1 From number to container

Perhaps the conclusion from the last section was not very thrilling, especially considering that ℕ is a candidate to be replaced by a more suitable unsigned integer type when compiling to Haskell anyway. More relevant to the average Haskell programmer are containers, and their associated laws.

As an example in the same vein as the last section, we could define a type of inefficient lists, and then define a type of more efficient trees. We can show the two to be equivalent again, so that if we show that lists trivially satisfy a set of laws, then trees will satisfy them as well. But even before that, let us reconsider the concept of containers, and inspect why trees are more efficient than lists to begin with.

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 [purely functional], and is formalized in [calcdata]. Numerical representations form the starting point for defining more complex datastructures based off of simpler basic structures, so let us run through an example.

1.1 Vectors from Peano

We can compute the type of vectors starting from the Peano naturals [this is worked out in full detail in calcdata]. For simplicity, we define them as a type computing function via the "use-as-definition" notation from before. Recall that we expect $VAn = Finn- > A$, so we should define $Finn$ first. In turn $Finn = \Sigma[m \in]m < n$.

Fin : →Type Fin zero = Σ[m ∈] m < zero = Σ[m ∈] ⊥= ⊥Fin (suc n) = Σ[m ∈] m < suc n = 0 < suc n + Σ[m ∈] suc m < suc n = ⊤+ Fin n
Vec : Type →→Type Vec A zero = Fin zero -> A Vec A (suc n) = Fin (suc n) -> A = ⊤+ Fin n -> A = A ×Fin n →A = A ×Vec A nFin : →Type Fin zero = Σ[m ∈] m < zero = Σ[m ∈] ⊥= ⊥Fin (suc n) = Σ[m ∈] m < suc n = 0 < suc n + Σ[m ∈] suc m < suc n = ⊤+ Fin n
Vec : Type →→Type Vec A zero = Fin zero -> A Vec A (suc n) = Fin (suc n) -> A = ⊤+ Fin n -> A = A ×Fin n →A = A ×Vec A n

[SIP doesn't mesh very well with indexed stuff] We can some basic operations [lookup/tail] and show some properties. Again, we can transport these proofs 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 $(A + B)- > C = A- > C \times B- > C$?)

2 Ornaments

2.1 Numerical representations as ornaments

We could vigorously recompute a bunch of datastructures from their numerical representation, but we note that these computations proceed with roughly the same pattern: each constructor of the numeral system gets assigned a value, and is then replaced by a constructor which holds elements and subnodes using this value as a "weight". But wait! The "modification of constructors" is already made formal by the concept of ornamentation!

Ornamentation, exposed in [algebraic ornaments; progorn], lets us formulate what it means for two types to have "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, an 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 Nat and Vec share the same recursive structure, so Vec can be formed by introducing indices and adding a field A at each node.

We can already tell that attempting the same for trees and binary numbers fails: they have very different recursive structures! Still, the correct tree constructors relate to those of binary numbers via the size of the resultant tree. In fact, this relation is regular enough that we can "fold in" trees into a structure which $_can_bedescribedasanornamentonbinarynumbers$.

2.2 Folding in

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 heaps in [progorn].