Compact Linear Types

John Skaller

October 28, 2022

1 Motivation

During the development of the Felix programming language, an interesting problem arose. Compact linear types were developed to solve this problem, so we will present the system progressively.

1.1 Primitives

Felix has a structural type system based on cartesian closed categories, augmented with pointers. We start with a collection of primitve semi-regular types which are not intrinsic, but instead are lifted with binding constructions from C++. For example

```
type cint = "int";
const zero: cint = "0";
const one : cint = "1";
fun add: cint * cint -> cint = "$1+$2";
proc print: cint = 'printf("%d\n", $1);'
  requires header '#include <cstdio>'
;
print (one + one);
```

1.2 Products

The structural type system has two standard type constructors for sums and products. Products use the familiar C operator `*` for the Cartesian product, and the the comma `,` for tuple value formation:

```
var tuple : int * double = 1, 2.0;
println$ "Tuple " + tuple.str + ", first is " + tuple.0.str;
```

We're using integer literals for component selection, that is, to specify categorical projections. (The `str` operator is a generic function to convert values to a string for printing).

1.3 Tuple Projections

A projection is a function with properties required by category theory. For a tuple of type `int * double * string` the projection function to find the second, double, component, must have type:

```
int * double * string -> double
```

Felix has a way to write this projection function:

```
proj 1 of int * double * string
```

So to grab the double component we do this:

```
var a : int * double * string = 1, 2.4, "Hello";
var d = (proj 1 of int * double * string) a;
  // apply projection
var d2 = a . (proj 1 of int  double * string);
  // reverse application
```

The dot notation is precisely reverse application, it has nothing to do with ”componant access” as it does in C. It looks like it does, but this is a deliberate trick to make it easy to learn Felix.

These projections are builtin to the compiler because they're a fundamental part of the structural type system. Now I'm going to digress temporarily to show what happens with structs, a nominal type:

```
struct X { i:int; d:double; s:string; }
var x = X(1, 2.4, "hello");
var d2 = (d of X) x;
var d2 = x. (d of X);
var d3 = d x; // HUH??
var d4 = x.d; // HUH??
```

Notice the projections have the same form as for tuples except we use the component names instead of numbers. But what that third and fourth case?

Of course the third case is *obviously* overload resolution! There could be more than one function `d`, but only one accepts an argument of type X. And so the fourth case follows, it's just reverse application!

Now Felix is all about *symmetry* and so by symmetry:

```
      var x : int * double * string = 1, 2.4, "Hello";
      var d = (proj 1 of int * double * string) a;
        // apply projection
      var d2 = a . (proj 1 of int ( double * string);
        // reverse application
      var d3 = 1 a; // overload resolution
      var d4 = a.1; // with sugar!
```

We're just allowing an integer constant to imply a projection when the argument is a tuple, and to select the projection function based on the tuple type. So this is a little more than mere overload resolution, the 1 literal is actually somewhat more generic.

1.4 Pointer Projections

So far we have only looked at reading tuple components by using projection functions, but now we have to consider how to modify them. First we need to understand Felix model of mutation:

```
    var x = 1;
    storeat(&x, 2); // intrinsic
    &x <- 3; // sugar
    x = 4; // sugar
```

In Felix to store a value somewhere you have to call the `storat` procedure, passing a pointer to the location into which you want to store something, and of course the value you want to store there. The left arrow `<-` is syntactic sugar for a call to this procedure, and the assignment operator `=` is syntactic sugar for storing in a variable which works by taking the address of the variable.

But this does not work:

```
    var x = 1,"Hello";
    x . 1 = "World"; // BZZT!
```

It fails because it would mean this:

```
    &((proj 1 of int * string) x) <- "World";
```

and you cannot take the address of an expression! Only variables are addressable!

But this does work:

```
var x = 1,"Hello";
&x . 1 <- "World"; // WORKS!
```

and here's how it works:

```
var x = 1,"Hello";
(proj 1 of &(int * string) &x <- "World"; // WORKS!
```

The type of that projection term is:

```
&(int * string) -> &string
```

in other words, it takea a pointer to a tuple, and returns a pointer to the selected components. I call this a *pointer projection*. The `proj` operator is *overloaded* to work with pointers to tuples as well as tuples.

