Background

This is a rough sketch of what a “VampIR VM” might look like. Goals of this VM design:

- Abstract (not tied to a specific proof system)
- Minimal (no more complexity than necessary)
- Clean separation of semantic specification and efficiency / optimisation concerns

Basics

The entire VM is parameterized over a totally ordered finite field F

, with:

- Multiplication: *

, with type $F \to F \to F$

- Addition: +

, with type $F \to F \to F$

- Additive identity: 0

, with type F

- Multiplicative identity: 1

, with type F

- Additive inverse: -

, with type $F \to F$

- Comparison: <

(less than, exclusive), with type $F \to F \to \{0 \mid 1\}$

Input program representation

Programs are represented as extended multivariate polynomials over $a_0 \ldots a_n$

in canonical form, with designated variable b

(where $a_0 \ldots a_n$

can be understood as the inputs, and b

can be understood as the output result):

$a_0^3 + 2a_1^2 - 3a_0 + 2 = b$

This is, of course, equivalent to $a_0^3 + 2a_1^2 - 3a_0 + 2 - b = 0$

.

To provide control flow, we introduce two new primitives:

- Comparison $lt(a, b)$

, where: * $lt(a, b) = 0$

if $a < b$

- $lt(a, b) = 1$

otherwise

- $lt(a, b) = 0$

if $a < b$

- lt(a, b) = 1

otherwise

- Branching branch(a, b, c)

, where: * branch(0, b, c) = b

- branch(1, b, c) = c

- branch(_, b, c)

(if a

is neither identity element) is undefined

- branch(0, b, c) = b

- branch(1, b, c) = c

- branch(_, b, c)

(if a

is neither identity element) is undefined

TODO: Consider whether “undefined” is the right semantics here. “Crash” might also be appropriate.

Note: Assuming that a \in {0, 1}

, branch(a, b, c)

is semantically equivalent to ((1 - a) * b) + (a * c)

, and some kind of smart bidirectional transformation between these two should be possible.

Thus, for example,

branch((lt(a_0, a_1), a_2, a_3^2) = b

is a valid program.

Transformation to circuit

In order to transform programs to a circuit, they are first factorized (via some algorithm), resulting in a program of a form

seq_0 * seq_1 * seq_2 ... = b

TODO: Figure out how lt

and branch

interact with factorization.

The factors are ordered in canonical form.

The program is then transformed to a circuit-style DAG, with a_0 ... a_n

as the inputs and b

as the output. There are only five gate types:

- ADD

- MUL

- NEG

- LT

- BRANCH

Transformation to instructions

Now, we have an AST of the form (for example):

data Circuit = Input Nat | Neg Circuit | Add Circuit Circuit | Mul Circuit Circuit | Lt Circuit Circuit | Branch Circuit Circuit Circuit

For sequential execution, this AST can be compiled with SSA to a finite set of registers and a sequence of instructions of the same names, where evaluation is eager, except for branch which is compiled to compare-jump (avoiding execution of the branch not taken).

For example, define the sequential VM state as the tuple (pc, ins, rt)

where

- pc

is the program counter

- ins

is the instruction sequence

- rt

is the register table

Instructions are then:

data Instruction = Neg R R | Add R R R | Mul R R R | Lt R R R | Branch R P P

Each step, the VM reads the instruction at the current pc

, and executes as follows:

- NEG

, ADD

, MUL

, LT

: Perform the appropriate operation, writing to the last (output) register, and increment pc

by 1.

- BRANCH

: Read register R

, and set pc

to the left branch if R = 0

, or to the right branch if R = 1

.

At the end, the output value will be stored in the register assigned to b

.

For parallel execution, disjoint parts of the circuit may be executed in parallel as usual, and compile-time branch prediction would lead to early parallel evaluation of (parts of) both branches.

Note: the optimisation function here is input-value-dependent, so a well-defined solution would depend on a probability distribution over input values)

Mixins

Self-reflection

Add primitives and associated circuit gates + instructions:

- reflect(a)

- reflects a \in F

up to a program PROG

- repr(PROG)
- represents PROG

as a field element a \in F

Future tasks

- Figure out how and when to deal with overflow (seems like this might be hard/annoying)
- Ideally we just do this at the definition stage by using a large enough field
- But otherwise we might have to do intricate things
- Ideally we just do this at the definition stage by using a large enough field
- But otherwise we might have to do intricate things
- Consider different finite fields (of program definitions and execution backends)
- Consider prefixes / sub-fields
- Consider what happens if the field is only partially ordered / what implications this structure has
- Consider a potential non-deterministic VM version where (basically) some of the inputs are undefined