The Monad.Reader Issue 8

by Brent Yorgey ⟨byorgey@gmail.com⟩
and Conrad Parker ⟨conrad@dl.kuis.kyoto-u.ac.jp⟩

September 10, 2007

Wouter Swierstra, editor.

Contents

Editorial

by Wouter Swierstra ⟨wss@cs.nott.ac.uk⟩

Haskell is on a roll: #haskell has hit 400 users; Simon Peyton Jones was invited to speak at OSCON; the first printing of *Programming in Haskell* has completely sold out; and we can all look forward to reading about Real World Haskell. Given the buzz Haskell has been causing in the blogosphere, I was hardly surprised when I started receiving 'functional spam,' which I thought I'd share:

> Hello XXX@@cs.nott.ac.uk,
>
> R u sick of wearing teh hare shirt? Do women laff at ur tiny monads and say your fully-nerd??? Its time to takcle yur awkward squad and make her VERY happy!
>
> We offer TOP quality lambda shirts delivered to you in full confidentiality. Order now and get theorems for FREE!
>
> `www.cafepress.com/TheMonadReader`
>
> Approved by teh top doctors in the field!!

I think this can only mean that Haskell is on the verge of a breakthrough.

I may not have received as many submissions for this issue as for the last, but I am no less pleased with result: Brent Yorgey describes a very tidy implementation of several combinatorial functions on multisets; Conrad Parker, on the other end of the spectrum, gives a mind-boggling solution to a classic problem, completely implemented in Haskell's type system! I think both articles complement one another quite nicely. I hope you enjoy reading them as much as I did.

Generating Multiset Partitions

by Brent Yorgey ⟨byorgey@gmail.com⟩

Multisets *are a generalization of sets that allow elements to occur more than once. In this article, I present a general algorithmic framework in Haskell for generating the unique* ***partitions*** *of multisets using time linear in the number of partitions. The framework is based on the key observation that multisets can be represented using vectors in $\mathbb{N}^n$. It is general enough that it also lends itself to efficient computation of a number of related combinatorial functions, including vector partitions, integer partitions, and power sets.*

This article is literate Haskell; the reader is encouraged to load it into their favorite interpreter and experiment while reading along. Every example shown in the text was actually computed by the listed code, through the magic of lhs2TeX [1].

Introduction

A few months ago, working on a Project Euler [2] problem (#159, to be precise), I wanted to write a function to generate all factorizations of a given integer $n \geqslant 2$. A factorization of n is simply a nonempty list of integers, all greater than 1, whose product is n. For example, 30 has five distinct factorizations: $30 = 15 \times 2 = 10 \times 3 = 6 \times 5 = 5 \times 3 \times 2$. (Note that we include the trivial factorization $30 = 30$ and consider factorizations equal up to permutation of the factors, e.g. 15×2 and 2×15 are the same factorization.) How can we compute all factorizations of a given integer efficiently?

Consider representing integers by multisets of their prime factors. For example, $30 = \{2, 3, 5\}$, and $12 = \{2, 2, 3\}$. We must use multisets of prime factors rather than sets, since we want to be able to distinguish between, say, $6 = \{2, 3\}$ and $12 = \{2, 2, 3\}$. Given an integer n and its multiset of prime factors, a factorization of n corresponds to a particular grouping of the prime factors into subsets. For example, for $n = 30$, the factorization 10×3 corresponds to the grouping $\{\{2, 5\}, \{3\}\}$.

In general, such a grouping into subsets is known as a **partition**. Formally, a partition of a multiset M, or **multipartition**, is a collection of sub-multisets (usually just referred to as subsets) $T_i \subseteq M$ for which

$$\biguplus T_i = M,$$

where $\uplus$ denotes multiset union; for example, $\{1, 2\} \uplus \{1, 3\} = \{1, 1, 2, 3\}$.

As an example, we can list all the multipartitions of $\{2, 2, 3\}$:

$$\{\{\{2, 2, 3\}\}, \{\{2, 2\}, \{3\}\}, \{\{2\}, \{2, 3\}\}, \{\{2\}, \{2\}, \{3\}\}\}.$$

These correspond to the factorizations $12 = 4 \times 3 = 2 \times 6 = 2 \times 2 \times 3$, and in general it's not hard to see that factorizations of an integer exactly correspond to partitions of its multiset of prime factors.

So, I set out to write some code for generating multiset partitions. This turned out to be rather tricky, and it took me a few days to come up with a good solution. The solution I eventually found, however, ended up being quite general, and serves as a nice case study in the expressiveness of Haskell. I found I was largely able to think "through" the language directly about the problem, without having to do too much translation between my head and the code. (Or perhaps this only means that I think in Haskell, but that wouldn't be such a bad thing either.)

Algorithms for generating multiset partitions have been published previously, by, for example, Knuth [3] (who also notes the correspondence between multisets and vectors in $\mathbb{N}^n$). However, Knuth's implementation uses an imperative paradigm. To my knowledge there have been no published treatments of the multipartition problem in a functional context, although I would gladly welcome correction on this point.

A naïve approach: set partitions

Since multisets are a generalization of sets, let's start by generating set partitions and see how far it takes us. Given a set S, we can generate its partitions as follows: first, choose an arbitrary element $s \in S$; then generate the power set (set of all subsets) of the remaining elements of S; each one of these subsets can be combined with s and a partition of the remaining elements to form a partition of S. That is, for each $T \subseteq (S \setminus \{s\})$, recursively find the partitions of $S \setminus (\{s\} \cup T)$, and add $\{s\} \cup T$ to each to form a partition of S.