Pointer projections are *not* categorical projections. However they obey a commutative diagram showing the law:

```
*(&x.j) = x.j
```

in other words, fetching the value of a projection of the address of a tuple is the same as fetching the value directly. Internally, a pointer projection is just the addition of an offset to the base pointer.

I had to introduce pointer projections here because down the track we will show how to construct pointer projections for compact linear types so that we can modify components of variables containing them.

1.5 Sums

Sum types use `+` for the discriminated union, and a simple `case` construction, or a shorthand with a backquite ` for the injection functions:

```
var intcase : int + double = (case 0 of int + double) 42;
var doublecase : int + double = (`1:int + double) 2.1;

proc show(x:int + double) =>
  match x with
  | case 0 x => println$ "Int " + x.str;
  | case 1 x => println$ "Double " + x.str;
  endmatch
;
show intcase;
show doublecase;
```

We're using pattern matching here to decode the values of the sum type.

1.6 Arrays

Now one of the key unifying principles in Felix is that an array is nothing more than a tuple for which all the components have the same type:

```
    var array: int * int * int = 1, 2, 3;
    for i in 0 <..3 perform println$ i,array.i;
```

Now this may seem pretty obvious but an astute reader will immediately spot a serious problem! It just shouldn't work! Why? Because the array index is a variable! And an array is a tuple! So you cannot access a tuple component with a variable, it has to be a constant, since, potentially, the result of such an access could have a different type for each component!

Obviously the compiler is doing something tricky! Yes indeed, here it is again:

```
    var array: int ^ 3 = 1, 2, 3;
    for i in 0 <..3 perform println$ i,array.i;
```

The trick is that `int * int * int` is the same type as `int ^ 3` and the latter is indeed an array type. The compiler always uses an array type constructor when possible because this is the specified canonical form.

Whew! That's how it works! Yes? Bzzzz! Nope!

There's another problem! The compiler cannot use that internal form because the compiler doesn't know what an integer is! Unlike other programming languages, arrays lengths are not integers.

In fact the problem is even worse than you think, because we want the exponentiation expression to take *type* arguments and integer constants are *not* types.

1.7 Enumerations

How can we fix this! Here's the answer: in the type language, we specify 0 as the name of the empty sum, and 1 as the name of the empty product. There are no values of type 0, and only one value of type 1, namely `()`, the empty tuple. Now we provide a shorthand so that

$$5 = 1 + 1 + 1 + 1 + 1$$

```
    typedef void = 0;
    typedef unit = 1;
    typedef bool = 2;
```

So now, a sequence of decimal digits is indeed a type. However, it is not an integer literal: the value `42_1u` is an integer literal of type `uint` and is not acceptable in the type language.

So now, we have partially solved the problem, but we have a serious problem:

```
    var x : int ^ string = ... // WOOPS!
```

1.8 UNITSUM kind

The exponentiation operator obviously cannot make sense if the exponent can be any type. It has to be one of those sums of units:

```
    typedef array[t,n:UNITSUM] = t ^ n;
    var a: array[int, 3] = 1,2,3;
```

The first solution is illustrated here: the first type variable `t` defaults to kind `TYPE` a general kind of type, but the compiler has a builtin kind `UNITSUM` which only allows types which are sums of units.

The kind `UNITSUM` is a subkind of the kind `TYPE` and acts as a constraint on the type variable n above so the compiler can reject type expressions like `int^string`.

The kind `UNITSUM` is more commonly known as an *enumeration*, but in particular it is a canonical enumeration because it is constructed entirely using basic type constructors of the structural type system.

Ok, so it may seem we're done but this is just the tip of the iceburg! We have sane constructors for linear arrays, but we have to have projections! Unlike tuple projections, array projections can be expressions, provided they have the correct type!

