Maclean
A new language for the modern world

John Skaller

October 31, 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.

Behind the birth of the crypto world, other new requirements arose. Clearly with the internet itself, there is a requirement for distributed, concurrent processing, along with a need for sophisticated communications. Yet whilst type systems for single threaded local memory access are reasonably advanced there is only flegdling support for typing communications, and only rudimentary understanding of concurrent processes (even at the local level).

Today, early vectorisation in supercomputers, primarily to compute inner prodcts rapidly, has be superceded by more general and capable GPUs, capable of running thousands of threads concurrently. Although these devices were originally for improving graphics display rendering speeds, they have rapidly been adopted for more general applications amendable to parellel processing. Yet mostly they're programmed with a separate language than the CPU, and with very little type safety.

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.

Chapter 2

Category Theory

Definition 1. A *category* is a directed multi-graph whose vertices are called *objects,* and whose edges are termed *arrows* or *morphisms*, together with an equality operator for both objects and arrows, and a partial binary operator called *composition* with two constraints:

1. If A, B, C are objects, and $f : A \to B$ is an arrow from A to B, and $g : B \to C$ is an arrow from B to C, then there exists an arrow $h : A \to C$ such that
$$h = g \circ f = f \odot g$$
where $\circ$ is composition operator in forward notation and $\odot$ is the (vastly prefered) composition operator in reverse notation.

2. For every object X there is an arrow $1_X : X \to X$ called an *identity* arrow, such that for all arrows $f : A \to X$
$$f \odot 1_X = f$$
and for all arrows $g : X \to C$
$$1_X \odot g = g$$

The head object of an arrow is called the *domain* and the tail object the *codomain*.

Note that due to the bijection between opjects and identity arrows we can throw out the object altogether; such a category is called an *arrows only* category.

Example 1. *The collection of all small sets and functions between them is the category* Set. *In this case the objects are sets, and sometimes called types; the arrow are functions.*

Example 2. *For any graph, the collection of all paths on the graph is a category.*

Definition 2. A *subcategory* of a category is a subset of objects and arrows which forms a category. This means, objects and identity arrows correspond, and the set of arrows must be closed under composition.

Definition 3. The *dual*, or *opposite* of a category X is the same category with all the arrows reversed, usually written X^{op}.

Definition 4. A *covariant mono-functor* is a structure preserving map $F : A \to B$ from categopry A to category B, which maps objects to objects and arrows to arrows and which preserves composition; that is,

$$F(f \odot g) = F(f) \odot F(g)$$

Visually this means triangles map to (possibly degenerate) triangles.

A *contravariant functor* reverses the arrows, so that if $f : A \to B$ then $F(f) : F(B) \to A$ and so the structure preservation rule becomes

$$F(f \odot g) = F(g) \odot F(f)$$

A contravariant functor is a covariant functor from the opposite category.

The notion can be extended to include multiple domain catgeories by forming a product and if necessary, dualising a category. A domain categpory has positive variance + if its action is covariant and negative variance (-) if contravariant.

Such a multifunctor is also known as a *type constructor* or simply a *polymorphic type*.

We use category theory [1] to create new types from old ones. Our type system must be at least a distributive category. This gives us products [3] (tuples and arrays) and sums (variants) [2].

2.1 Finiteness

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 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 a 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.

2.2 Type Model

We define two classes of categories.

2.2.1 Structural Categories

Structural categories are full subcategories of *Set* whose types are local, compact subranges of the natural numbers, that is, integers in the range $0..m-1$ for some m. Since the category is full, all functions are available. Structural categories are used for systems programming.

One advantage of structural categories is that; being full subcategories means that full subcategories are easily specified by simply listing a subset of the types: the subcategory then consists of all the arrows between these types.

2.2.2 Abstract Categories

These are finite non-structural non-full subcategories of *Set*. Since they are not full subcategories, some arrows of the corresponding structural categeory of the same size are missing, which is what makes them abstract (abstraction is a matter of forgetting details).