For example, let $S = \{2, 3, 5\}$ and choose $s = 2$, so the first subset of every generated partition will contain 2. Next we generate the power set of $S \setminus \{2\} = \{3, 5\}$, namely, $\{\{3, 5\}, \{3\}, \{5\}, \varnothing\}$. Each one of these can be combined with s. Combining the first with s yields our first partition, $\{\{2\} \uplus \{3, 5\}\} = \{\{2, 3, 5\}\}$, since there is nothing left to recursively partition. Combining $\{3\}$ with s yields a first subset of $\{2, 3\}$, and recursively partitioning the remaining $\{5\}$ yields only itself, giving us $\{\{2, 3\}, \{5\}\}$. Combining $\{5\}$ with s similarly gives us $\{\{2, 5\}, \{3\}\}$. Finally, we combine $\varnothing$ with s, giving $\{2\}$ as the first subset; recursively partitioning the remaining $\{3, 5\}$ gives both $\{\{3, 5\}\}$ and $\{\{3\}, \{5\}\}$, and prepending $\{2\}$ gives us the final two partitions, $\{\{2\}, \{3, 5\}\}$ and $\{\{2\}, \{3\}, \{5\}\}$.

Some thought (and/or trying more examples) should convince the reader that this succeeds in generating each partition of S exactly once.

This algorithm is implemented in Listing 1, using lists as a simple representation of sets. (Incidentally, most of the code in this article, including that in Listing 1, was tested with QuickCheck [4]; the QuickCheck properties are not shown but can readily be found in the source.)

```
-- pSet s generates the power set of s, pairing each subset
-- with its complement.
-- e.g. pSet [1,2] = [([1,2],[]),([1],[2]),([2],[1]),([],[1,2])].
pSet :: [a]   -> [([a],[a])]
pSet []     = [([],[])]
pSet (x:xs) = mapx first ++ mapx second where
   mapx which = map (which (x:)) $ pSet xs
   first f (x,y) = (f x, y)
   second f (x,y) = (x, f y)

-- setPartitions S generates a list of partitions of S.
setPartitions :: [a] -> [[[a]]]
setPartitions []     = [[]]
setPartitions (s:s') = do (sub,compl) <- pSet s'
                          let firstSubset = s:sub
                          map (firstSubset:) $ setPartitions compl
```

Listing 1: Computing set partitions.

For example, evaluating `setPartitions [2,3,5]` yields

```
[[[2,3,5]],[[2,3],[5]],[[2,5],[3]],[[2],[3,5]],[[2],[3],[5]]] ,
```

just as we computed before. Now, given a suitable implementation of `factor`, which returns a list of the prime factors of its argument, generating factorizations is straightforward:

```
factorizations2 :: (Integral a) => a -> [[a]]
factorizations2 = map (map product) . setPartitions . factor
```

Let's try `factorizations2 30`:

```
[[30],[15,2],[10,3],[5,6],[5,3,2]] .
```

Lovely! However, there's a problem: as you may have guessed, `setPartitions` doesn't do so well when given a multiset instead of a set. For example, evaluating `setPartitions [2,2,3]` yields

```
[[[2,2,3]],[[2,2],[3]],[[2,3],[2]],[[2],[2,3]],[[2],[2],[3]]] .
```

The two copies of 2 are treated as if they are distinct, and we end up with both `[[2,3],[2]]` and `[[2],[2,3]]`, which are equivalent when considered as multiset partitions. This means that an expression such as `factorizations2 12` will give duplicate results as well:

```
[[12],[6,2],[6,2],[3,4],[3,2,2]] .
```

This certainly won't do. We need to get rid of the duplications, but how? The easiest and most obvious way is to create a new function, `msetPartitions`, which generates set partitions and then culls duplicates:

```
msetPartitions :: (Ord a) => [a] -> [[[a]]]
msetPartitions = nub . map (sort . map sort) . setPartitions
```

Normalizing the partitions with `map (sort . map sort)` before applying `nub` is necessary, since otherwise `nub` will not consider partitions such as `[[2,3],[2]]` and `[[2],[2,3]]` equal. Evaluating `msetPartitions [2,2,3]` yields

```
[[[2,2,3]],[[2,2],[3]],[[2],[2,3]],[[2],[2],[3]]]
```

as expected. With a definition of `factorizations3` using `msetPartitions` in place of `setPartitions`, we can now correctly compute the factorizations of 12:

```
[[12],[2,6],[4,3],[2,2,3]] .
```

Are we done? Some degree of inefficiency is clearly involved here, since throwing away duplicate partitions means wasted work to generate them. The question is, how much inefficiency? The code certainly has conciseness and elegance going for it, so we might be willing to overlook a little inefficiency if it means we can have a concise implementation that still works for all practical purposes.

Unfortunately (you probably saw this one coming), `msetPartitions` turns out to be staggeringly inefficient. The worst case occurs when we have the most possible overlap between elements, namely, a multiset with n copies of the same element. First, since `setPartitions` never examines the elements of its argument, it clearly generates the same number of partitions for such a list as it does for any other n-element list, namely, the nth Bell number ϖ_n. Bell numbers [5, 6] count the number of distinct set partitions of an n-element set, and satisfy the recurrence

$$\varpi_0 = 1; \quad \varpi_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \varpi_k.$$

(Extra credit: how does this recurrence relate to `setPartitions`?) Thus, the first few Bell numbers for $n \geqslant 0$ are $1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147 \ldots$ [7].

