Maclean
A new language for the modern world

John Skaller

March 23, 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 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 operations.

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.

1.2 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).

However we have a problem. Many of the categories involved in traditional type system theory are infinite. For example a distributive category is one of the most basic essentials for any first order type theory, provding all finite sums and products related by the distributive law.

Such category, whilst nice in theory, is not directly useful in computing because all computers are finite state machines. We are stuck between a rock and hard place. This problem is not merely evident in higher mathematics .. it is intrinsic to the essential requirement for representations of (signed) integers which is clearly intractable.

Traditional languages such as C simply give the wrong results on overflow, whilst some newer languages vainly attempt to ensure a timely program failure occurs.

To solve these problems we need a new mathematical concept which turns out to be a very old concept instead: the idea is that of *limits.* We want to say we compute with a *local approximation* to the integers, whose correctness is bounded by a a representation size n, so we can say that our calculation will indeed be correct, if only we have a large enough n.

In particular, the integers can now be regarded not as a countably infinite set, since such a beast is unrepresentable, but instead as an idealised limit of a sequence of ever larger finite representations.

Similarly, a type category can be regarded not as distributive for unicorns do not exist, but rather as a bounded local approximation and member of a sequence of ever larger representations, which in the limit approaches the structure of an idealised distributive category.

However it is not enough to merely posit such an theoretical solution and continue as before; instead we must in fact be able to compute the bounds required to ensure the correctness of a program, a new and nontrivial task.

1.3 Addressing

Addressing is a difficult topic. If you consider memory to be one huge integer, 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 several classes of types: discrete sets, sequences, totally ordered sets, groups, rings, and fields. Each of these types will be considered a finite subset of the integers from 0 to $n-1$ where n is the size of the set.

One of the *key* ideas behind our type system is that a type is not merely determined by its abstract semantics, in terms of the equality of composition of functions, but is also dependent on the underlying representation.

By suitable constraints on the representation type, the abstract operations of the type being defined can now be generated automatically by the compiler. For example given the ring `u64` all the operations of the ring `u32` can be generated by the compiler: it suffices to write

```
    typedef u32 = N<2^32, u64>;
```

to have a convenient alias for this type. The compiler simply replaces `a * b` in `u32` with `a * b umod 2^32` in `u64`. It can do this because the rules of algebra are builtin to the compiler, and because `u64` is known to be big enough to implement all the operations of `u64`.

Furthermore the compiler can, potentially, replace `a + b + c` in `u32` with `(a + b + c) umod 2^32`, optimising the computation by removing a `umod` operation, since the sum cannot come even close to exceeding $2^6 4$. Unlike hand implementations, the compiler can generate good code which is guarranteed to be correct.

2.1 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.

2.1.1 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.

2.1.2 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}.

2.2 Type void

The type void is represented by an empty set. Since there no values of this type, it is representation independent; that is, it is a memory of all families. It is also a degenerate memory of all categories.

There are many cases where instantiation of a type variable needs to exclude void. For example a pair of type $A * B$ only has two components if neither is void. If A is void, the type is void, and projections of $A * B- > B$ do not exist.

Nevertheless it is a useful type, for example an array of length zero is the unit tuple and the usual index laws for integers apply: $T^0 = 1$.

It is an open issue how to handle void correctly.

2.3 Type unit

The set $\{0\}$ is the unit type. Since we know the value from the type, the type requires no representation. Instead most uses of a unit value can be eliminated. For example, functions returning unit always return the value 0, so their application can be replace by that value.

2.4 Type bool

Bool is a special type. Although it is not representation independent, comparisons cannot proceed without it. It is, variously, a categorical sum of units, a group, a ring, and indeed a field.

2.5 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.

2.5.1 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.

2.6 Derived structures

2.6.1 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 $n^2 \leq |\mathtt{b}ase|$, 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. One of the jobs of the compiler is to successively reduce operations down a path until intrinsic operations are computed. For example is `u64` is intrinsic, `u32` can be defined in terms of it, `u16` in terms of `u32` and `u8` in terms of `u16`. However `u8` is in fact ultimately in the `u64` family and will use a that as its representation. If the client is targetting a 32 bit machine, they may choose to make `u32` intrinsic as well: this may make the operations more efficient but it will have no impact on the semantics of the program.

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;
```

2.7 Families

All data types derived directly or indirectly on a particular intrinsic base form a *type 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
```

2.8 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.

2.8.1 Ring Products

Definition 8. Let R_i for i in $\mathbb{N}_n$ be a tuple of n finite rings, none of which are void, then the *tensor product* of the rings, denoted by

$$R_0 \otimes R_1 \otimes ... \otimes R_{n-1}$$

is a ring with values tuples of corresponding elements, operations defined componentwise, comparisons defined by the usual lexicographic ordering, and iterators sequencing through values in the defined order.