The corresponding structural category is call the *representation.* An abstract category is usually specify by nominating certain arrows call *generators* which form a basis for the category. This is called the *interface.* The interface is then *instantiated* by mapping arrows from the abstraction to the representation using a forgetful functor.

2.2.3 Free Categories

An abstract category specified only by the carestian closure of the generators under composition is said to be *free.* Free categories form an *initial algebra.*. A free category may be regarded as *abstract syntax.*

2.2.4 Constrained Categories

A constrained category is a free category with a non-trival constraint specified by a set of equations. The constraint is a functor which is not an isomorphism. The effect of the constraint is also called a *representation invariant* in which some distinct terms of the initial alegra are *equivalent,* this equivalence is sometimes called *semantics.*

Example 3. *The representation of a rational number is usually constructed by first using a pair of intgers* (p, q) *as the presentation and then adding two constraints:* $q > 0$ *and* $\gcd(p, q) = 1$*, the latter eqivalent to a claim* p *and* q *are relatively prime.*

Chapter 3

Type System

3.1 Theory

Let *Set* be the usual distributive category consisting of all small sets and functions. Let Set_n be the full subcategory containing only sets of size less than n.

Let *Nat* be the full subcategory of *Set* consisting of sets of integers $N_n = [0, n)$ for some n, and let Nat_n be the full category consisting of the sets N_k for $k \in N_n$.

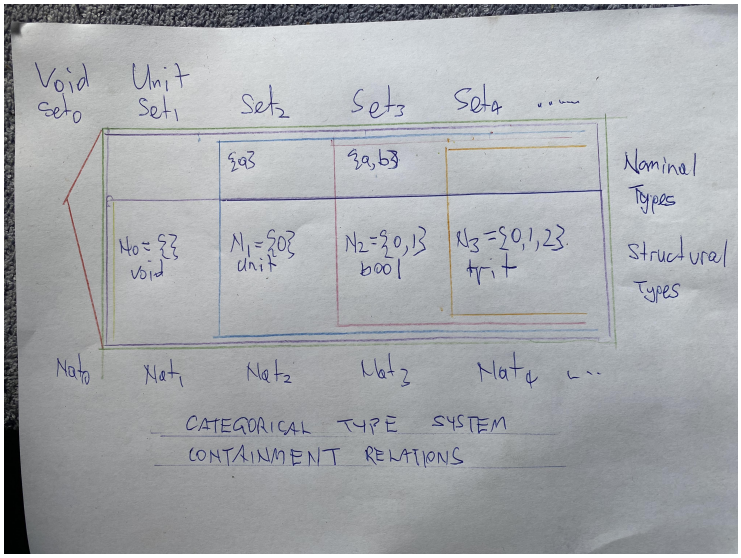

Figure 3.1: Categorical inclusion diagram

The objects of these categories are sets which we will also call *types*.

The category with no objects is called *Void*, with one object, the empty set, is called *Unit*. The category Nat_2 has two objects, the empty set and a set containing only 0, and is also called *Bool*.

The empty set N_0 is also called *void*, and the set N_1 with just $\{0\}$ *unit*, the set with two values, 0 and 1 is called *bool*.

The number of objects in a finite category is given by a the function *size* which is overloaded to also given the number of values of a set, and is written

$$\text{size}X = |X| \tag{3.1}$$

so that in particular

$$|N_n| = n \tag{3.2}$$

The fundamental theorem of our construction is

Lemma 1. *Every object X of size $k < n$ in Set_n is isomorphic to the unique object N_k in Nat_n.*

The basic idea is that we have a functor

$$\text{Rep}_1 : Nat_n \to Set_n, \tag{3.3}$$

where Rep_1 is an isomorphism.

To perform a computation in the image of the functor in Set_n we first apply the inverse functor, perform the calculation using integers in Nat_n and then apply the functor to go back from the representation space to the represented space.