1.9 Array Projections

Now, finally, we can get back to the array case! Here's the real low level code:

```
    var x = 1,2,3;
    var index : 3 = `1:3;
    var x1 = (aproj index of int ^ 3) x;
    println$ x1;
```

There are two things to notice here: first, the constructor for an array projection is named `aproj`. We cannot use a tuple projection, because that only accepts an integer literal. Arrays can be indexed in a loop, so a variable must be used in that case. More generally you can use any expression of the correct type.

So now you notice the type of the variable is 3, the same as the type of the array length. And it's an enumeration or `UNITSUM` type so the value is given as such, namely `` `1:3 ``, the second case of three cases. Which of course selects the second component of an array. The type of the array projection is:

```
      3 -> int ^ 3 -> int
```

It's not only a first class function, it's a higher order function to boot. There's something important to notice: Felix never needs array bounds checks because values of the index type cannot be out of bounds!

Of course .. since an array *is* a tuple you can also use tuple projections as well.

1.10 Coarrays

Coarrays are the dual of arrays, they involve a repeated sum.

```
    var index = `0:3;
    var s = (ainj index of 3**+int) 42;
```

The name `ainj` is a compiler intrinsic used to construct coarray injections. The big advantage of coarrays, dually to arrays, is that the arguments of all injections have the same type. The `ainj` operator can therefore accept an expression to designate the selected injection, and conversely, a coarray can be decoded to extract the injection index and argument with a function.

```
    var index = `1:3;
    var y = (ainj index of 3 *+ int) 42;
    println$ caseno y;
    println$ casearg y;
```

1.11 Iteration

It's possible to iterator over all the values in a unitsum type:

```
    var x = 1,2,3;
    for i in ..[3] perform println$ x . i;
```

which also guarantees completeness as well as correctness: precisely all the indexes are used, no more and no less.

1.12 Motivation: Matices

Now we are ready for the big time! Consider the following code:

```
var a : (int ^ 2) ^ 3 = (1,2),(3,4),(5,6);
for i in ..[3]
  for j in ..[2]
    perform println$ a.i.j;
```

As you can see, we have an array of arrays. The elements are actually stored contiguously so we should be able to access them with a single loop. To do that we first have to use the usual index laws to convert the array of arrays into a matrix.

```
var a : (int ^ 2) ^ 3 = (1,2),(3,4),(5,6);
var b = a :>> int ^ (3 \* 2);
  // coercion to matrix
  // NOTE REVERSED ORDER
for i in ..[3]
  for j in ..[2]
    perform println$ b.(i\,j);
```

Here, we coerce the array of arrays to a linear array with a structured index type. We use the product constructor `\*` and the tuple constructor `\,` because these construct a new kind of type and new kind of value called a *compact linear type*. These operations have the same categorical semantics as ordinary product and tuple formation, however the compact tuple packs the component values into a single natural number.

The coercion we used to convert a to b is called a reshaping coercion. It's clear that we need that

```
(i\,j) = i * 2 + j
```

for the formula to work. The general rule for coercions to and about compact linear types is that the coercion is allowed if, and only if, the corresponding index law for plain natural numbers is valid.

You should note carefully this works too:

```
var sum = 0;
for i in ..[3]
  for j in ..[2] do
    var index = i\,j;
    perform sum += b.index;
  done
```

What we have actually done here is flatten the data into a linear array, but retain the shape information by moving it into the index type. But we can discard that too:

```
    var a : (int ^ 2) ^ 3 = (1,2),(3,4),(5,6);
    var b = a :>> int ^ (3 \* 2);
      // coercion to matrix
    var c = b :>> int ^ 6;
      // coercion to linear index
    for i in ..[6]
      perform println$ b.i;
```

Now we have flattened the index type as well, since the standard laws for natural numbers say $3 * 2 = 6$.

In particular we have done something radical and powerful: we have linearised the computation so it only uses one loop.

