Maclean
A new language for the modern world

John Skaller

March 17, 2023

Contents

Chapter 1

Rationale

For many years binary byte addressable von Neumann machines dominated the computing world. However the world has changed, with increasing demands for security and privacy and trusted access to computing power required in an untrustworthy environment.

With the rise of cryptography and the invention of the block chain, new capabilities have arisen, along with new requirements. The established von Neumann architectures are no longer suitable.

Block chain operations are basically microservices operating in an untrusted environment, where proof that a server faithfully executed a program is required, in addition to the older requirement that the program be correct. Program correctness has been addressed by advanced type systems, but the new breed of system, often represented by a circuit or a virtual machine, no longer uses the same basic types as the traditional signed and unsigned integers of 8, 16, 32 or 64 bits.

Instead, finite fields are required for cryptographic reasons, addressing models vary, and performance is now dominated by the cost of proof generation, rather than the cost of execution.

We still need strong static typing for proofs of correctness, but computation and optimisation must now deal with both execution, and proof generation. Performance of virtual machines is considerably degraded so that the proof generation becomes tractable.

A new language is required, in which at least the basic data types are no longer simple signed or unsigned integers and machine word size addresses, but which can support a more general class of finite rings for conventional computations, and finite fields of hashing and other cryptographic calculations.

In addition, the ad hoc nature of older languages is no longer tolerable, instead

we need a system soldidly based in algebra: ring theory and finite field theory providing the basic types, and category theory providing ways to construct new types from the base, abstraction, polymorphism, and modularity.

1.1 Development outline

The details of our new system can only be established by research and experiment. Nevertheless for a language targetting CairoVM the following plan is proposed.

- Phase 1 The compiler is first written in any combination of languiages, so it supports the required semantics, though perhaps not efficiently.
- Phase 2 The compiler is improved so generated code is more efficient.
- Phase 3 The compiler is improved so the compiler itself is more efficient.
- Phase 4 The compiler is rewritten in its own source language.
- Bootstrap The compiler is applied to itself, generating a compiler which now runs on CairoVM.
- Enhancement Now the compiler and CairoVM both will need to be modified to tune the resulting system to obtain the desired performance outcome.

The front end of the compiler should be as generic (non Cairo specific) as possible because now the language itself becomes the defining standard, abstracting away the back end VM. This then allows the VM to be modified to improve performance without impacting the semantics of existing programs. It is expected that initially new builtins will be added by the experience of using the system will probably lead to significant architectural changes as well.

1.2 Non-determinism

The current Cairo system uses two languages: Cairo language is used for deterministic computations which can be required to generate proofs of execution. However Cairo has a feature where an "oracle" can "guess" the result of computation, and the system merely checks that the guess is correct. At present Python is used for the Oracle.

Neither Cairo nor Python are even marginally suitable as application languages. Cairo is to low level and orientied towards the VM, whilst Python doesn't even have a static type system so hardly counts as a real language at all.

Instead we need something like this:

```
    fun sqrt(x:u32):u32 invariant result^2 = x { ... }
```

Here, the code in the curly braces computed the square root and is an oracle: the result is not found by traced VM execution but by magic. Then the invariant is evaluated by the VM and the evaluation is traced in the generated proof, if it evaluates to true, the oracle was right and it doesn't matter how it guessed the right answer.

If the invariant is omitted, the body of the function is executed by the VM and traced. Conversely, if you have a function without an invariat, one could be added to improve performance.

The key thing is that a single language is used to generate code for both the function body and invariant, and the oracle could be a copy of the VM with proof generation turned off, or a C or Rust program running on a concurrent distributed network. in the tight coupling model the trustworthiness of the communication is easier to establish.

1.3 Ring Theory

The heart of a computer is just a ring called the ALU. We propose a system as follows: a linear ring is a subrange of the integers 0..n-1 with the usual ring operations plus (integer) division and modulus.

The user starts by declaring one or more rings to be *intrinsic* and can then define new rings using any already defined rings, including the intrinusics. These are just the usual operations modulo the ring size, to handle multiplication, the base ring needs to be at least size $(n-1)^2$.

For example you can define `u32` in the instrinsic prime field provided it is big enough, the compiler will generate all the necessary operatiuons.

It is possible to define ring operation given only a smaller ring, using the equivalent of high school long addition, multiplication and division, *provided* we also supply a memory model for addressing.