However this is not enough because for example we can have two distinct objects in Set_n of the same size which "behave differently" so we would need a second functor

$$\text{Rep}_2 : Nat_n \to Set_n \tag{3.4}$$

which is also an isomorphism. In fact, the same object can be used in different ways, and so "behave differently" in different contexts.

However, we cannot have this second functor model operations on a single objects behaving "the same way" in each context give different results, these functors must be constrained so that it does not matter which representation is used, we get the same answer. This leads to the following core theorem:

Theorem 1. *Any two representations must be naturally equivalent, in other words, the representation functors must be related by a natural transformation.*

3.1.1 Compact Tuple

Definition 5. *Compact Tuple.* The compact tuple functor has kind

$$\text{Tup} : (Nat - N_0)^{N_n} \to Nat,$$

where the index set is considered as a discrete set. [1] It maps types

$$(N_{i_0}, N_{i_1}, ..., N_{i_{n-1}}) \mapsto N_m$$

where

$$m = \prod_{j=0}^{n-1} j_k.$$

The value map is given by

$$(v_0, v_1, ..., v_{n-1}) \mapsto v_0 r_0 + v_1 r_1 + ... + r_{n-1} v_{n-1}$$

where

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

where the empty product is 1.

The projection functions of type

$$\mathrm{prj}_k : N_m \to N_k$$

of the $k'th$ index is given by

$$v \mapsto v/r_k \mod i_k$$

using Euclidean (integer) division.

The common notation for tuple types is

$$N_{i_0} \times N_{i_1} \times ... \times N_{i_{n-1}}$$

and values are usually written

$$v_0, v_1, ..., v_{n-1}$$

.

When it is clear, we will just write n instead of N_n, for example 5 instead of N_5.

Note that the result of the functor is *NOT* a type: this cannot be emphasised enough. The result is a family of projections with the same index as the functor.

Note

- Category theory explained [3]
- the domain categories have the empty set excluded because its setwise cartesian product with a non-empty set is the empty set, and is not a categorical product due to the non-existence of a projection for the non-empty component.

[1] the index structure is crucial and will be generalised later

- for a domain component of unit type N_1 a projection will be the unique map $N_m \to N_1$ and the result will be the value 0, independent of the constructed value v, since $x \mod 1$ always has value 0. In particular no state is required for storage of the component and our encoding does not allocate any. Nevertheless the encoding of the projection yields the correct result, namely 0.
- if the index set itself is empty, the product exists and is the unit type $N_1 = \{0\}$ and there are no projections so the categorical requirement is satisfied in vaccuo.
- the indexing scheme is big-endian so a component with index 0 has the highest value. This is un-natural mathematically but conforms to the usual Western representation of numbers, lexicographical ordering, and the usual convention in programming languages. Of course we use Arabic numbers, and Arabic is read right to left.
- product formation is not associative: associates are isomorphic but not equal; they have different projections; however, they all have the same domain. For example

  $$5 \times 3 \times 2 \neq (5 \times 3) \times 2 \neq 5 \times (3 \times 2)$$

  have projections $30 \to 5$, $30 \to 3$, $30 \to 2$ and $30 \to 5 \times 3$, $30 \to 2$ and $30 \to 5$, $30 \to 3 \times 2$, respectively.

The construction given is well known and is commonly called a *variadic number system*.

British Monetary System				
name	pounds	shillings	pence	farthings
k	0	1	2	3
i_k	100	20	12	4
r_k	960	48	4	1
Constructor				
v_k	2	3	4	1
$v_k \times r_k$	1920	144	16	1
$v = \sum_0^3 v_k \times r_k$				2081
Projections				
v/r_k	2	43	520	2081
$v/r_k \mod i_k$	2	3	4	1

For example in the British monetary system, 20 shillings is a pound, 12 pence is a shilling, and 4 farthings is a penny. So v0=2 pounds, v1=3 shillings, v2=4 pence, and v3=1 farthing is equal to 2 * (20 * 12 * 4) + 3 * (12 * 4) + 4 * (4) + 1 * (1) farthings. We store 2081 farthings. We extract pounds as 2081 / (20 * 12 * 4) = 2, shillings as (2081 / (12 * 4)) mod 20 = 3. pence as 2081 / (4) mod 12 = 5 and 2081 / (1) mod 4 = 1 farthing where the formulae in parentheses are precisely the r_k above.

