A Verified Algebraic Representation of Cairo Program Execution

Jeremy Avigad[1] Lior Goldberg[2] David Levit[2] Yoav Seginer
Alon Titelman[2]

January 17, 2022

[1]Carnegie Mellon University

[2]StarkWare Industries Ltd.

Table of contents

Outline:

STARKs and formal verification

STARK stands for Scalable, Transparent, Argument of Knowledge. It is a cryptographic proof protocol.

StarkWare develops STARK-based solutions for blockchain. It offers methods of carrying out exchanges, auctions, and other operations on blockchain efficiently.

We have been using the Lean 3 proof assistant to verify their methods and code.

Blockchain

The magic:

- A ledger in the sky.
- The ledger grows, but nothing is ever deleted.
- Users can own resources, like bitcoin, and transfer them.

Application: banking without government or institutional involvement.

Under the hood:

- Decentralized maintenance.
- Proof of work or proof of stake.
- Incentives to maintain and extend the blockchain.
- Public key cryptography.

Smart contracts

The magic:

- One can put a computer program on the ledger that executes when some conditions are met.

Application: contracts, auctions, exchanges, voting, etc., again without government or institutional control.

Under the hood:

- A programming language.
- Blockchain.

Smart contracts

Limitations:

- It is expensive (everyone maintaining the blockchain has to execute the code).
- It is slow. (For example, the number of transactions that Visa performs in a second dwarf what can be done on blockchain.)

Cryptographic proof protocols

Let $R(x, y)$ be some relation, and let a be some public data.

My goal: convince you that I know a b such that $R(a, b)$.

One solution: I show you b, and you check $R(a, b)$. If b is long and $R(a, b)$ is hard to check, this is inefficient.

Cryptographic proof protocols are designed to do this probabilistically but more efficiently.

Computing efficiently on blockchain

First we specify a model of computation, a CPU and execution semantics. Keep that fixed.

Memory is read only! Programs check claims about what is in memory.

We make the following data public:

- A partial assignment to the memory (typically the program, the input, the output).
- The starting state of the processor.
- The final state.

Computing efficiently on blockchain

I, the prover, want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

I don't have to show you what else goes in the memory. Maybe you don't even care.

STARKs can do this efficiently.

Table of contents

Outline:

Markets for verification

An interesting formalization project: verify the cryptographic claims about STARKs.

Here, we'll take them as a black box.

Two current jobs for Lean:

- Verify the reduction from execution semantics to STARKs.
- Verify particular programs in the Cairo language.

Here we report on the first of these.

The formalization is available online at `https://github.com/starkware-libs/formal-proofs`.

The algebraic intermediate representation (AIR)

Fix a large prime number p, and let F_p be the finite field of integers modulo p.

A STARK lets me convince you that I have a table of field elements, satisfying some constraints.

For polynomials $P_1(\vec{x}_1), \ldots, P_s(\vec{x}_s)$ and periodic sets $D_1, \ldots, D_s$, the constraints say that for every $j = 1, \ldots, s$ and every $r \in D_j$, we have $P_k(\vec{x}_k^{(r)}) = 0$, where $\vec{x}_k^{(r)}$ is $\vec{x}_k$ shifted by r rows.

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

- Encode statements about execution traces in terms of polynomials. (The AIR.)

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

- Encode statements about execution traces in terms of polynomials. (The AIR.)
- Prove to you that the encoding is correct. (The whitepaper.)

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

- Encode statements about execution traces in terms of polynomials. (The AIR.)
- Prove to you that the encoding is correct. (The whitepaper.)
- Verify the proof. (Lean.)

CPU semantics

The CPU state consists of three registers: a program counter, an allocation pointer, and a frame pointer.

Instructions:

- Assert two things are equal (e.g. $[x] = [y] + 3$).
- Jump.
- Conditional jump.
- Call.
- Return.

Memory consists of values in a large finite field. Arithmetic operations are $+$ and $*$.

CPU semantics

We have:

- a whitepaper specification of the CPU
- a formal specification in Lean

Formally, we define the next state relation:

```
def next_state (mem : F → F) (s t : register_state F) :
  Prop := ...
```

There may be no next state (when assertions fail) or multiple next states (undefined behavior).

CPU semantics

An instruction consists of 15 one-bit flags and three 16-bit bitvectors.