On the other hand, the number of distinct multipartitions of such a worst-case multiset is the nth partition number P_n, which counts the number of ways of writing n as a sum of positive integers. Partition numbers [8, 3] have the generating function

$$(1+z+z^2+\cdots)(1+z^2+z^4+\cdots)(1+z^3+z^6+\cdots)\cdots = \prod_{n\geqslant 1}\left(\sum_{k\geqslant 0} z^{nk}\right) = \prod_{n \geqslant 1} \frac{1}{1-z^n},$$

and also satisfy a recurrence discovered by Euler,

$$\begin{aligned} P_0 = 1; \quad P_n &= P_{n-1} + P_{n-2} - P_{n-5} - P_{n-7} + P_{n-12} + P_{n-15} - \cdots \\ &= \sum_{\substack{-\infty<k<\infty \\ k\neq 0}} (-1)^{k+1} P_{n-(3k^2+k)/2}, \end{aligned}$$

if we stipulate that $P_n = 0$ when $n < 0$. The first few partition numbers for $n \geqslant 0$ are $1, 1, 2, 3, 5, 7, 11, 15, 22, 30 \ldots$ [9].

As you can see, Bell numbers grow far more quickly than partition numbers. (This can be shown analytically; ϖ_n grows as $(n/\log n)^n$, whereas P_n only grows as $A^{\sqrt{n}}/n$ for a certain constant A [3].) For example, `msetPartitions` correctly computes the $P_{10} = 42$ unique multipartitions of `replicate 10 1`, but it takes a few seconds to do so, since it must cull duplicates from an initial list of $\varpi_{10} = 115975$. And it isn't long before things become completely hopeless: computing the modest $P_{30} = 5604$ unique multipartitions of `(replicate 30 1)` in this way would require culling duplicates from among the whopping $\varpi_{30} = 846749014511809332450147$ (that's 8.5×10^{24}) generated by `setPartitions`!

So, for all its conciseness, `msetPartitions` won't work after all. We need a way to directly generate all the unique partitions of a multiset, using only time linear in the number of partitions generated.

Multiset partitions are vector partitions

In order to directly generate partitions of a multiset M, we need to impose some sort of ordering on the subsets of M, and to be able to generate them in order.

Using such an ordering, we could guarantee that the subsets in a partition always occur in order, thus ruling out the possibility of partitions occurring multiple times with their subsets arranged differently. Indeed, this is how `setPartitions` works. Considering the elements of a set S (represented by a Haskell list) to be "ordered" by the order of their occurrence in the list, the partitions generated by `setPartitions` always contain strictly increasing subsets in increasing order (where subsets are ordered by their first elements).

We could also order multiset subsets lexicographically, but it is not clear how we could efficiently generate them in order. Considering some elements equal to one another throws a big wrench into things, since we can no longer simply recurse over a multiset one element at a time, as `setPartitions` does.

Suppose we have a multiset M, which has n unique elements. The key insight is that we can represent M by a pair (v, e), where v is a vector in $\mathbb{N}^m$ (that is, an m-tuple of non-negative integers), e is an ordered list of length m, and $m \geqslant n$. In particular, e is a list of distinct elements which are a superset of the unique elements of M, and v records the number of occurrences of each element (possibly zero for elements not occurring in M). For example, the multiset containing $\{6, 5, 7, 4, 4, 5, 4\}$ can be represented by the vector $(3, 2, 0, 1, 1)$ together with the list of elements $[4, 5, 29, 6, 7]$.

Using this representation, multiset union corresponds to componentwise vector addition, which reduces the multiset partition problem to that of finding all sets of vectors in $\mathbb{N}^n$ which sum to a given vector. In other words, multiset partitions are vector partitions in disguise!

As an example, if we represent the multiset $M = \{2, 2, 3\}$ by the vector $(2, 1)$ and element list $[2, 3]$, then the partitions of M correspond to partitions of v, as shown below:

$$\{\{\{2,2,3\}\},\{\{2,2\},\{3\}\},\{\{2,3\},\{2\}\},\{\{2\},\{2\},\{3\}\}\}$$
$$\cong$$
$$\{\{(2,1)\},\{(2,0),(0,1)\},\{(1,1),(1,0)\},\{(1,0),(1,0),(0,1)\}\}.$$

This is good, because it is easy to impose an ordering on vectors, and, as we will see, they can also be efficiently generated in order.

Multisets

We will represent vectors in Haskell as `[Int]`. This is a simplification, but works fine for our purposes. Multisets are represented as discussed previously:

```
type Vec = [Int]
data MultiSet a = MS [a] Vec deriving Show
```

Of course, this Haskell definition of `MultiSet` is too lax; in particular, it doesn't preclude the element list and count vector having different lengths. We could correct this by using a list of pairs instead of a pair of lists; however, most of the time we'll be manipulating the element list and count vector separately, and the headache of constantly zipping and unzipping the two lists simply isn't worth it.

Converting between multisets and regular lists is straightforward (Listing 2). Although an `Eq` instance for the element type is all that is needed for the conversion, an `Ord` instance allows it to be done more efficiently, so we include both as options.

```
fromList :: (Ord a) => [a] -> MultiSet a            -- O(n lg n)
fromList = uncurry MS . unzip . map (head &&& length)
             . group . sort

fromListUnord :: (Eq a) => [a] -> MultiSet a        -- O(n^2)
fromListUnord xs = MS nx counts
  where nx     = nub xs
        counts = map (\elt -> (length . filter (==elt) $ xs)) nx

toList :: MultiSet a -> [a]
toList (MS es cs) = concat $ zipWith replicate cs es
```

Listing 2: Converting between lists and multisets.

For example, here's `fromList [2,3,3,2,3,5]`:

```
MS [2,3,5] [2,3,1] .
```

Now that we can easily convert between lists and our multiset representation, we can get on with the real work: generating vector partitions.

Vectors

First, some notation: if $u, v \in \mathbb{N}^n$, then $u \trianglelefteq v$ indicates that u is componentwise less than or equal to v, and $u \leqslant v$ indicates that u is lexicographically less than or equal to v. In other words, $u \trianglelefteq v$ means that every element of u is no greater than the corresponding element of v, whereas $u \leqslant v$ implies only that u is no greater than v in the first element where they differ. Therefore, $u \trianglelefteq v$ implies $u \leqslant v$, but the converse does not hold. For example, $(1, 2, 7) \leqslant (1, 3, 5)$ (since $2 \leqslant 3$), but $(1, 2, 7) \ntrianglelefteq (1, 3, 5)$ (since $7 \nleqslant 5$).