Note the first modulo is not required by construction, and the final division will always be by 1. The store of farthings is a compact product, and the extractors of each denomination are projections. The general formulae and category theoretic rationalisation reduces to primary school arithmetic!

3.1.2 Compact Enumeration

In order to understand compact enumerations with arguments we will first start with a special case, the unit sum. A type we would write as 1+1+1 can be thought of as a 3 slot check box, where you mark 1 of the three slots with your choice. The encoding of your choice will simply be its position, 2, 1 or 0. In otherwords the type N_3 can be considered a unit sum.

Now suppose you have a check box that gives two choices, so that if you select the first, you get 3 more choices, and if you select the second, you get 4 choices. Thus can be written (1+1+1)+(1+1+1+1) or more simply 3+4, and is equivalent to having 7 choices. In particular we can just use the value 6 for the first choice, 5 for the second choice, and so on down to 0. More precisely, values in the range 4..6 inclusive represent the first case and 3..0 the second case.

Therefore we can figure out which of the top level two cases are encoded by testing if the stored value is in the inclusive range 4..6 (first case), or 3..0 (second case).

If we find the first case we have to subtract 4 to get the summand. otherwise we have the second case and the value is the answer.

It takes only a second to realise that the argument of a sum type in our system is always a type with k states; that is, N_k for some k, even if it was constructed as a product, the formula will work to find the argument.

Category theorists call this type constructor a sum or coproduct, and programmers usually call them variants, however we will follow Rust and just call them enumerations.

Definition 6. *Compact Enumeration.* The compact enumeration functor has kind

$$\text{Enum} : (Nat - N_0)^{N_n} \to Nat,$$

where the index set is considered as a discrete set. It maps types

$$(N_{i_0}, N_{i_1}, ..., N_{i_{n-1}}) \to N_m$$

where

$$m = \sum_{k=0}^{n-1} i_k$$

.

The injection function index k type

$$\text{inj}_k : N_{i_k} \to Nat_m$$

constructs the enumeration as follows:

$$v \mapsto v + s_k$$

where

$$s_k = \sum_{j=k+1}^{n-1} i_j$$

where the empty sum is 0.

Decoding an enumeration requires extracting both the index and injected value. The index is given by the largest k such that $v \geq s_k$ and the argument is then $v - s_k$.

Note

- the ordering is big-endian, with the operand of the choice in position 0 being given the highest value.
- the required k always exists because $s_{n-1} = 0$.
- as for products we exlude N_0 from the input categories, however for a different reason. There is a unique map from $N_0 = \{\}$ to every set which is a perfectly good injection, but it cannot be applied and so does not construct an enumeration, so has no representation in our encoding scheme. A sum containing voids is well defined but isomorphic to a sum with the voids removed (except if it is the only one), so we will treat that isomorphismn as an equality and exclude void cases.

3.1.3 Compact Arrays

If all the components of a compact product are the same type, we can use a special array functor to construct it.

Definition 7. *Compact Array* The compact array functor has kind

$$\text{Arr} : (Nat - N_0) \times Nat \to Nat,$$

It maps types

$$(N_b, N_n) \mapsto N_m$$

where

$$m = b \times n.$$

The value map is given by

$$(v_0, v_1, ..., v_{n-1}) \mapsto \sum_{k=0}^{n-1} v_k r_k$$

where

$$r_k = b^{n-1-k}$$

The projection functions of type

$$\text{aprj} : N_m \times N_n \to N_b$$

of the $k'th$ index is given by

$$(v, k) \mapsto v/b^{n-1-k} \mod b$$

using Euclidean (integer) division.