Flags determine the relevant instruction, memory addressing modes, and so on.

```
structure register_state (F : Type*) :=
(pc : F) (ap : F) (fp : F)

def next_fp : option F :=
match i.opcode_call, i.opcode_ret, i.opcode_assert_eq with
| ff, ff, ff := some s.fp
| tt, ff, ff := some (s.ap + 2)
| ff, tt, ff := some (i.dst mem s)
| ff, ff, tt := some s.fp
| _,  _,  _  := none
end
```

The polynomial constraints

The polynomial constraints need to guarantee the existence of a valid trace consistent with the public data.

- Unpacking and interpreting instructions.
- Verifying that certain field elements are integers, and in range.
- Verifying that the memory assignment is valid, and extends the public one.
- Verifying each step.

These are found in three places:

- Locally, in each file.
- Globally, collected together.
- In a slightly different format, autogenerated from StarkWare code.

The polynomial constraints

The raw polynomial constraints are generated from the same code used by the StarkWare compiler to generate STARK certificates.

Clients can verify from the published certificates that these are the polynomials that were used.

The Lean proof verifies that the existence a table of values satisfying these polynomials implies the existence of the relevant execution trace.

The polynomial constraints

```
structure cpu__update_registers (inp : input_data F) (c : columns F) :
    Prop :=
(update_ap__ap_update : ∀ i: nat, (((i % 16 = 0)) ∧ ¬ (i = 16 *
    (inp.trace_length / 16 - 1))) → (c.column21.off i 16) -
    ((c.column21.off i 0) + ((c.column1.off i 10) - ((c.column1.off i
    11) + (c.column1.off i 11))) * (c.column21.off i 12) +
    (c.column1.off i 11) - ((c.column1.off i 12) + (c.column1.off i
    12)) + ((c.column1.off i 12) - ((c.column1.off i 13) +
    (c.column1.off i 13))) * 2) = 0)
(update_fp__fp_update : ∀ i: nat, (((i % 16 = 0)) ∧ ¬ (i = 16 *
    (inp.trace_length / 16 - 1))) → (c.column21.off i 24) - ((1 -
    ((c.column1.off i 12) - ((c.column1.off i 13) + (c.column1.off i
    13)) + (c.column1.off i 13) - ((c.column1.off i 14) +
    (c.column1.off i 14)))) * (c.column21.off i 8) + ((c.column1.off i
    13) - ((c.column1.off i 14) + (c.column1.off i 14))) *
    (c.column19.off i 9) + ((c.column1.off i 12) - ((c.column1.off i
    13) + (c.column1.off i 13))) * ((c.column21.off i 0) + 2)) = 0)
(update_pc__pc_cond_negative : ∀ i: nat, (((i % 16 = 0)) ∧ ¬ (i = 16 *
    (inp.trace_length / 16 - 1))) → (1 - ((c.column1.off i 9) -
    ((c.column1.off i 10) + (c.column1.off i 10)))) * (c.column19.off
    i 16) + (c.column21.off i 2) * ((c.column19.off i 16) -
    ((c.column19.off i 0) + (c.column19.off i 13))) - ((1 -
```

Table of contents

Outline:

The main theorem: first cut

Given

- a partial assignment to memory,
- a start state,
- a final state,
- a table of data satisfying the polynomial constraints with respect to these,

there exists

- a total assignment to memory extending the partial one and
- a valid execution trace from the start state to the final state.

A challenge

Suppose at a stage in the trace, the code asserts that memory position a_i contains value v_i.

To establish the the execution succeeds, the prover needs to establish two things:

- All the pairs (a_i, v_i) form a consistent memory assignment.
- The assignment extends the public partial assignment that the verifier agreed to.

This can't be done efficiently with polynomial constraints, which are *local* constraints.

A similar problem: verifying that a field element is a cast of an integer in a certain range.

A solution

The protocol uses yet another cryptographic solution.

Within the STARK, the prover commits to 23 columns of data.

A cryptographic hash is generated, roughly, a random challenge.

The prover responds with two more columns of data.

We prove that with high probability, the fact that the data satisfies the constraints implies the desired conclusion.

The main theorem

Given

- a partial assignment to memory,
- a start state,
- a final state,
- a table of data,

there exist small sets B_1, B_2, B_3, such that given

- field elements x_1, x_2, x_3,
- two more columns of data,

if the xs are not in the Bs and the data satisfies the polynomial constraints, there exists

- a total assignment to memory extending the partial one and
- a valid execution trace from the start state to the final state.

