Maclean
A new language for the modern world

John Skaller

January 14, 2024

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Today, early vectorisation in supercomputers, primarily to compute inner prodcts rapidly, has been superceded by more general and capable GPUs, able to running thousands of threads concurrently.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

2.1 Categories

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 three constraints:

1. *Closure:* 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. *Identity:* For every object X there is a unique 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 $g : X \to C$
$$1_X \odot g = g$$

3. *Associativity:* If $f \to B$, $g : B \to C$ and $h : C \to D$ then
$$(f \odot g) \odot h = f \odot (g \odot h)$$
which as usual means the parentheses may be elided without ambiguity.

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

Note that due to the bijection between objects and identity arrows we can throw out the objects 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 directed graph, the collection of all paths on the graph is a category; provided we add, if necessary, an identity arrow for each vertex.*

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

Definition 3. If D and C are objects of a category, the set of all arrows from D to C is called the *hom set* from D to C and is written

$$Hom(D, C).$$

Definition 4. The *dual*, or *opposite* of a category X is the same category with all the arrows reversed, usually written X^{op}.

2.2 Functors

Definition 5. A *covariant 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,

$$F(f \odot_A g) = F(f) \odot_B F(g)$$

where $\odot_A$ is the composition operator on A, and $\odot_B$ is the composition operator on B.

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

Definition 6. 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.

Definition 7. If A and B are categories, the *product* category, written $A \times B$ consists of pairs of objects and arrows from A and B. The definition can be extended to any finite number of categories.

A functor from a pair of categories is sometimes called a *bifunctor*. In computer science, a functor is also known as a *type constructor* or simply a *polymorphic type*.

2.3 Products

Definition 8. An object I is *terminal* for any object X, there is exactly one function from X to I.

In the categpry of sets, any one element set is terminal. In *Nat* (defined below) the set $\{0\}$ is the unique terminal.

If there is more than one terminal in a category it is sometimes useful to pick one and label it as *canonical* or special and call it *unit*. In Nat, there is only one terminal so unit is unambiguously determined.

Definition 9. Given two objects A and B of some category, a *product* of A and B is an object P called the *product object* and a pair of arrows $\pi_A : P \to A$ and $\pi_B : P \to B$ called *projections*, usually written $A \times B$ such that for any pair of arrows $f_A : X \to A$ and $f_B : X \to B$ there exists a unique arrow $\langle f_A, f_B \rangle : X \to P$ forming the *product of functions with a common domain* such that

$$f_A = \langle f_A, f_B \rangle \odot \pi_A$$

and

$$f_B = \langle f_A, f_B \rangle \odot \pi_B$$

The construction $\langle \_, \_ \rangle$ is a *universal construction* the existence of which, with the above properties, defines what a product is. Note our construction differs from the usual notation in that a product $A \times B$ is a pair of functions (π_A, π_B) with a common domain C, and this is the product (not the common domain C). We emphasise this very strongly because in our system we will have products like 4×4 and 2×8 which are distinct but have the same common domain, namely 16.

Note that the core is the universal and unique existence of the product of the arrow f_A and F_B: that is the constraint on the projections that ensures they form a product. Whilst the product of funcions with a common domain varies according to the choice of functions, the pair of projections is independent of this choice, depending only on the choice of codomains A and B.

The notion can be extended to any finite number of components by selecting an index set J for the categories, and using the same indicies in appropriate positions for the projections and test functions. A common choice is in fact N_n; the subrange of integers from 0 to $n - 1$; in this case the product is called a *tuple*: tuples allow the index to be positionally determined when using notation such as $A \times B$, where the index of the left most component is 0 and increases going from left to right.

Another common kind of index set to use is a finite set of strings called labels, in which case the product is called a *record*.

Every object P is already, trivially, a 1-ary product, with the identity function 1_P as the projection and any chosen arrow $f : X \to P$ is also the required universal arrow.

Every terminal is a nullary product where the universal arrow from X of the product construction is the unique arrow from X required by the definition of a terminal. There are no projections so it satisfies the other requirements in vaccuuo.

Definition 10. An object O is *initial* if for any object X, there is exactly one function from O to X.

In *Set* and *Nat* the empty set is initial. Note the empty function from the empty set to any set: the set of pairs of elements from the domain to codomain is empty, but that is a legitimate set.