The type N_b is the base type of the array, and the type N_n is the index type, the array will be length n. Not that unlike the ordinary projection prj, the aprj operator takes an argument k of the index type N_n, in other words the index can be computed at run time. The reason is that the type of the projection is independent of the index.

3.1.4 Generalised Compact Arrays

It seems a pity we cannot calculate the indices of a tuple at run time because the tuple components all have different types. Luckily there's a way to fix this! We simply unify projection codomains into a coproduct type, namely, the categorical dual of the product type!

Definition 8. *Generalised Compact Array* The generalised compact array functor has kind

$$\text{GArr} : (Nat - N_0) \times Nat \to Nat,$$

It maps types

$$(N_{i_0}, N_{i_1}, ..., N_{i_{n-1}}) \mapsto N_p$$

where

$$p = \sum_{k=0}^{n-1} i_k.$$

The value map is given by

$$(v_0, v_1, ..., v_{n-1}) \mapsto v_0 r_0 + v_1 r_1 + ... + r_{n-1} v_{n-1}$$

where

$$r_k = \sum_{j=k+1}^{n-1} i_j$$

where the empty sum is 0.

The projection functions of type

$$\text{gprj} : N_m \times N_n \to N_p$$

of the $k'th$ index is given by

$$(v, k) \mapsto (v/r_k \mod i_k) + s_k$$

where

$$s_k = \sum_{j=k+1}^{n-1} i_j$$

Note that for example $A \times B \to A + B$, in other words the projections all retun the sum type which is dual to the product type. In particular the usual projection first returns the $k'th$ component which is then injected into the dual sum at the same position. To retrieve the value could do the usual case analysis. however since we know the index it can be retrieved immediately by just subtracting s_k.

3.1.5 Compact Coarrays

A coarray, or repeated sum, is the dual of an array, that is, it is a special case of the enumeration in which all the enumerants have the same type. In this case the injection functions can take a run time argument. Now the sum of the RHS components can be computed by a single multiply, and instead of a linear search through subranges, we can use a binary chop to reduce the decoding time.

In fact, a different encoding could be used consiting of a pair, the index and value which would admid constant time encoding (by just using projections).

[add specs]

3.2 The ALU

We now have almost enough machinery to specify our first device.

Definition 9. An ALU_n is a device with two registers, an instruction register F of size k and a data register A of size n. The instruction register contains an ALU implementation specific encoding of a functin $f : N_n \to N_n$ which is applied to the value in the A register, which is replaced by the result of applying the function to it.

A simple case of an ALU is a *synchronous* ALU which has a clock input. It reads the instruction and data on the rising edge of the clock pulse, and must write the result to the data register prior to the falling edge of the clock.

An *asynchronous* ALU reads its input on an input clock pulse, writes the result to the data register, and then emits an output clock pulse to signify completion.

The key observation to be made, in either case, is that composition of functions is represented temporaly by successive ALU operations.

Let S be a subset of the functions of the type N_n. The collection is said to be a *spanning set* if its transitive closure under composition is the set $Hom(N_n, N_n)$, the set of all functions from N_n to N_n.

A spanning set is said to be a *basis* if the removal of any one function would leave a non-spanning set.

An ALU is said to be *absolutely complete* if the set of instructions it accepts map to a is the set of all functions of the category. Only very small ALUs can be absolutely complete, since the number of functions from a set of size n to itself is n^n.

An ALU is seid to be *temporaly complete* or just *complete* is the set of mapped functions is a spanning set, otherwise the ALU is *incomplete* or *abstract*.

We wish to extend these definitions to the whole category Nat_{n+1} (the $+1$ being required in the notation so that N_n is the largest type.)

We can do this as follows: first, a function with codomain N_c for $c < n$ can be extended to a function with codomain N_n by post composition with the embeddding from N_c to N_n. The beauty of this arrangement is that, since N_c is simply a subset of N_n, and this an integer of N_c is the very same integer of N_N, the operation is merely a type cast to ensure type correctness and has no run time impact; in other words it is has no operation and does not require a an instruction to effect it.