A few simple utility functions for manipulating `Vec` values are shown in Listing 3. Note that the `Ord` instance for lists is lexicographical. Thus, for `u,v :: Vec`, the Haskell expression `u <= v` corresponds exactly to the notation $u \leqslant v$ introduced earlier. The relation $u \trianglelefteq v$ is implemented by the custom `(<|=)` operator.

```
-- recall that type Vec = [Int].

-- componentwise comparison of vectors.
(<|=) :: Vec -> Vec -> Bool
xs <|= ys = and $ zipWith (<=) xs ys

-- (vUnit v) produces a unit vector of the same length as v.
vUnit :: Vec -> Vec
vUnit []     = []
vUnit [_]    = [1]
vUnit (_:xs) = 0 : vUnit xs

-- (vZero v) produces a zero vector of the same length as v.
vZero :: Vec -> Vec
vZero = map (const 0)

-- test for the zero vector.
vIsZero :: Vec -> Bool
vIsZero = all (==0)

-- do vector arithmetic componentwise.
(.+), (.-) :: Vec -> Vec -> Vec
(.+) = zipWith (+)
(.-) = zipWith (-)
```

Listing 3: Implementation of `Vec`.

Generating vector partitions

Given a vector v, the general method we will use to recursively generate all its partitions is as follows: first, we include the singleton set $\{v\}$ as a special case; then, for each smaller vector v' in some appropriate subset of $\{u : u \trianglelefteq v\}$, recursively generate all partitions P' of $v - v'$, and combine to form the partitions $\{v'\} \cup P'$.

Since the vectors in each P' sum to $v - v'$, adding v' will create a partition of v. All we have left to determine is what the "appropriate subset" of $\{u : u \trianglelefteq v\}$ should be.

First, each partition should contain vectors in lexicographically non-decreasing order; as discussed before, this will guarantee that we don't get duplicate partitions. In order to enforce this restriction as we recurse, we must keep track of a current lower limit v_L, which is the largest (hence, most recent) vector in the current partially built partition. We will only choose vectors v' for which $v' \geqslant v_L$.

Second, we need not bother with vectors v' for which $v - v' < v'$, since in that case it would be impossible to complete a non-decreasing partition starting with v'. In other words, we only need to consider vectors v' which are less than or equal to "half" of v, defined as the vector in the exact middle of the lexicographically ordered list of vectors $u \trianglelefteq v$ (rounding down if the list has an even number of elements).

In summary, we want to choose vectors v' for which $v' \trianglelefteq v$ and $\frac{1}{2}v \geqslant v' \geqslant v_L$. To realize this, we first need a function to compute "half" of a vector v. The implementation of `vHalf` is straightforward, recursively splitting along each dimension until finding one that splits evenly, then copying the remainder of the vector. We also need a function to generate a lexicographically sorted list of vectors which fall within a certain range. The function `withinFromTo` takes three vectors `m`, `s`, and `e`, and generates the list of vectors v for which $m \trianglerighteq v$ and $s \geqslant v \geqslant e$; that is, vectors falling **within** the n-dimensional box framed by the origin and m, starting **from** s, and running down **to** e. Listing 4 puts all of these ideas together.

Let's try an example, and compute the partitions of the vector $(2, 1)$:

```
[[[2,1]],[[1,0],[1,1]],[[0,1],[2,0]],[[0,1],[1,0],[1,0]]] .
```

Looking good! We're almost done, but there is still one loose end to tie up. The single quote at the end of `withinFromTo'` (Listing 4) is not a typo; in fact, `withinFromTo'` is an impostor! It only illustrates the intended behavior of its quote-less cousin used in the definition of `vPartitions`. Although `withinFromTo'` elegantly matches its intuitive definition described earlier, it is inefficient. Just like `msetPartitions`, it generates some vectors only to discard them. Since our goal is to write "productive" code that doesn't do any unnecessary work, we'll actually use a more efficient version at the cost of some readability.

Efficiently enumerating vectors

The real `withinFromTo` is shown in Listing 5. The implementation of this function is tricky to get right; careful attention must be paid to appropriate base cases, especially the case when $s \ntrianglelefteq m$. In such a case we must first "clip" s against m

```
vPartitions :: Vec -> [[Vec]]
vPartitions v = vPart v (vUnit v) where
  vPart v _ | vIsZero v = [[]]
  vPart v vL = [v] : [ v' : p' | v' <- withinFromTo v (vHalf v) vL,
                                 p' <- vPart (v .- v') v' ]

vHalf :: Vec -> Vec
vHalf [] = []
vHalf (x:xs) | (even x)  = (x `div` 2) : vHalf xs
             | otherwise = (x `div` 2) : xs

downFrom :: Int -> [Int]
downFrom n = [n,(n-1)..0]

-- (within m) generates a decreasing list of vectors v <|= m.
within :: Vec -> [Vec]
within = sequence . map downFrom

-- This is a lie.
withinFromTo' :: Vec -> Vec -> Vec -> [Vec]
withinFromTo' m s e =
    takeWhile (>= e) . dropWhile (> s) . within $ m
```

Listing 4: Computing vector partitions.

before proceeding. Otherwise we risk generating vectors v for which $v \leqslant s$ but $v \not\trianglelefteq m$. In the general case, we recurse over each dimension and ensure that the generated vectors stay lexicographically between s and e.

```
clip :: Vec -> Vec -> Vec
clip = zipWith min

withinFromTo :: Vec -> Vec -> Vec -> [Vec]
withinFromTo m s e | not (s <|= m) = withinFromTo m (clip m s) e
withinFromTo m s e | e > s = []
withinFromTo m s e = wFT m s e True True
  where
    wFT [] _ _ _ _ = [[]]
    wFT (m:ms) (s:ss) (e:es) useS useE =
        let start = if useS then s else m
            end   = if useE then e else 0
        in
          [x:xs | x  <- [start,(start-1)..end],
                  let useS' = useS && x==s,
                  let useE' = useE && x==e,
                  xs <- wFT ms ss es useS' useE' ]
```

Listing 5: A better implementation of `withinFromTo`.

