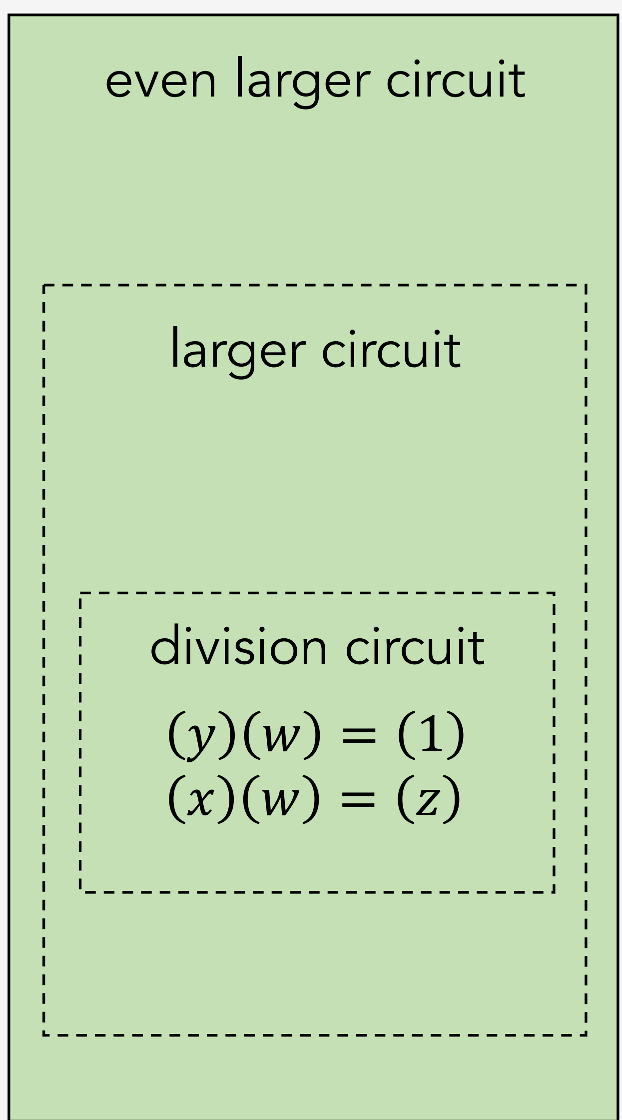
Besides increasing confidence in snarkVM's circuit constructions, this work led us to discover a few bugs and several optimizations.



We ran into another scalability problem, related to the auxiliary variables of circuits.

The auxiliary variables of a circuit are exposed as parameters of the ACL2 functions.

The parameters of the ACL2 functions grow as larger and larger circuits are built.




```
(defun even-larger-circuit (... w ...) ...)
```

pass

```
(defun larger-circuit (... w ...) ...)
```

pass

```
(defun field-div-circuit (x y z w) ...)
```

input
output
auxiliary

The auxiliary variables of a circuit are also exposed in the ACL2 correctness theorems.

The theorem must talk about the auxiliary variables, but they are not part of the specification.

This propagates up to the correctness theorems of all the containing circuits.

division specification

$$z = \begin{cases} x/y & \text{if } y \neq 0 \\ \mathcal{E} & \text{if } y = 0 \end{cases}$$

```
(defun field-div-spec (x y p) ...)
```

$\Updownarrow$

```
(defthm field-div-circuit-correctness
  (implies ... ; <- boilerplate hypotheses
     (equal (satp (field-div-circuit x y z w) asg p)
            (and (equal zval (field-div-spec xval yval p))
                 (equal wval (pfinv yval p))))))
```

division circuit

$$(y)(w) = (1)$$
$$(x)(w) = (z)$$

```
(defun field-div-circuit (x y z w) ...)
```

The auxiliary variables of a circuit are not exposed in our new PFCS formalism.

PFCS (= Prime Field Constraint Systems) generalize R1CS (= Rank-1 Constraint Systems):

(1) equalities can be over any prime field expressions;

(2) constraints can be grouped into named predicates.

captures R1CS and other forms

captures hierarchy

The auxiliary variables of a circuit are not exposed in our new PFCS formalism.

R1CS

division circuit

inversion circuit

$(y)(w) = (1)$

multiplication circuit

$(x)(w) = (z)$

PFCS

inversion circuit

$inv(a, b) = \{(a)(b) = (1)\}$

multiplication circuit

$mul(u, v, w) = \{(u)(v) = (w)\}$

division circuit

$$div(x, y, z) = \left\{ \begin{array}{c} inv(y, w) \\ mul(x, w, z) \end{array} \right\}$$

Predicate with name inv, parameters a, b, one equality constraint, no auxiliary variables.

Predicate with name mul, parameters u, v, w, one equality constraint, no auxiliary variables.