However we have cheated: `2\*3` is not a `UNITSUM`! It is, in fact a more general kind `COMPACTLINEAR` which is still less general than `TYPE`. So in particular the array constructor is now written:

```
    typedef array[T,N:COMPACTLINEAR] = T ^ N;
```

We should note now that we not only also provide compact sums with `\+` but also projections, injections, compact arrays with `\^` and compact coarrays with `\*+_`, and we will also provide a new kind of pointer.

1.13 Why reversed order?

You may wonder why the order of the indices appears to be reversed. In fact if you may wonder why tuples use big-endian representation. The reason is that we want, arbitrarily, that in a tuple type like

```
    \var x : 10^3 = `1:10\, `2:10\, `3:10;
```

that the representation by 123, with the first digit being for hundreds because Europeans were rather stupid when adopting Arabic numbers and neglected the fact that Arabic is read from right to left. So in Arabic numbers are little-endian, but in Latin languages those numbers are read left to right, which is backwards.

So in a tuple, with Latin lexicographical ordering, the most significant component comes first, but it is numbered 0, in the zero origin indexing system: in other words the lowest index finds the most significant component.

In turn this means, when iterating, that the right hand term moves fastest; which you can see in a digital clock, with the right hand digit flashing over quickly and the left hand one moving slowly. This means in an array of arrays the natural iteration scans the inner arrays quickly, and the outer ones more slowly. So the outer array indices must come on the left, or first.

It's unfortunate that the mathematics is less natural this way, but the alternative would be to read positional number systems represented in tuples from right to left. I found when debugging test cases this so was way too confusing.

1.14 Distinctions

The primary difference between the two systems is that whilst ordinary products have machine addressable components, compact ones do not. Nevertheless we can still modify components of a compact product using compact pointers!

In the next section we will develop the mathematics which allows the functional operations to work. Components of products can then be made addressable by combining machine pointers with negative compact projections (that is, a type of operation which identifies a "hole" into which a value can be written, rather than an extractor like an ordinary projection). This turns out to need the idea of pointer projections.

I want to note that all of the mathematics here is trivial high school algebra. There is nothing novel in the index laws. It is the application to type systems which appears to be novel by constructing the type system with a sound theoretical basis, the addition of parametric polymorphism gives powerful optimisations such as array polyadic programming (operations on matrices independent of rank) for free.

2 Theory

With just two type constructors for sums and products, we construct a value representation in the natural numbers. We first define some convenient auxilliary functions with comprehensible semantics, because the essence of the raw formulas is emboddied in superscripts and subscript details and the meaning would soon be lost.

2.1 Size function

We define the `size` of a type as follows, using the usual `|.|` notation:

$$|\sum_i X_i| = \sum_i |X_i|$$
$$|\prod_i X_i| = \prod_i |X_i|$$

This just says the size of a sum is the sum of the sizes of its components, and the size of a product is the product of the sizes of its components. Note that by the convention for nullary terms, the size of the empty sum 0 is 0, and the size of the empty product 1 is 1: the first mentioned number is a type whilst the second is a natural number.

2.2 Product Encoding

We can now present the encoding of values of the type system as natural numbers. First, for products the constructor is given by

$$\texttt{rep} : \prod_{\texttt{i=0}}^{\texttt{n-1}} \texttt{X}_\texttt{i} \longrightarrow \mathbb{N}$$
$$(v0, ..., v_{n-1}) \mapsto \sum_{i=0}^{n-1} v_i | \prod_{j=i+1}^{n-1} X_j |$$

and the projections are given by

$$\pi_j(p) = p \operatorname{div}(| \prod_{i=j+1}^{n-1} X_i |) \text{ rmd } |X_j|$$

In other words, we simply divide by the size of the tail of a component in the product to discard terms on the right, and then discard the terms on the left by using the modulus function. The formula is therefore a simple generalisation of bthe usual right shift and mask operations applied to bitfields. Similarly, the construction is little more than the usual formula for variadic radix number systems.

It is clear the projection specified is indeed a categorical projection. The significance, however, is that products are encoded in a single natural number.

