Advanced Functional Programming: Strongly Typed Vector Implementation

Your Name
1234567

Your Name
9876543

Samuel Klumpers
6057314

February 16, 2022

1 Why

Probably essentially a reimplementation of the functionality in vector-sized and numpy, reverse engineering some things of type-combinators, using constraints. All while avoiding the built-in GHC.TypeNats and their coerced axioms.

The project desires to encode the size of vectors in their type, so that we avoid out of bounds errors or operating on incompatible size vectors.

2 Tricks I used and am now forced to explain

2.1 Singletons

Rather than [should probably use lhs2TeX or so]

```
data Vec a = ...
```

we would like our vector type to mention the size n like:

```
data Vec n a = ...
```

Because n is a type parameter, n should obviously be a type. Unfortunately `n :: Int` is not a type (nor should we expect negative lengths). [Let us ignore Nat because that makes it impossible for us to reason about Nats (unless we force our axioms into existence)] As a result we will construct n ourselves, for this we use the datatype

```
data N = Z | S N
```

Here `Z` encodes 0, while `S n` encodes $n + 1$.

Using the `DataKinds`, we promote data constructors to type constructors. Simply put, `Z` and `S` are promoted to the types `'Z` and `'S`. Particularly, the only value of `'Z` is `Z`, while `'S` takes `n :: 'n` to `S n :: 'S 'n`. This is why we refer to these as **singleton** types.

2.2 GADTs

Not all that tricky or confusing, but the generalization turns out to be necessary.

Our vector type becomes

```
data Vec n a where
    VN :: Vec 'Z a
    VC :: a -> Vec n a -> Vec ('S n) a
```

Similar to `[] :: [a]`, `VN` represents the empty vector, while `VC` prepends an element to the vector.

Note that `Vec` is strictly GADT, since the type depends on the constructor. With this we can already define most size-safe binary operators on `Vec`.

To safely index a vector, we need a type for allowed indices. For this we use `Fin n`, the type representing a set of n elements. If we use shorthand $n = \{0, \dots, n-1\}$, then the `FZ :: Fin ('S n)` constructor asserts that $0 \in n$ for any $n \geq 1$, while `FS :: Fin ('S n) → Fin ('S ('S n))` asserts that if $n \in m$, then $n+1 \in m+1$.

2.3 Dependent types

Suppose we wanted an analogue for `np.full`, and we tried to write down the type:

```
full :: a -> N -> Vec a n
```

This is not going to work, because n would be unbound, while it should be specified by the `N` argument. Unfortunately, Haskell is not Agda, so we cannot write something like `a → (n :: N) → Vec a n`. We will need something to carry n on the type level, so we define

```
data Nat n where
    NZ :: Nat 'Z
    NS :: Nat n -> Nat ('S n)
```

mirroring `N`. With this we can define `full :: a → Nat n → Vec a n`, so that the `Nat n` argument both binds and represents the value n.

2.4 Known/representable

Clearly we cannot define a (meaningful) function `N → Nat n`, otherwise we would not have had to introduce `Nat` in the first place.

However, this also obstructs defining `size :: Fin n → Nat n`, as pattern matching `FZ :: Fin 'n` does not provide a value `n :: 'n`.

The common solution, expounded on the master branch, uses class

```
class Known n where
    nat :: Nat n
```

and defines the obvious instances for `'Z` and `'S n`. This lets us write `size = nat`, provided we add the constraint `size :: Known n => Fin n → Nat n`.

However, an alternate approach, as on the unknown branch, is to rewrite `Fin n` so that each constructor *does* provide sufficient information to recover n. To me, it is unclear whether we will find a situation that actually forces us to rely on `Known`.

2.5 Constraints

The current implementation of the determinant `det :: (Known n, Num a) => Vec (Vec a n) n -> a` relies on `Known n` to provide indices to compute the signs in the sum of the minors (which is somewhat contrived, because it can also be rewritten to not).

Here, `det` calls `minor` on smaller matrices, which also requires `Known n`. However, `Known n` cannot be deduced from `Known ('S n)`, nor can we add an instance `Known ('S n) => Known n`, as that would be incoherent with respect to the necessary instance `Known n => Known ('S n)`.

We remedy this with the tools provided by the constraints package, which lets us handle instances as values via `Dict :: a => Dict a`. For example, we could convert instances `a => b` into entailments as `a :- b` or equivalently `Dict a → Dict b`.

The key to using `Dict p`, is that when such a value is pattern matched, the instance `p` is brought into scope. If used correctly, one can bring `Known n` into scope precisely when it is not deducible from the context, avoiding overlap and incoherence.

To deduce `Known n` from `Known ('S n)`, we use the lemma

```
know :: Nat n -> Dict (Known n)
know NZ = Dict                  -- 'Z is Known by definition.
know (NS n) = case know n of    -- if n is Known,
    Dict -> Dict                -- then 'S n is Known.
```

stating that if we can construct `Nat n`, then `n` is also `Known`. With this, we can prove the intuitive `Known ('S n) => Known n`

```
down :: Dict (Known ('S n)) -> Dict (Known n)
down Dict = step nat where                 -- given a Nat ('S n),
    step :: Nat ('S n) -> Dict (Known n) -- deconstruct to Nat n,
    step (NS n) = know n                   -- assert that we know n.
```

Now we can apply `x`
`(down Dict :: Dict (Known n))`, which evaluates `x` in the context of `Known n`, while we were only given `Known ('S n)`.

2.6 Type families and operators

Consider vector concatenation `vConc :: Vec n a -> Vec m a -> Vec k a`. Clearly $k = n + m$, however, there is no $+$ on the type level. We see that $+$ should send two types n, m to a type representing their sum.

That is, $+$ is a family of types indexed by n, m:

```
type family (n :: N) + (m :: N) :: N
type instance 'Z + m     = m
type instance ('S n) + m = 'S (n + m)
```

The instances represent the usual addition rules in Peano arithmetic. We can now define `vConc :: Vec a n -> Vec a m -> Vec a (n + m)` in the usual way, while the rules ensure that types like `Vec (n + m) a` behave nicely with respect to `VC` and such.