A walk in the park

Now that we've descended into the depths, tamed multisets, slain the `Vec` beast, and bludgeoned `withinFromTo` into submission, the climb back out will be nothing but a scenic stroll! We have only to wave one of our magic vector functions, and Things will Happen. Our vector functions are so magical, in fact, that we can use them to compute not just multiset partitions, but a number of other interesting functions as well.

For one, consider the problem of splitting an integer n into a multiset of smaller integers whose sum is n. Such integer partitions can be seen as degenerate vector partitions. In particular, the partitions of the integer n correspond exactly to partitions of the 1-dimensional vector (n). This is implemented by the function `intPartitions`, shown in Listing 6. For example, `intPartitions 5` yields

```
[[5],[2,3],[1,4],[1,2,2],[1,1,3],[1,1,1,2],[1,1,1,1,1]] .
```

```
-- Integer partitions.

intPartitions :: Int -> [[Int]]
intPartitions = map (map head) . vPartitions . return

-- Power sets.

mPowSet :: MultiSet a -> [MultiSet a]
mPowSet (MS elts v) = map (MS elts) $ within v

powSet :: (Ord a) => [a] -> [[a]]
powSet = map toList . mPowSet . fromList

powSetUnord :: (Eq a) => [a] -> [[a]]
powSetUnord = map toList . mPowSet . fromListUnord

-- Partitions.

mPartitions :: MultiSet a -> [[MultiSet a]]
mPartitions (MS elts v) = map (map (MS elts)) $ vPartitions v

partitions :: (Ord a) => [a] -> [[[a]]]
partitions = map (map toList) . mPartitions . fromList

partitionsUnord :: (Eq a) => [a] -> [[[a]]]
partitionsUnord = map (map toList) . mPartitions . fromListUnord
```

Listing 6: Casually waving some magic vector functions around.

It seems to work! As a check, we can also (rather inefficiently!) compute the partition numbers [9] with `map (length . intPartitions) [1..]`:

```
[1,2,3,5,7,11,15,22,30,42,56,77,101,135,176,231,297,385,490] ...
```

In particular, `length $ intPartitions 30` yields `5604` almost instantly. This is clearly an improvement over `length $ msetPartitions (replicate 30 1)`, which theoretically produces the same value but probably wouldn't finish before the heat death of the universe.

Computing power sets is easy, too (Listing 6), since there is a natural bijection between unique subsets of a multiset (e, v) and the set of vectors $u \trianglelefteq v$. For example, here's `powSet [2,2,3,3]`:

```
[[2,2,3,3],[2,2,3],[2,2],[2,3,3],[2,3],[2],[3,3],[3],[]] .
```

Last (but not least!), finding partitions of a multiset (also in Listing 6) is now a simple matter of finding partitions of the representative count vector. Here's `partitions [2,2,3]`:

```
[[[2,2,3]],[[2],[2,3]],[[3],[2,2]],[[3],[2],[2]]]
```

No repeated partitions here!

Factorizations, reloaded

We can finally use the `partitions` function for its original intended purpose, to efficiently generate integer factorizations. In fact, we can use `powSet` to efficiently generate divisors, as well. Listing 7 shows how. (This certainly is not the fastest or most robust factoring code possible, but it isn't really the point. Any implementation of `factor` could be combined with `partitions` and `powSet` in this way.)

For example, we can compute the factorizations of 30:

```
[[30],[15,2],[3,10],[5,6],[5,3,2]] .
```

But that's easy, since 30 has no repeated prime factors. Let's try something like 24:

```
[[24],[6,4],[2,12],[2,2,6],[3,8],[3,2,4],[3,2,2,2]] .
```

Or how about `length $ factorizations 1073741824`?

```
5604
```

```
primes :: (Integral a) => [a]
primes = 2 : 3 : [ p | p <- [5,7..], isPrime p ]

isPrime :: (Integral a) => a -> Bool
isPrime n = all ((/= 0) . mod n) $ upToSqrt n primes

upToSqrt :: (Integral a) => a -> [a] -> [a]
upToSqrt n = takeWhile (<= (round . sqrt . fromIntegral $ n))

factor :: (Integral a) => a -> [a]
factor 1 = []
factor n = factor' n [] primes
  where
    factor' n fs ps@(p:pt)
      | (fromIntegral p > (sqrt . fromIntegral $ n)) = n:fs
      | (n `mod` p == 0) = factor' (n `div` p) (p:fs) ps
      | otherwise = factor' n fs pt

divisors :: (Integral a) => a -> [a]
divisors = map product . powSet . factor

factorizations :: (Integral a) => a -> [[a]]
factorizations = map (map product) . partitions . factor
```

Listing 7: Factorizations as multiset partitions.

Funny, I think I remember seeing that number somewhere recently...

As a final comment, the reader may note that we are actually taking advantage of the fact that multisets and vectors also correspond to monomials. For example, the multiset $\{1, 1, 2, 3, 3, 3, 3\}$ corresponds to the monomial $x_1^2 x_2 x_3^4$. Positive integers can be viewed as monomials over the primes, $2^{\alpha_2} 3^{\alpha_3} 5^{\alpha_5} \dots$, which leads directly to the code in Listing 7. Writing a function to compute general monomial factorizations is left as an exercise for the reader.