The size of the ring is the product of the ring sizes.

Theorem 2. Compact Linear Product Representation. *A compact linear product can be represented by a single value* $0..N-1$ *where* N *is the product of the sizes of the rings. The encoding of a value* $(v_0, v_1, ..., v_{n-1})$ *is given by*

$$v_0 * r_0 + v_1 * r_1 + ... + r_{n-1} * v_{n-1}$$

where $r_{n-1} = 1$ and r_k for k in $0..n-2$ is the product of the sizes of the rings R_j for $j > k$:

$$r_k = \prod_{j=k+1}^{n-1} |R_j|$$

where the empty product is 1. That is, the product of the sizes of the rings to the right *of ring R_k in the ring product formula.*

The projection p_k of the k'th ring is given by

$$v/r_k \mod |R_k|$$

where $|R|$ is the size of the ring R.

2.8.2 Product Functors

The constructor of a product is a special kind of type mapping called a *functor*. Category theory requires functors to have certain properties so that they act parametrically. In particular functors map functions from the domain space to the codomain space, not just types.

The most important thing about these functors is the indexing type. Consider the functor

$$ringtuple5 : Ring^5 -> Ring$$

where $5 : Set$ is considered as the discrete set of integers from 0 to 4. This is a commonly used functor which maps 5 ring types into a single ring type, with indexes the constants 0,1,2,3,4.

For example given:

$$ringtuple5(N2, N3, N4, N5, N6) \mapsto N2 * N3 * N4 * N5 * N6$$

then a value you might write

$$proj2 : N2 * N3 * N4 * N5 * N6 -> N4$$

and for a value extraction

$$proj2(1, 2, 3, 4, 5) = 3$$

with appropriate types assumed. This is just pseudo code, the point here is there is a discrete projection for each component.

This is necessary for this kind of projection, because the type of each projection differs.

There are two projections which are more advanced.

2.8.3 Generalised products

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.

This kind of projection can be indexed by a ring or even a field because now we have made the projection type uniform.

2.8.4 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.

That index type therefore, to allow index calculations, could be, for example a ring. This gives us random access to the array.

In fact we can also generalise array projections, so the result consists of both the input index and value. It's very useful in constructions like

```
    for key,value in a ...
```

because a plain iteration through the values is often not enough, since the index is implicit and hidden by the loop.

2.9 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).

Chapter 3

Categories

Categories, or *kinds* are used to provide structure to collections of types. A category can be used as a *constraint* on a type varible which has two implications. First, use of a type not of that category will lead to a compilation error. More significantly, operators are introduced into the scope of the quantified entity, which allow algorithms to be written in a type safe manner.

During monomorphisation when the type variable is replaced by a monotype, the polymorphic operators will also be replaced by appropriate monotypic operations, this can be done safely with no risk of a type error so the process does not require further type checking.

All our categories provide operators parametrised in two ways: in the abstract, and concretely.

The abstract semantics are based on the mathematics of integers, whilst the concrete semantics provide a representation which allows the actual implementation in a finite environment, and thus provides bounds.

The abstract semantics are all well known basic mathematics, however the proper handling of the bounds remains an open issue.

The concept is as follows: consider a category bounded by size n, then some operations may fail due to insufficient capacity of the representation. Therefore we say the concrete category is a *local approximation* to the mathematical abstraction, and provide the argument that for sufficiently large n any given operation will work correctly.

In other words, in the limit, the local approximations provide the exact semantics that would be provided by the abstract category of countably infinite size based on the integers.

3.1 Category set

The category set contains types considered as discrete collections of values, admitting only comparison for equality as an operation. The primary operation on a mathematical set is membership, which together with other set axioms is equivalent to equality.

A set can be represented by any type, provided it is sufficiently large. One common use of such weak types is as indexes to the tuple constructor. This is because, for example, sequencing through the indices in a loop would not product projections of a uniform type.

```
    category set[T]
      eq: T * T -> 2
      ne: T * T -> 2
```

3.2 Category seq

In this category we can iterate through elements of the underlying integral representation within the bounds of the abstract type. The category corresponds roughly to what C++ calls an input iterator.

It is important to note the implied total ordering is, however not available. This is because such comparisons can be very expensive in proof generating machines using AIR constraints.

Note also, the *next* and *prev* operations are cyclic and cannot fail, in other words, the value which is next afer the last one is the very first one again.

```
    category seq[T] : set[T]
      zero: 1 -> T
      last: 1 -> T
      succ: T * T -> 2
      pred: T * T -> 2
      next: T -> T
      prev: T -> T
```

3.3 Category linear

This category extends sequences to allow observation of a total ordering.

```
    category linear[T] : seq[T]
      lt  : T * T -> 2
      gt  : T * T -> 2
      le  : T * T -> 2
      ge  : T * T -> 2
```

3.4 Category group

The types of this category are additive groups. Recall all our types are just subranges of integers, and the implementation is simply the computation in a bigger group modulo the group size, then it is clear these groups are all Abelian (commitative) and indeed cyclic.