1.4 Category Theory

We use category theory to create new types from old ones. Our type system must be at least a distributive category. This gives us products (tuples and arrays) and sums (variants).

1.5 Addressing

Addressing is a difficult topic. If you consider memory to be one huge integers, an object can be extracted by dividing by a another large integer and then applying a modulus, these are simply generalisation of the usual shift and mask operations without the power of two requirement.

However the divisor must be about the same size as all of memory, which is not tractable. How is this problem solved on a traditional machine?

The answer is to take its logarithm, which is called an address. However the result causes a loss of information, which then requires memory to be broken into chunks. On a traditional machine the address is 64 bits long and the chunk is 8 bits long. Consequently, we cannot address fine details (such as individual bits), and we also cannot easily span large values.

This problem is solved in C by padding. So we must do the equivalent, only now we must forget binary coding, since our rings and fields and addressing no longer need to be powers of 2.

Note that from the point of view of a circuit the application of an exponential to recover the value which we previously took the logarithm of is done using a decoder which creates a fanout dependent on the base of the logarithm,

Chapter 2

Type System

We start needing four classes of types: sequences, groups, rings, and fields. Each of these type will be considered a finite, linear, subset of the integers from 0 to $n-1$ where n is the size of the set.

2.0.1 Preliminary definitions

Definition 1. A *semi-group* is a set together with an associative binary operation; that is

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

where infix $\cdot$ is taken as the symbol for the binary operation.

Associativity is a crucial property for an operator because it allows concurrent evaluation of arbitrary subsequences of a sequence of operands. For example given the operand expression

$$(((1 \cdot 2) \cdot 3) \cdot 4) \cdot 5$$

we can compute $1 \cdot 2$ and $4 \cdot 5$ concurrently:

$$(1 \cdot 2) \cdot 3 \cdot (4 \cdot 5)$$

then then combine 3 to either the LHS or RHS subterm before performing the final combination. Another way of looking at this property is that the values to be combined can be stored in the leaves of a tree and any bottom up visitation algorithm can be used to find the total combination.

Associativity means you can add or remove (balanced pairs of) parentheses freely. In particular it is common practice to leave out the parentheses entirely.

Definition 2. A *monoid* is a semigroup with a unit u, that is

$$x \cdot u = x \text{ and } u \cdot x = x$$

for all x in the set.

The existence of a unit means you can freely add or remove units from anywhere in your computation.

Definition 3. A *group* is a monoid in which every element has an inverse, that is, for all x there exists an element y such that

$$x \cdot y = u \text{ and } y \cdot x = u$$

where u is the unit of the underlying monoid. For integers of course, the additive inverse of a value is its negation.

Definition 4. An operation is *commutative* if the result is the same with the operands reversed, that is, for all a and b.

$$a \cdot b = b \cdot a$$

A group is said to be commutative if the group operation is commutative.

Commutativity says you can switch the order of children in the tree representation of an expression.

If an operation is also associative, commutative, and has a unit, then the operation is well defined on a set of operands, taking the operation on the empty set to be the unit.

This means irrespective of what data structure you use to hold the values to be combined, and what algorithm you use to scan them, provided you visit each value exactly once, the result of the operation on them is invariant.

Definition 5. A *ring* is a set with two operations denoted by $+$ and $*$ such that the set with $+$ is a group, and the set excluding the additive unit is a monoid, and the following rule, called the *distributive law* holds for all a, b and c

$$a * (b + c) = a * b + a * c$$

If the multiplication operation is commutative then it is called a commutative ring.

2.0.2 The rings $\mathbb{N}_n$

Definition 6. Let $\mathbb{N}_n$ be the subrange of the integers $0..n-1$ with addtion, subtraction, multiplication, division and remainder defined as the natural result modulo n. Then $\mathbb{N}_n$ is a commutative ring called a *natural ring*.

The usual linear order is also defined. Negation is defined by

$$-x = n - x$$

Natural computations prior to finding the modular residual present an issue we resolve by performing these computations in a much larger ring.

Definition 7. The *size* of a finite ring R, written $|R|$, is the number of values of the underlying set.