2.3 Sum Encoding

For sum types, the constructors are injection functions:

$$v_i \mapsto \sum_{i=0}^{n-1} v_i | \sum_{j=i+1}^{n-1} X_j |$$

The interpretation is that, for component we sequentially assign a subrange of natural numbers correpsonding to the number of values of that component. The formula is the same as for the product constructor except we replace the multiplication of sizes with an addition.

Decoding sums is more difficult. We first define

$$\mathrm{case(i,v)} = \begin{cases} i, v - \sum_{j=i+1}^{n-1} X_j & v \geq \sum_{j=i+1}^{n-1} X_j \\ \mathrm{case(i+1,v)} & \text{otherwise} \end{cases}$$

This function returns both the index number of the injection function as well as its argument. Since both these values are natural numbers, the function happens to be well typed. The actual decode is performed by

$$\mathrm{case(v)} = \mathrm{case(0,v)}$$

We note the recursion must be terminated, since all natural numbers are greater than or equal to zero, the value of an empty quantified sum.

In general, however, sum types cannot be handled by functions. Instead, continuations must be used to accept the injection argument.

The mathematical formulation above if encoded literally has poor performance: it is linear in the number of components. Clearly with preparation a binary chop can be used to fine the subrange of integers in which a particular integer resides, obtaining logarithmic performace. In principle constant time performance can be obtains by a simple lookup array, but this would only be practical for type of a quite small size. However constant time is obtainable for plain enumerations.

2.4 Properties

The most fundamental properties of interest are isomorphism which induce equalities in the representation. If an isomorphism induces an equality, the isomorphism is said to be *natural*.

Definition 1 *An* associator *is a transform that adds or remove parentheses in the expanded form of a type term.*

Lemma 1 *Associators are natural.*

The consequences of this lemma are profound. Consider the isomorphism

$$5 * (4 * 3) * 7 \simeq 5 * 4 * 3 * 7$$

then we have two equivalent ways to extract the 4 term:

$$\pi_0(\pi_0(5 \star (4 \star 3))) = \pi_1(5 \star 4 \star 3)$$

which says the composition of the two projections on the left equals the projection on the right. In other words, projection composition is equivalent to an associator: indeed, associators are categorical natural transformations. In particular in the example we have a formula which says the divisor of the RHS is the product of the divisors of the LHS projections, in other words, we can compute the composite of the projections by multiplication

$$7 * 3 \text{ from composition} = 7 * 3 \text{ from projection formula}$$

Lemma 2 *Insertion or removal of units into a product or voids into a sum is natural.*

2.5 Ordering

So far we have not defined an ordering for our types, but we have constructed them carefully so that we can. The formulas we have given, somewhat unnaturally, are big endian: the decision is arbitrary but is designed so the following ordering is possible:

Definition 2 *The ordering $<$ is imposed on values of a enumeration based on the ordering of the injection indices, for a general sum and product the standard lexicographical ordering is used.*

Lemma 3 *Ordering is natural and total.*

This result has profound implications, and in many senses is the entire purpose of this development. It says, for example, than a pair of iterators used to scan a two dimensional array can be replaced by a single iterator over the array formed by the associator which drops the nesting, and the same values will be scanned in the same order.

What this means is that we suddenly have array polyadic programming, in other words we can use static casts to reshape any array indexed by any compact linear type, to a linear array, without moving any array elements around, and retaining the order in which elements are visited.

2.6 Distributivity

So far we have a category with sums and products and a representation with natural properties. Our aim is to say that for any compact linear type, a large class of operations can be effected by shape changing isomorphisms for which the underlying computations involve only the well known index laws for natural numbers, and, in addition, allow many operations to be performed with a single flat loop.

But we are missing the final result required:

Lemma 4 *The distributive law*

$$X * (Y + Z) \simeq X * Y + X * Z$$

holds and is natural.

Since we are dealing with a subcategory of Set, distributivity is automatic. However some thought is required to prove naturality although it clearly follows from the equality of the representations which itself follows from the distributive law over natural numbers.

We do not present the results for coarrays but they follow from duality.