While the practical applications of the last example do not stretch very far[1], the approach generalizes to the more relevant containers and their associated laws.

In the same vein as the last section, we could define a simple but inefficient array type, and a more efficient implementation using trees. Then we can show that these are equivalent, such that when the simple type satisfies a set of laws, trees will satisfy them as well. We could then start developing all sorts of complex implementations fine-tuned to each operation and figure out how these are equivalent to some simpler type, but let us first take a step back, and investigate how we can make this approach run smoothly in an even simpler example.

Rather than defining inductively defining a container and then showing that it is represented by a lookup function, we can go the other way and define a type by insisting that it is equivalent to such a function. This approach, in particular the case in which one calculates a container with the same shape as a numeral system, was dubbed numerical representations in [**purelyfunctional**], and has some formalized examples in, e.g., [**calcdata**] and [**progorn**]. Numerical representations form the starting point for defining more complex datastructures based off of simpler basic structures, so let us run through an example.

0.1 Numerical representations: from numbers to containers

We can compute the type of vectors starting from .[2] For simplicity, we define them as a type computing function via the "use-as-definition" notation from before. We expect vectors to be represented by

$$lookup$$

where we use the finite type Fin as an index into vector. We define this as

$$finfromsigma$$

The computation of vectors proceeds as follows

$$vectors$$

SIP doesn't mesh very well with indexed stuff, does HSIP help?

Arrays are made to be indexed, but let us list some expectations do this
The implementation of vectors as functions is very straightforward

[1] Considering that ℕ is a candidate to be replaced by a more suitable unsigned integer type when compiling to Haskell anyway.

[2] This is adapted (and fairly abridged) from [**calcdata**]

and clearly satisfies our interface

Again these proofs transport to vectors.

(This computation can of course be generalized to any arity zeroless numeral system; unfortunately beyond this set of base types, this "straightforward" computation from numeral system to container loses its efficacy. In a sense, the n-ary natural numbers are exactly the base types for which the required steps are convenient type equivalences like $(A + B) \to C = (A \to C) \times (B \to C)$?)

If one was determined to cobble together the path over path over path we need now.

0.2 Numerical representations as ornaments

We could peform the same computation for , which would yield the type of binary trees, but we note that these computations proceed with roughly the same pattern: each constructor of the numeral system gets assigned a value, and is amended with a field holding a number of elements and subnodes using this value as a "weight". But wait! Such modifications of constructors are already made formal by the concept of ornamentation!

Ornamentation, as exposed in [**algorn**] and [**progorn**], lets us formulate what it means for two types to have a "similar" recursive structure. This is achieved by interpreting (indexed inductive) datatypes from descriptions, between which an ornament is seen as a certificate of similarity, describing which fields or indices need to be introduced or dropped. Furthermore, a one-sided ornament: an ornamental description, lets us describe new datatypes by recording the modifications to an existing description.

This links back to the construction in the previous section, since and Vec share the same recursive structure, so Vec can be formed by introducing indices and adding a field holding an element at each node.[3] For this, we first have to give a description of to work with Now we can write down the ornament which adds fields to the suc constructor

Again not sure if it helps to reiterate Desc, Orn, and OrnDesc.

includeme

With the least fixpoint and description extraction from [**progorn**], this is sufficient to define Vec.

Clearly this can use more explanation (the question is, how much?)

Note that we cannot hope to give an unindexed ornament from into trees, since trees have a very different recursive structure! Instead, we must keep track at what level we are in the tree so that we can ask for adequately many elements.

In fact, this "folding in" technique seems to apply rather generally, let us digress.

include this

0.3 Folding in

include this

Let us describe this procedure of folding a complex recursive structure into a simpler structure more generally, and relate this to the construction of binary

[3] These and similar examples are also documented in [**progorn**]

heaps in [**progorn**].