Acknowledgments

Thanks to David Amos and Wouter Swierstra for their helpful comments on a first draft of this article.

About the author

Brent Yorgey has a BA in Computer Science from Williams College in Massachusetts, USA, and hopes to begin studying for a PhD in the fall of 2008. He spends entirely too much time chatting in #haskell (as byorgey) when he should be doing other things (such as editing this article).

References

[1] Andres Löh and Ralf Hinze. lhs2TEX. `http://www.informatik.uni-bonn.de/~loeh/lhs2tex/`.

[2] Project Euler. `http://projecteuler.net/`.

[3] Donald E. Knuth. **The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions**. Addison-Wesley Professional (2005).

[4] Koen Claessen and John Hughes. Quickcheck: Automatic specification-based testing. `http://www.cs.chalmers.se/~rjmh/QuickCheck/`.

[5] Eric W. Weisstein. Bell Number. `http://mathworld.wolfram.com/BellNumber.html`. From MathWorld—A Wolfram Web Resource.

[6] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. **Concrete Mathematics: A Foundation for Computer Science**. Addison-Wesley, Boston, MA, 2nd edition (1994).

[7] N. J. A. Sloane. On-line encyclopedia of integer sequences. Sequence #A000110.

[8] Eric W. Weisstein. Partition Function P. `http://mathworld.wolfram.com/PartitionFunctionP.html`. From MathWorld—A Wolfram Web Resource.

[9] N. J. A. Sloane. On-line encyclopedia of integer sequences. Sequence #A000041.

Type-Level Instant Insanity

by Conrad Parker ⟨conrad@dl.kuis.kyoto-u.ac.jp⟩

We illustrate some of the techniques used to perform computations in the Haskell Type System by presenting a complete type-level program. Programming at this level is often considered an obscure art with little practical value, but it need not be so. We tame this magic for the purpose of practical Haskell programming.

Overview

This article discusses an implementation of Instant Insanity, an appropriately named puzzle game. We will first familiarize ourselves with a straightforward Haskell solution to the puzzle, and then translate that solution so that it is evaluated by the Haskell Type System at compile time. By discussing the implementation of a complete solution to a problem, we necessarily provide more than just the flavour of programming in the Haskell Type System.

Rather than simply demonstrating clever tricks, we progressively introduce useful type-level programming features. We introduce constructs similar to the functions, lists, maps and filters of conventional functional programming, and in so doing make clear the basic techniques required to construct programs in this manner. Only after building an understanding of the syntactical oddities and transformations involved in programming in the Haskell Type System do we consider the practical aspects of using these techniques in conventional Haskell programming.

Familiarity with the syntax of the Haskell Type System is a prerequisite for understanding the details of general Haskell programming. What better way to build familiarity with something than to hack it to bits? Through this tutorial, we hope to show that the Haskell Type System is not as scary as it first seems.

Textbook implementation

In a discussion of problem-solving via back-tracking, Bird and Wadler [1] introduce the following puzzle, **Instant Insanity**:

> It consists of four cubes, with faces coloured blue, green, red or white. The problem is to arrange the cubes in a vertical pile such that each visible column of faces contains four distinct colours.

The solution provided in Listing 8 stacks the cubes one at a time, trying each possible orientation of each cube.

Our task is to translate this solution into the Haskell Type System.

Type-Level Implementation

The Haskell Type System is a completely symbolic language with very few amenities. You are swimming in a sea of atomic constants, like **X**. These aren't numeric constants; they do not hold a value. They simply exist, and all you can do is create relations between them. If you want features like numbers and arithmetic, or even boolean logic, then you have to create them yourself.

In order to discuss our implementation, we will treat the Haskell Type System as a programming system in its own right. Although the keywords such as **data** and **class** are named for their purpose in Haskell, when introducing them we will ignore that purpose and provide an alternative interpretation within the Haskell Type System. Only after completing the implementation of Instant Insanity will we relate these concepts back to conventional Haskell programming.

Of course, the Haskell Type System wasn't explicitly designed for programming, and there are some limitations which we'll cover later. But for now, let's jump right in!

Type system features

This tutorial uses the Haskell98 type system extended with multi-parameter typeclasses and undecidable instances. We need to enable some GHC extensions to play with this type-hackery:

```
 $ ghci -fglasgow-exts -fallow-undecidable-instances
```

These are included in your GHC; you just need to pass the options to enable them, as they were not part of the Haskell98 language specification.

```
cubes = ["BGWGBR", "WGBWRR", "GWRBRR", "BRGGWW"]

-- Rotate a cube 90 degrees over its Z-axis, leaving up and down in place.
rot [u, f, r, b, l, d] = [u, r, b, l, f, d]

-- Twist a cube around the axis running from the upper-front-right
-- corner to the back-left-down corner.
twist [u, f, r, b, l, d] = [f, r, u, l, d, b]

-- Exchange up and down, front and left, back and right.
flip [u, f, r, b, l, d] = [d, l, b, r, f, u]

-- Compute all 24 ways to orient a cube.
orientations c =
  [c''' | c' <- [c, rot c, rot (rot c), rot (rot (rot c))],
          c'' <- [c', twist c', twist (twist c')],
          c''' <- [c'', flip c'']]

-- Compute which faces of a cube are visible when placed in a pile.
visible [u, f, r, b, l, d] = [f, r, b, l]

-- Two cubes are compatible if they have different colours on every
-- visible face.
compatible c c' = and [x /= x' | (x, x') <- zip (visible c) (visible c')]

-- Determine whether a cube can be added to pile of cubes, without
-- invalidating the solution.
allowed c cs = and [compatible c c' | c' <- cs]

-- Return a list of all ways of orienting each cube such that no side of
-- the pile has two faces the same.
solutions [] = [[]]
solutions (c:cs) = [c' : cs' | cs' <- solutions cs,
                               c' <- orientations c,
                               allowed c' cs']

main = print $ solutions cubes
```