Secondly, if the domain of the function f is N_d, we can replace it with a function g provided $g \restriction_{N_d} = f$ and pre compose another embedding, which as above is a type cast with no run time impact.

Indeed, this second method subsumes the first if $g : N_n \to N_n$.

Thus we can represent functions of the whole category, provided we have suitable function on the largest type. This is not a universal property of categories of course, but peculiar to a specific construction of the *Nat* family.

It is well known, for any category, that given any chain of arrows, appropriate identity arrows can be added at any point, without changing the resulting composite, and, identities can be removed at any point, unless the chain consists of a single identity.

However for the *Nat* family we have a much stronger result: for any category Nat_n the chain of arrows an ALU can implement embedding functions required to ensure type correctness can be ignored when constructing the corresponding instruction stream for the ALU. This is called *type erasure*. In effect it means that instructions are *generic* in the sense that each instruction can represent a whole family of functions.

It is important to note that more than one g can represent f, because the action of g on values outside the domain of f can be anything, since, when g is used to replace f, the argument to g is guarranteed to be in the domain of f and the result of applying it to values outside the domain are irrelevant, since such application cannot occur.

Another way to view this result is to say that an ALU for a category Nat_n can

perform all the *supported* operations for all the contained categories. The qualification *supported* is required in case the ALU can only represent an abstraction; that is, if the instructions do not form a spanning set for the largest type of the largest category.

A particular mapping of functions

$$\text{Repr} : Hom_n(\_, \_) \to Hom(N_n, N_n)$$

used to represent them is called a *representation* of the category Nat_{n+1} and in effect reduces the design space of the ALU to the monoidal subcategory with only the largest type retained.

Chapter 4

Compact Linear Types

The theory presented in the previous chapter appears reasonably general, but in fact it is somewhat crude *because* it starts with subranges of integers, and uses them as indices for functors.

We now present a more general system of mind blowing expressive power. Note that, desipte this, the representational computations are the same.

Definition 10. Given two functors for products and sums with the empty product, or unit, denoted 1, and the empty sum, or void, denoted 0, then, a *compact linear type* is defined recursively as follows:

1. 0 is a compect linear type
2. 1 is a compect linear type
3. any finite product of compact linear types is compact linear
4. any finite sum of compact linear types is compact linear

Note that in fact the first two conditions can be dropped, since they are subsumed by the second two conditions.

A *unitsum* is a special case which is a finite sum of units, that is, a type of the form $1 + 1 + ... + 1$. This is just our original type N_n for the sum of n units.

However there is a significant extension to our original formulation: where previously we used sum and product functors indexed by N_k for some k, we now allow *any compact linear type* as the functor index type.

To understand how radical an upgrade this is, and what it actually means, consider the product $2\times(3\times4)$. Products are not associative so we first construct 3×4 with a bifunctor indexed by N_2, then construct the whole term, with a second bifunctor, also indexed by N_2.

This is clearly inconvenient, we would like to compose the functors so there is only one functor, and we do that *by changing the index type* to $1+(1+1) = 1+2$.

It's important to see that the index is an exponential, and so obeys the usual index laws. In particular the domain category is

$$Nat^{1+2} = Nat \times Nat^2$$

which means the LHS is precisely the functor composite of the RHS. What we have done is *lifted the structure out of the codomain into the domain.* In particular, we have *linearised* the structured type in doing so.

To see what this means, lets consider that the value of the RHS type

$$(1, (2, 3))$$

is in fact

$$(1, 2, 3)$$

in the LHS type. What's more, the *representation* type of both is precsiely N_{24} and iterating through the representation will find all the values, in order, for both types.