2.0.3 Representation

Lemma 1. *The C data types*

```
uint8_t uint16_t uint32_t uint64_t
```

with C builtin operations for addition, subtraction, negation, and multiplication are the rings $\mathbb{N}_{2^8}$ $\mathbb{N}_{2^{16}}$ $\mathbb{N}_{2^{32}}$ $\mathbb{N}_{2^{64}}$ *respectively, with the usual comparison operations, unsigned integer division, and unsigned integer modulus.*

Theorem 1. Representation Theorem. *The values of a ring* $\mathbb{N}_n$ *can be represented by values of a ring* $\mathbb{N}_{n^2}$ *and the operations addition, substraction, negation multiplication and modulus computed by the respective operations modulo* n*. Comparisions work without modification.*

In particular we can use `uint64_t` to represent rings of index up to 2^{32}.

2.1 Intrinsics

An instrinsic is a ring or field which is provided natively by the target system. We use the following example to show how:

```
field goldilocks =
  intrinsic size = 2^64 - 2^32 - 2,
    add = primitive cost 1,
    neg = primitive cost 1,
    sub = fun (x,y) => add (x, neg y),
    mul = primitive cost 1,
    udiv = primitive cost 4,
    umod = primitive cost 4,
    udivmod = primitive cost 4,
    recip = primitive cost 1,
    fdiv = fun (x,y) => mul (x, recip x),
    ... // TODO: finish list
;
```

Each of the required operations for a field (or ring if a ring is being specified), must be either implemented in the target natively, in which case the number of execution cycles required must be specified, or is defined in terms of another defined operation, in such a way no cyclic dependencies exist. In the latter case the compiler derives the cost from the definition.

2.1.1 Versions

In version 1, a definition must be given first before it can be used. In later versions, the a dependency checker will ensure completeness and consistency. In still later versions defaults may be provided.

2.2 Derived structures

2.2.1 One step derivations

Once we have one or more intrinsic, we can use the type notation

```
N<n, base>
```

to specify a ring of size n, defined using the already defined ring `base`. The compiler will define all operations automatically, using modular arithmetic etc, provided $n^2 \leq |\mathtt{base}|$, otherwise it will issue a diagnostic error message that the base ring is not large enough and terminate the compilation.

Note, the base ring does not have to be intrinsic.

To define a new field, we need only a ring sufficiently large for the underlying ring operations, however we must define the `recip` operation natively:

```
field nufield = based goldilocks recip = primitive cost 24;
```

2.2.2 Families

All data types derived directly or indirectly on a particular base for a family, Operations with mixed families require the specification of isomorphism between abstractly equivalent types, or embeddings if approproiate. Research is required here to decide how to handle computations with mixed families.

```
to be done
```

2.3 Compact Products

We first provide *compact linear products*. This is a categorical product with the type given by the n-ary constructor like

```
compactproducttype<R0, R1, ... Rnm1)
```

and values like

```
compactproductvalue(v0,v1,... vnm1)
```

Note syntactic sugar is yet to be determined.

All the data types must be in the same family, and the product of their sizes must be less than or equal to the family base size. All the operators are defined componentwise. However sequencing is based on the representation.

Projections and slices are provided automatically

```
    compactprojection<index, base> : T -> Rindex
```

This is a function which extracts the component selected by the index. The index must be a constant.

A generalised projection is also provided which accepts an expression as an argument. Its codomain is the type dual to the product, that is a sum type consisting of the component index and value. We note that the representation is uniform because the constructor arguments are all structures from the same family.

2.4 Arrays

If all the types of a product are the same, it is called an array. An array projecton accepts an expression as an argument, which cannot be out of bounds, since the type of the expression must be the index type of the constructing functor. The array projection is defined by first applying a generalised projection and then applying the smash operator which throws out the constructor of a repeated sum.

Note there is a related operator which retains the index value, so that an iteration will get a index,value pair.

The math is straightforward but the syntax needs to be established.

2.5 Sums

Just as for products, we provide categorical sums including repeated sums, which are the dual of arrays, along with injections functions.

However the sum of rings is not a ring in the category or rings. In the category of types, the operations on the sum of two rings is instead define by the operations on the representation.

Since our rings are cyclic, the sum of N¡3¿ and N¡4¿ behaves like N¡7¿. However note, it is still a proper categorical sum, since we can decode it to extract a value of one of the injection types.

In a sum of rings, addition is precisely addition with carry, and multiplication is modulo the size of the sum (which is the sum of the component sizes).