Maclean
A new language for the modern world

John Skaller

March 13, 2023

Contents

1 Rationale **2**
1.1 Development outline 3
1.2 Non-determinism 3
1.3 Ring Theory 4
1.4 Category Theory 4
1.5 Addressing 4

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,