In particular, every structured type is an array, with an index that models the type as an exponential. Consider another example:

$$((1, 2, 3), (4, 5, 6))$$

This is an array length 2, of arrays length 3; that is, it has type

$$(N^3)^2$$

However it is equivalent to an single linear array of type

$$N^{2\times 3}$$

with the corresponding value being

$$(1, 2, 3, 4, 5, 6)$$

In particular you can see when we raise an exponential to an index we get a product, and when we raise a product to an index we get a sum. Conversely, if the index of a functor is a sum, the type designated is a product, and if a product it is an exponential.

To put this another we, the index is the *logarithm* of the type. In our array of arrays case, we have converted the type to a matrix so instead of writing $(x.1).2$ for a component we can now write $x.(1, 2)$ and the subcript is now a pair, instead of two successively applied subscripts. An array of arrays is not a matrix, because a matrix does take a single index which is a pair.

Linearisation is an isomorphism, not an identity; but it has the special property that the underlying representation is not changed; that is, the order of array elements is invariant. In code, this means it is a suitable static cast, involving no run time operations. Contrast this with symmetry, which is also an isomorphism, but requires reordering of values in the store.

The interpretation is that:

1. When the functor has an index of type k, the array subscript has type k
2. The element accessed is the representation of k

Here are more examples. Consider

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

The type is

$$N^3 \times N^2$$

This can be represented as a linear array

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

by using a functor index of type

$$3+2$$

This means, you first chose the left summand of type 3 or the right summand of type 2, which is picking which tuple you're interested in, either $(1,2,3)$ of $(4,5)$, respectively.

In the first case, you now pick a value of type 3, and in the second, a value of type 2. In other words, the array index is a variant. To actually compute the array index, we simply use the representation which is 0,1, or 2 in the first case, and 3 or 4 in the second. So the representation value is an index of the original data structure, but *linearatsed* or *flattened*.

Compact linear types were invented with the intention they be used as array indices allowing polyadic array operations; that is, operations on matrices of arbitrary rank and dimension, in a manner independent of the rank and dimension. The rules are the generalistions of the binary index laws:

$$\log(X^n \times X^m) = X^{n+m}$$

and

$$\log X^{nm} = X^{m \times n}$$

Note the reversed order of the second case law. This is necessary so that in sequencing, the n, or inner index moves fastest.

Chapter 5

blah

Neither Set_n nor Nat_n are distributive because not all finite sums and products exist, however the distributive law does hold, provided the relevant sums and products exist. We will say these categories are *locally distributive*.

We note that Nat_m is a full subcategory of Nat_n when $m < n$ which means there is a linear sequence of these categories forming a total order by inclusion. As a consequence if a finite computation is parametrised by the category size a lower bound must be calculated for the category size.

The size of a tuple of type `2 * 4 * 3` is 24, of a sum `2 + 4 + 4` is 9. More generally the size of a type given by an expression involving digits, product symbols, and sum symbols is given by the same expression reinterpreted as an ordinay arithmetic expression.

A second observation is that Nat_n is a full subcategory of Set_n The interpretation is as follows: any type T in Set_n of size k is isomorphic to N_k so all operations in Set_n can be performed by mapping to Nat_n using isomorphisms, doing the operations, and mapping the results back with the inverses. For example to find the next element in an enumeration of size k we can map to N_k, find the next enumerant, and map back to enumeration. Types other than Nat_n are called *nominal types* and can be represented with suitable isomorphisms using *structural types*. Therefore it suffices to work exclusively with the structural type system.

As an example, consider the representation of a rational number as a pair satisfying the usual invariant that the denominator is positive and is relatively prime to the numerator. We can represent these exact value by using Cantor's diagnonal method which proved the number of rational numbers is the same as the number of integers. This accounts for both pairs which do not correspond with any rational because the denominator is zero, and also the fact that all pairs of form (ki, kj) represent the same rational. In practice we simply use ap-

propriate operations in *Nat* to enure the pair is stored in canonical form. The important fact is we have shown that an arbitrary abstraction can be presented by using a structural representation.