The main theorem

```
theorem final_correctness
  {F : Type} [field F] [fintype F]
  (char_ge : ring_char F ≥ 2^63)
  /- public data -/
  (inp : input_data F)
  (pd  : public_data F)
  (pc  : public_constraints inp pd)
  (c   : columns F) :
  /- sets to avoid -/
∃ bad1 bad2 bad3 : finset F,
  bad1.card ≤ (inp.trace_length / 2)^2 ∧
  bad2.card ≤ inp.trace_length / 2 ∧
  bad3.card ≤ inp.trace_length ∧
```

The main theorem

```
∀ ci : columns_inter F,
  /- autogenerated constraints-/
  cpu__decode c ∧
  cpu__operands c ∧
  cpu__update_registers c inp ∧
  cpu__opcodes c ∧
  memory inp pd c ci ∧
  rc16 inp pd c ci ∧
  public_memory c ∧
  initial_and_final inp c ∧
  /- probabilistic constraints -/
  pd.hash_interaction_elm0 ∉ bad1 ∧
  pd.interaction_elm ∉ bad2 ∧
  pd.interaction_elm ≠ 0 ∧
  pd.rc16__perm__interaction_elm ∉ bad3 →
```

The main theorem

```
/- the conclusion -/
let T := inp.trace_length / 16 - 1 in
∃ mem : F → F,
  option.fn_extends mem inp.m_star ∧
  ∃ exec : fin (T + 1) → register_state F,
    (exec 0).pc = inp.initial_pc ∧
    (exec 0).ap = inp.initial_ap ∧
    (exec 0).fp = inp.initial_ap ∧
    (exec (fin.last T)).pc = inp.final_pc ∧
    (exec (fin.last T)).ap = inp.final_ap ∧
    ∀ i : fin T,
      next_state mem (exec i.cast_succ)
        (exec i.succ)
```

Final thoughts

Related work: formal processor semantics, verification of cryptographic protocols.

Pinocchio compiles runs of *individual programs* to polynomials, unrolling loops, etc.

The next step, verifying specifications for specific Cairo programs relative to the machine semantics, is like conventional verification, with some twists.

Final thoughts

Blockchain and smart contracts are a good target for verification.

- The programs are generally small, and not too complicated.
- People care about the results.

Extra slides

. . .

The internal cryptographic trick

The key idea: efficient verification that one list is a permutation of another.

Given $a_0, \ldots, a_{n-1}$ and $a'_0, \ldots, a'_{n-1}$, how do I convince you that one is a permutation of another?

Consider the polynomial $\prod_{i<n}(a_i - z) - \prod_{i<n}(a'_i - z)$.

If one list is a permutation of the other, this is the zero polynomial.

If not, it has at most n roots.

Pick a random z.

The internal cryptographic trick

The protocol:

- Commit to initial data.
- Pick a random z (with a hash).
- Prove that the polynomial evaluates to 0.

This handles:

- Memory consistency.
- Integer range checks.
- The fact that the memory extends the public partial assignment.

Cairo has other “builtins” like that.

First landmark

We have proved the correctness of the encoding.

- The polynomial constraints are autogenerated from the StarkWare code that produces STARK certificates.
- The final theorem says the constraints guarantee the existences of an execution trace.

This is a landmark, but ultimately we also want to prove that the existence of an execution trace says something about the auction or exchange we are performing.

Verification tasks

We distinguish between “completeness” and “soundness” of particular programs.

Completeness: this program will run to completion. (If something holds, the prover can prove it.)

Soundness: if this program runs to completion, P holds. (If the prover proves something, the result holds.)

Verification tasks

StarkWare isn't as worried about proving termination or completeness.

STARK magic plus Lean will convince users that a program ran to completion.

(And they do lots of testing on their code to make sure it does.)

Since financial transactions depend on the results, however, they do want strong soundness claims.

The programs can be moderately complicated, and there can be hidden data. Users want to know that the prover can't cheat.

Verification tasks

Levels of of a program description:

- Numbers in a field.
- Low-level instructions (four bitvectors).
- Assembly instructions.
- A sort-of-functional programming language. (See the Cairo docs.)

What we have now

Beyond the verification of the encoding, we have:

- A description of the assembly language in Lean, which can be evaluated and compared to the numbers produced by the compiler.
- Verified Hoare-style semantics, i.e. a description of each instruction in terms of the change of state and register assertions.
- Tactics that make it possible to step through a program and add this information to the context.

What we have now

We have modified the compiler to store some extra bits of information at compile time.

From that, we generate:

- Lean descriptions of the assembly code (whose numbers evaluate to the numbers produced by the compiler).
- Auto-generated specifications for the functions, following the Hoare descriptions.
- Auto-generated correctness proofs.
- Hooks for the user to write their own specifications and prove that they follow from the auto-generated ones.

We are now testing the method on applications.