Groups in effect give random access iterators.

```
  category group[T] : seq[T]
    add: T * T -> T
    sub: T * T -> T
    neg: T * T -> T
```

3.5 Category ring

This category adds multiplication and integer division to a group to form a commutative ring with unit.

```
  category ring[T] : group[T]
    one : 1 -> T
    mul : T * T -> T
    udiv : T  * (T - {0}) -> T
    umod : T  * (T - {0}) -> T
    udivmod : T  * (T - {0}) -> T * T
```

3.5.1 Division by Zero

Careful observation of the signature of the integer division and modulus operators above shows that the dividend is not allowed to be zero.

Unlike other languages, this is enforced by the type system. At run time division by zero is impossible.

In order that this be the case, we have to construct the type $T - \{0\}$ which can be done with a conversion or cast together with a *proof* that the argument cannot be zero. the programmer *is required to prove the dividend cannot be zero.*

This is an example of a type discipline called *dependent typing.* In general dependent typing is very hard for programmers, luckily our main primary requirement only involve proving one particular case.

In almost all cases of real code, a programmer *knows* the value cannot be zero and not only that, they *know why* but now they actually *have* to write that proof or the compilation will fail.

To make this tenable in all cases, we provide a simplistic construction which provides that proof automatically: we allow the programmer to pattern match the proposed dividend in the same manner an option type can be matched.

```
match v with
| Zero -> cc0
| Nonzero x -> cc1 x
endmatch
```

where the type of the `Nonzero` constructor is indeed the type of the argument v minus $\{0\}$. Note that in both cases a continuation is invoked: this operation is performed in the dual cofunctional space.

If the programmer likes, however, they can actually write a formal proof sketch. To make this work we need a language for writing proofs, and we need a proof assistant. In the simplest cases, if enough information is available, the assistant can generate a correct proof automatically and the programmer need only write an assertion the argument is non-zero.

If the assistant cannot perform the proof, the programmer will need to provide it more data.

Automatic provers for simple properties of integers are no available and are comprehensible. More advanced proofs required for full dependent typing would be beyond most programmers.

3.6 Category field

Fields are usually intrinsic, since the core operation, computation of the reciprocal and division, must be optimised for a particular field.

Fields are intended for crypto operations and require two hash functions: one hashes a single field value, and the second is intended to combine such a hash with another field value to produce another hash.

```
category ring[T] : group[T]
  recip:  T - {0} -> T
  fdiv : T  * T - {0} -> T
  hash : T -> T
  hash2 : T * T -> T
```

Chapter 4

Modularity

Contrary to popular belief and design, an appropriate unit of modularity is the *routine.*

A basic routine is a labelled lexically contiguous sequence of instructions accepting an argument and with a terminal operation which either halts the routine, or invokes another.

The terminal is a sum type in which the constructors names must be labels of routines, and in which the arguments of the constructor become the argument of the invoked routine. The exit selected is determined by the injection which created the value of the sum type.

In a more conventional and relaxed form, a routine can contain multiple exits possibly selected by a conditional.

In this model, a *subroutine* is a special case of a routine in which the caller passes it's current continuation (the next instruction) as, or as part of, the called routine's argument implicitly. Then the caller can be resumed by use of a return instruction.

A *function* is a special case of a subroutine, in which the caller's suspension contains an unitialised reserved slot for a value which is assigned a value by the return instruction before the called routine terminates and the caller is resumed.

A coroutine is a different special case of a routine which is passed a continuation, but not necessarily the callers continuation. The coroutine can then exit by resuming any suspension.

The usual way to organise coroutines is using a schedular and channels in the style of Tony Hoares communication sequential processes (CSP) but with indeterminate evaluation replacing concurrency.

Central to the execution model is the idea that a routine can suspend itself, and be resumed by others, in particular the notion of continuation passing is fundamental, and functions are merely a special case.

In particular, functions are very weak at handling failure, for example division by zero, so a routine that performs a division should in fact have two continuations to choose from. in this case favouring one as special, which is what happens with a subroutine due to the implicit acceptance of the callers continuation makes the divide by zero exceptional, which is clearly bad design.

Chapter 5

Addressing

One of the core tenants of a von Neumann architecture is that memory is addressable, and programs are stored in memory.

Standard ISA usually involve bytes and words which are rings of size powers of 2^8, and there will be a word size, typically a ring $2^6 4$ which is used to address memory.

We must generalise this model, because many modern VM use finite fields as basic memory elements, and such fields never have a size which is a power of 2 since for a ring to be a field, it must be a product of rings of a size which is a prime number, and 2 is too small for most calculations.

This all leads to the question, exactly what is an address?