To put this another way, all abstractions are simply a non-full subcategory formed by keeping some functions and discarding the rest. Of course this is well understood in principle, where for example in an object oriented system and abstract data type is represented by public methods whose implementation details are private. However, we have generalised these intuitions and presented a rigourous mathematical model of abstraction using the notion of subcategories in category theory. The model transcends not only object orientation but *all* module systems. In particular using category theory we shall see later abstractions can be made polymorphic.

But we are not finished yet because we have not specified how to actually perform the calculations. To handle this problem, we need some more machinery.

Computers have hardware which can perform some computations. Many computers have instructions which can do arithmetic. Therefore what we must do is provide a way to specify which operations can be done, and how to combine them to perform other computations we desire.

The first thing we must do is enrich our type system with constraints. Suppose we want to add two values of type 5. There will certainly be a function `add: 5 * 5 -> 5` that can do this, but how do we find it?

To make this work, we will introduce another subcategory AGroup_n of additive groups sizes 0 to $n-1$ with a specific set of functions including $add : J * J- > J$ and $sub : J * J- > J$ where J is the type of size J. We have to define these functions. To do this, we will pick a type R_k from category Ring_m where $k \geq 2n$ and define

$$add_J(x, y) = J(\text{mod}_R(add_R(Rx, Ry))$$

In this formula, the function R is the embedding $J- > R$ which maps a value of type J to the same value in R. J is the partial inverse mapping back; it's precondition is satisfied due to the modulo operation.

What this formula is doing is specifying operations for a group size n in terms of computations in a ring of size at least $2n$. The minimum size ensures the result cannot wrap around in the ring, and the modulo operation then wraps it around in the group. Although we have written the formula using applications, there is a corresponding formula which is purely compositional.

In other words, we're doing computations for one data type using another sufficiently large one, which is called the *representation type*. Now to completely specify a data type we need to say its kind: is it a group, ring, or field, we need to say its size, and we need to specify its representation type.

Now you may wonder, how then do we calculate in the representation type? Of course, we do that using its representation type! This would lead to an infinite

regress unless we take some action! So we will allow some types to be declared *intrinsic*.

For example common choices for intrinsics would be `u32` and `u64`, the rings with 2^{32} and 2^{64} elements, respectively. These are chosen for the obvious reason the computations are already implement in our target hardware. However, for many crypto VM, we are more likely to make say `goldilocks` prime field intrinsic. In fact for compatibility with existing code almost all crypto VM.

5.1 Categorical Models

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^{64}. Unlike hand implementations, the compiler can generate good code which is guarranteed to be correct.

5.2 Preliminary definitions

Definition 11. 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 12. 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 13. 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 14. 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 15. 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.

5.2.1 The rings $\mathbb{N}_n$

Definition 16. 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 17. The *size* of a finite ring R, written $|R|$, is the number of values of the underlying set.

5.2.2 Representation

Lemma 2. *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 2. 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}.

5.3 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.

5.4 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.

5.5 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.

5.6 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.

5.6.1 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.

5.7 Derived structures

5.7.1 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;
```

5.8 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
```

5.9 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)
```

We will use the syntax something like

```
    let x : R0 \* R1 \* R2 = (v0\, v1\, v3) in ..
```

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, domain, codomain> : index -> domain -> codomain
```

This is a function which extracts the component selected by the index. The index must be of *Set* kind, that is, a constant.

5.9.1 Ring Products

Definition 18. 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 3. 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*.*

5.9.2 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.

5.9.3 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.

5.9.4 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.

5.10 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 6

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.

6.1 category Void

the empty category with no types.

6.2 category Unit

A category with one type, the set {0}.

6.3 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
```

6.4 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
```

6.5 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
```

6.6 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
```

6.7 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
```

6.7.1 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.

6.8 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
```

Bibliography

[1] *Category Theory*. URL: https://en.wikipedia.org/wiki/Category_theory.

[2] *Coproducts*. URL: https://en.wikipedia.org/wiki/Coproduct.

[3] *Products*. URL: https://en.wikipedia.org/wiki/Product_(category_theory).