Let’s try this again, for now with infinite fields for simplicity, then we can map to finite ones.

One element type: F

. F

has:

- Addition
- Additive inverse
- Multiplication
- Multiplicative inverse
- Additive identity
- Multiplicative identity
- Equality

All operations satisfy the standard field laws.

Instructions:

- PUSH F
- ADD
- ADD_INV
- MUL
- MUL_INV
- EQ
- BRANCH F
- HALT
- REFLECT
- REPR
- REPLACE_SELF

Two stacks: one for code, one for data. Define S

as these stacks.

One additional append-only stack with an opaque element type, addressable by index. This is used for evaluator functions.

Basic operation loop:

1. Pop the code stack, this is the next instruction.
2. If the instruction is HALT, halt execution and return the data stack.
3. Call the current evaluator function eval_n

(latest item on the evaluator stack). This function is native code and performs some (unknown) transformation on the state.

The initial evaluator function eval_0

is defined as:

1. PUSH, ADD, ADD_INV, MUL, MUL_INV, and EQ behave as you’d expect, operating on the data stack.
2. BRANCH a

consumes and tests the top of the data stack. If the top is 0_F

, do nothing. If the top is 1_F

, drop a

elements in the code stack.

1. REFLECT moves the top of the code stack to the top of the data stack.
2. REPR moves the top of the code stack to the top of the data stack.
3. REPLACE_SELF:
4. Copies the entire current state, saves it to S_0

.

- Define eval_n

as some opaque function such that the result of executing eval_n

is always the same as the result of: * Starting in S_0

- Appending the code stacks
- Appending the data stacks
- Calling the evaluator recursively until it halts
- Reading the data stack
- Starting in S_0
- Appending the code stacks
- Appending the data stacks
- Calling the evaluator recursively until it halts
- Reading the data stack
- Append eval_n

to the evaluator stack

The next loop, eval_n

is called instead.

Some properties which should hold:

- REFLECT and REPR are always inverse.
- They always transform to a format which is known at compile time (eval_0

).

- eval_n

understands how to evaluate terms “compiled” with eval_0 ... eval_{n-1}

(this can be achieved just by using the evaluator stack).