Listing 8: The Bird-Wadler solution to Instant Insanity

Flipping out

We are going to declare new functions called *all*, *flip*, *map* and *filter*, so we hide the versions in Prelude.

import *Prelude hiding* (*all*, *flip*, *map*, *filter*)

⊥ (Bottom)

As we are working in the type system we don't actually care about the values of any of our variables, only their types. There is one special value which is a member of every type (i.e. it can be "cast" as any type we choose). For historical reasons it is called **bottom**, and in Haskell it is written **undefined**. When it is typeset here, it looks like this: ⊥.

For example:

```
*Main> :type undefined::Int
```

$\bot :: Int :: Int$

This simply confirms that the type of the value $\bot :: Int$ is Int.

$u = \bot$

To shorten expressions later in this article, we have abbreviated **undefined** (⊥) to the variable u. This is often convenient when programming in the Haskell Type System.

Simple types

Ok, let's get started.

We note that there are four possible colours. Rather than using arbitrary characters to represent colours, we make some type-safe definitions.

In this problem, the colours of the faces of cubes are **atomic**. They cannot be divided any further, and it is irrelevant to the problem to define them in terms of any other values. The puzzle is no easier if the red faces are pink instead.

In the Haskell Type System, we use the keyword **data** to introduce atomic constants:

data *R* -- Red
data *G* -- Green
data *B* -- Blue
data *W* -- White

These constants are concrete types, which means that it is possible to instantiate variables in these types. However, as we have not given these types any constructors, the only valid value for each is ⊥. We use them to represent atomic constants in the Haskell Type System.

We can check that ⊥ is a valid value of type R:

```
*Main> :type undefined::R
```

$\bot :: R :: R$

Parameterized types

A cube is a thing that can have six faces. In the Haskell Type System, we use the keyword **data** to introduce such a thing:

data *Cube u f r b l d*

Wait a minute – didn't we just use the keyword **data** to introduce atomic constants? Yes, and *Cube* is also atomic. That is to say, the concept of **a thing that can have six faces** is atomic, in the context of this puzzle. No matter how big your hammer is or how frustrated you are, it's not within the rules of the puzzle to break down a cube into something with more, or fewer, faces.

So, R, G, B, W and *Cube* are all atomic, but they are different **kinds** of thing.

```
*Main> :kind R
```

$R :: *$

```
*Main> :kind Cube
```

$\mathit{Cube} :: * \to * \to * \to * \to * \to * \to *$

In the Haskell Type System, the word **kind** is used to describe the structure of a type, and types must be of the same kind in order to be used in the same way.

Whereas R is a concrete type that we could instantiate variables in, *Cube* is not:

```
*Main> :type undefined :: Cube
<interactive>:1:13:
    Kind error: ‘Cube’ is not applied to enough type arguments
    In an expression type signature: Cube
    In the expression: undefined :: Cube
```

The *Cube* type needs to be applied to **type arguments**, namely $u\ f\ r\ b\ l\ d$. If we substitute concrete types for these arguments, the result is a concrete type. So, one way to think about *Cube* is that it is like a function, but at the type level: it takes **types** as arguments and returns **types**. We say that *Cube* is **parameterized** over the type arguments $u\ f\ r\ b\ l\ d$.

Now we can use the types we prepared earlier, R, G, B, W, as type arguments to the type-level function *Cube* to produce concrete types:

```
*Main> :type undefined :: Cube R R R R R R  -- a red cube
```

$\perp :: \mathit{Cube}\ R\ R\ R\ R\ R\ R :: \mathit{Cube}\ R\ R\ R\ R\ R\ R$

```
*Main> :type undefined :: Cube R G B R G B  -- a colourful cube
```

$\perp :: \mathit{Cube}\ R\ G\ B\ R\ G\ B :: \mathit{Cube}\ R\ G\ B\ R\ G\ B$

```
*Main> :type undefined :: Cube B W G G R G  -- another cube
```

$\perp :: \mathit{Cube}\ B\ W\ G\ G\ R\ G :: \mathit{Cube}\ B\ W\ G\ G\ R\ G$

Type aliases

Now we can define the actual cubes in our puzzle as "outputs" of the type function *Cube*, and in order to write clear code (in the Haskell Type System) we will create some handy aliases. In the Haskell Type System, we use the keyword **type** to introduce type aliases:

A completely red cube:

type *CubeRed* = *Cube* $R\ R\ R\ R\ R\ R$

A completely blue cube:

type *CubeBlue* = *Cube* $B\ B\ B\ B\ B\ B$

The cubes available in the problem at hand:

```
cubes = ["BGWGBR", "WGBWRR", "GWRBRR", "BRGGWW"]
```

type *Cube1* = *Cube* $B\ G\ W\ G\ B\ R$
type *Cube2* = *Cube* $W\ G\ B\ W\ R\ R$
type *Cube3* = *Cube* $G\ W\ R\ B\ R\ R$
type *Cube4* = *Cube* $B\ R\ G\ G\ W\ W$

```
*Main> :kind Cube1
```

Cube1 :: ∗

We see that *Cube1* has the same **kind** as the color red, *R*, has. And indeed, *Cube1* is a concrete type:

```
*Main> :type undefined::Cube1
```

⊥ :: *Cube1* :: *Cube1*

Multi-parameter type classes

We would like to define the following transformations on *Cube*s:

```
rot [u, f, r, b, l, d] = [u, r, b, l, f, d]
twist [u, f, r, b, l, d] = [f, r, u, l, d, b]
flip [u, f, r, b, l, d] = [d, l, b, r, f, u]
```

In the Haskell Type System, we use the keyword **class** to introduce functions. At first, we will simply introduce a collection of plain Haskell functions. These are grouped under the parameterized type class *Transforms*:

```
class Transforms u f r b l d where
  rot :: Cube u f r b l d → Cube u r b l f d
  twist :: Cube u f r b l d → Cube f r u l d b
  flip :: Cube u f r b l d → Cube d l b r f u
```

This collection of declarations defines an interface, and that interface consists of three functions *rot*, *twist* and *flip*. For example, *rot* takes a *Cube* as input, and outputs a different type of *Cube*. The exact types of these *Cube*s are related to the class parameters.