As with terminals, it is useful to pick a canonical initial object if there is more than one and call it *void* and sometimes denote it by the symbol 0.

The existence of the void object causes all sorts of issues.

Definition 11. A categopry is *positive* if it does not contain an initial object.

For any category there is a unique full subcategory which is positive formed by removing the initial objects, if any. We will call this *the positive* subcategory.

Definition 12. A category is said to be *Cartesian* if the positive subcategory has products of every finite sequence of objects.

We should note that products are not necessarily unique, even if we fix the index set. Indeed our system will require at least two distinct kinds of tuples, compact ones that fit into a single integer, and non-compact one that span multiple memory locations.

So far we've said how to use products but not create them. For an n-ary product of a particular kind we simply use an n-ary functor to construct the product. The functor is split into two pieces. We first use a domain with only postive subcategories, so there are no products involving initials. Then all the rest of the mapping maps to a particular initial in the codomain. The positive subcategory constraint in the first part ensures an n-functor produces n projections. In particular, products involving initial objects do not exist.

2.4 Coproducts

Coproducts, or *sums* are dual to products. We give the full definition anyhow:

Definition 13. Given two objects A and B of some category, a *coproduct* of A and B is an object S called the *coproduct object* and a pair of arrows $i_A : A \to S$ and $i_B : B \to S$ called *injections*, usually written $A + B$ such that for any pair of arrows $f_A : A \to Y$ and $f_B : B \to Y$ there exists a unique arrow $[f_A, f_B] : X \to P$ forming the *coproduct of functions with a common codomain* such that

$$f_A = i_a \odot [f_A, f_B]$$

and

$$f_B = i_B \odot [f_A, f_B]$$

If all finite coproducts exist, we can define a functor to find the injections. Notice that coproducts are created by the injections, and it remains to figure a way to decode them. WE need at least two pieces of data from the decoder: we need to know *which* injection was used, and also its argument. This is usually done with a case analysis.

Chapter 3

Systems Programming

Definition 14. *Systems programming* is programming exclusively with a bounded subset of natural numbers.

As a consequence, all systems programs must reduce to finite state machines.

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 $k \in N_n$.

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 *Nat₂* 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

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

Note: it is important to understand that the type N_3 in Nat_4 is the *same* type as found in Nat_5. It's not a different type because Nat_4 is a subcategory of Nat_4.

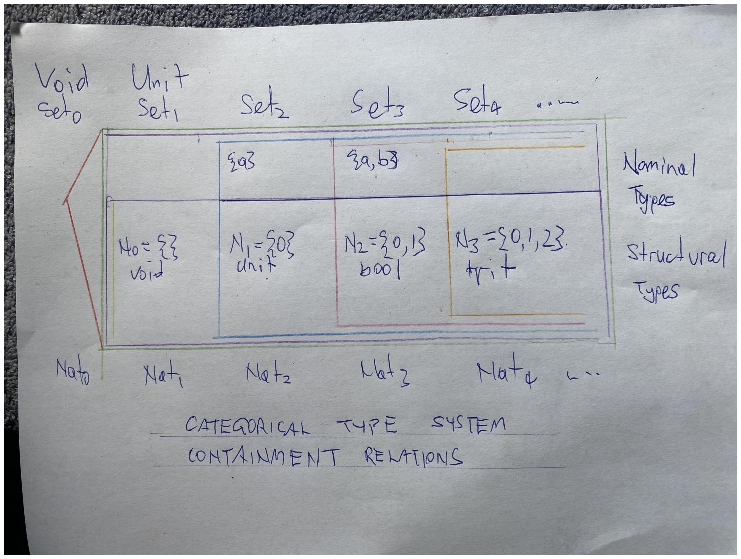

Figure 3.1: Categorical inclusion diagram

Similarly, an integer 3 in N_4 is the *same* integer as the 3 in N_5 even though the type appears different this is not so, because again, they're subtypes. So what this means is that for example 3 has type N_4 and type N_5 and type N_6 etc. In other words, the embedding is the identity map.

It's usual in algebra to use embeddings: for example, the integers are embedded in the rationals, but an integer 3 is *not the same* as the rational 3/1 it corresponds to.