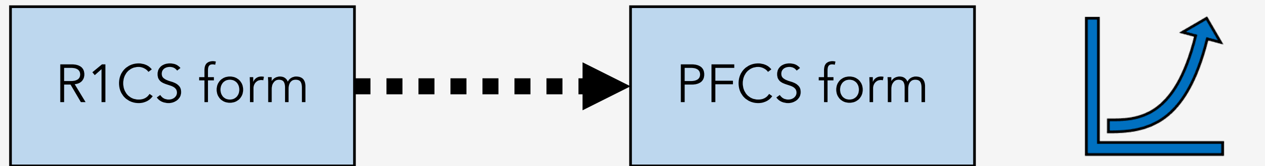
Predicate with name div, parameters x, y, z, two predicate constraints, auxiliary variable w.

captures $\exists w$ → $Q(x, y, z)$ is the semantics of the predicate itself

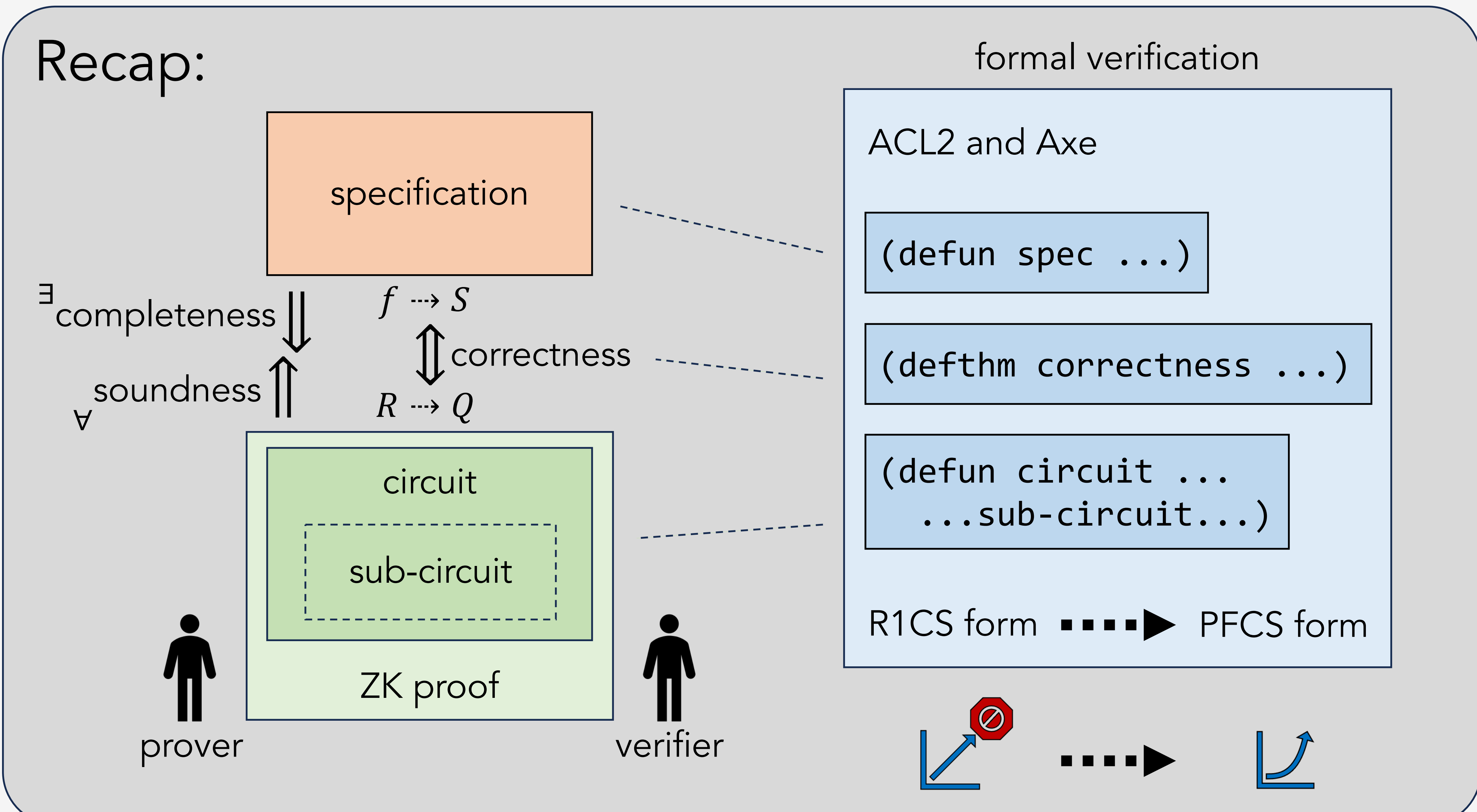
$R(x, y, z, w)$ is the semantics of the predicate body

We have formalized the PFCS syntax and semantics in ACL2.

We are porting our formalized and verified snarkVM circuits to PFCS form.



This solves the scalability problem with the auxiliary variables of circuits.



Recap:
specification
∃ completeness
∀ soundness
$f \dashrightarrow S$
correctness
$R \dashrightarrow Q$
circuit
sub-circuit
ZK proof
prover
verifier
formal verification
ACL2 and Axe
(defun spec ...)
(defthm correctness ...)
(defun circuit ...
...sub-circuit...)
R1CS form
PFCS form