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

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

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

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

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 15. *Compact Tuple.* The compact tuple functor has kind

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

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

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

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

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

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

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 [1]
- 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.
- 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 precisly 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 16. *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 17. *Compact Array* The compact array functor has kind

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

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

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

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

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

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 19. 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 function $f : N_n \to N_n$ 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 the set of all functions of the type. 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 thus 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 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

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

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.

Example 3. *Consider an ALU of size* 2^{64} *with instructions*

```
LDA #999    // load immediate
CLRH        // clear high 32 bits
ADD         // add high 32 bits to low 32 bits
NEG         // negate
MUL         // multiply high 32 bits with low 32 bits
```

These instructions suffice to implement all the operations of a ring of size 2^{32}*. In particular* `CLRH` *mathematically implements modulo* 2^{32}*. Note that the ALU can* NOT *perform ring operations of size* 2^{64} *because addition and multiplication both requires two arguments, which are represented in a single value as a compact product.*

Example 4. Register files. *Most real world processors have multiple registers and instructions which operator on them, for example one might have 8 registers of size 32 bits. Although our ALU model only allows a single register, we can easily consider it to be the compact product of 8 x 32 bit types and design our instruction set accordingly.*

Example 5. Stack machine. *Many VM in the crypto world use a stack instead of registers. In conventional applications this can be inefficient, but the simplification of the instruction set signicantly reduces the complexity of the associate AIR constraints. The run time performance is of little interest since ZK proof generation dominates processesing. Clearly, our ALU model allows for the use of a stack instead of, or as well as, registers in much the same way. However the stack must be bounded to a small size.*

3.3 The MMU

Although in theory there is no limit to the size of an ALU, in practice one step calculations with very large integers are intractable; and this observation extends to computations unpacked to any fixed number of composition steps.

At the heart of the problem is the representation of projections: for a very large integer maintaining state, the divisor of an arbitrary projection is roughly the same size as the state itself. Similarly, to obtain an arbitrarily large chunk of the state, the modulus is similarly sized.

Luckily, we already have a solution: if we are willing to throw out some precision, we can use a logarithm for the divisor an modulus, and have the projection operator implicitly raise them. This is what `aprj` does.

In most conventional machines today, a single array of base type 2^8, called a *byte* is used, with 2^{32} or 2^{64} index values, and this array is called *main memory*. The logarithms, base 2^8, used for projections, are called *addresses*, the sequence of subsequent k bytes is called an *object* where k is the logarithm of the projection modulus. The combination of a run time address with a compile time size is commonly known as a *pointer*.

The key difficulty here is that the digital logarithm loses information; that is, it is not injective. However, quite apart from the large values which can be handled, and desipite the loss of compactness resulting from the loss of injectivity, logarithms have a major advantage, to the point where very few current programming languages have any notion of compact sums and products at all: division, a notoriously expensive operation, can be replaced by subtraction, which is very much cheaper.

Most modern memory systems are little endian (just to confuse you) so in fact we use addition: every C programmer knows that to unpack a struct you can simply add the offset in bytes of the component.

However this leads to a complication in our maths: we now have *two* kinds of products, our original compact products, and now our non-compact ones as well. Furthermore, the programmer no has to make choices about which kinds of products to use.

Example 6. *An array of 64 boolean values has two viable representations: as a single 64 bit integer, or as an array of 64 bytes. In the C++ standard arrays of objects are used for all base types except* `bool` *which has a template specialisation using a compact representation. In fact this was an extremely poor choice because not all the usual array operations work, which means parametric polymorphism fails.*

Example 7. *In the Felix programming language, which has both compact and non-compact products, the author originally tried to have the compiler make the distinction and hide it from the programmer. Alas, the requirement to support polymorphic operations soon showed this could not work. In particular, non-compact data structures can be addressed with ordinary pointers of machine word size, and whilst compact pointers are indeed possible (and implemented!), they require three machine words: an object pointer, along with the usual quotient and modulus, and therefore have a distinct type.*

It is, in fact, worth noting that to preserve generic behaviour (since mathematically a product is a product), the author was pushed into finally introducing a first class kinding system so that type variables used in compact product types could be constrained to accept only compact types.

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 20. 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.

Bibliography

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