Two different substitutions of the **class** parameters would define different interfaces, as the resulting types of the functions *rot*, *twist* and *flip* would be different. Hence *Transforms* is actually a class constructor – we can use it to generate new classes, but it is not itself a concrete class definition. Programming in the Haskell Type System, we are only interested in the types of these functions, which are dictated by the **class** definition. Accordingly, we don't need to specify anything about their implementation when we create an **instance**; we can simply declare the functions to be ⊥.

For example, we could create an instance of *Transforms* which applies only if all six parameters (faces of cubes) are green:

```
instance Transforms G G G G G G where
  rot = ⊥
  twist = ⊥
  flip = ⊥
```

However, there is no need to instantiate *Transforms* for every possible combination of faces. We can fill in any or all parameters with variables, which can represent any concrete type (of **kind ***):

```
instance Transforms u f r b l d where
  rot  = ⊥
  twist = ⊥
  flip = ⊥
```

We are now able to evaluate some simple type transformations, such as:

```
*Main> :type              rot (undefined::Cube1)
```

rot (⊥ :: *Cube1*) :: *Cube B W G B G R*

```
*Main> :type        flip (rot (undefined::Cube1))
```

flip (*rot* (⊥ :: *Cube1*)) :: *Cube R G B G W B*

```
*Main> :type twist (flip (rot (undefined::Cube1)))
```

twist (*flip* (*rot* (⊥ :: *Cube1*))) :: *Cube G B R W B G*

You can see that we can already perform some basic computations, entirely in the type system.

Now, do you take the red cube, or the blue cube?

Functional Dependencies

So far we have seen how to construct simple types, and perform type transformations that transform a parameterized type into a differently parameterized type.

For this puzzle we will need some boolean algebra, so let's create it. First we make the truth values:

```
data True
data False
```

The first boolean function we will need is *And* that relates three values: two inputs and one output. We express the fact that the output *b* is dependent on the two inputs *b1*, *b2* by adding the dependency annotation $b1\ b2 \rightarrow b$. We use a vertical bar to append this dependency annotation:

```
class And b1 b2 b | b1 b2 → b where
  and :: b1 → b2 → b
```

We can define *And* by simply listing out its complete truth table:

```
instance And True  True  True  where and = ⊥
instance And True  False False where and = ⊥
instance And False True  False where and = ⊥
instance And False False False where and = ⊥
```

When using functional dependencies in this way, we are doing logic programming in the Haskell Type System, using **class** to introduce type-level functions and using the last class parameter as the "output."

Lists

We can define lists in the type system using the following atoms:

```
data Nil
data Cons x xs
```

The data type *Nil* represents the empty list; the *Cons* data type is used to append an element to a list. With this syntax, *Cons x Nil* denotes a list containing only one element; we can use *Cons* multiple times to build longer lists. However, this quickly becomes difficult to read. For example, the list $[R, G, B]$ would be represented by `Cons R (Cons G (Cons B Nil))`.

GHC allows us to introduce an alternative infix notation to represent *Cons*. Type-level infix operators must begin with a colon, so we choose ::: and define:

```
data x ::: xs
infixr 5 :::
```

The **infixr** 5 here sets the precedence level; we make ::: bind tightly so that we do not need to use parentheses inside of lists. Now the list $[R, G, B]$ can be represented more clearly by `R ::: G ::: B ::: Nil`.

Type constraints

We can define recursive functions in the Haskell Type System using this representation of lists. However, in order to do so we must use a **type constraint**. For a recursively defined list function, we use a type constraint to say that the value for *x* ::: *xs* uses a recursive call on *xs*. In general, a type constraint is used to set preconditions on a function; the given function definition is only valid when the preconditions are met. By carefully constructing the constraints you can set up a recursion, or call out to other type-level functions.

For example, to concatenate two arbitrary lists we would write:

```
class ListConcat l1 l2 l | l1 l2 → l where
  listConcat :: l1 → l2 → l

instance ListConcat Nil l l where listConcat = ⊥
instance (ListConcat xs ys zs)
    ⇒ ListConcat (x ::: xs) ys (x ::: zs) where listConcat = ⊥
```

```
*Main> :type listConcat (u:: R ::: G ::: B ::: Nil) (u:: W ::: Nil)
```

```
listConcat (u::R:::G:::B:::Nil) (u::W:::Nil)::(:::) R ((:::) G ((:::) B ((:::) W Nil)))
```

Note that the type constraint, *ListConcat xs ys zs* justifies the recursive call.

Applyable type functions

For this puzzle, we need to be able to do things like *flip* each of the cubes in a list, so let's build towards something like *map*, at the type level. First we will need a way of abstracting the application of a type-level function, so we introduce a class *Apply*:

```
class Apply f a b | f a → b where
  apply :: f → a → b
```

The *Apply* class takes a function *f* of one argument, and an argument *a*, and returns the application of *f* to *a*.

Unfortunately, the functions *rot*, *twist*, and *flip* that we defined earlier are not declared at the type level; they cannot be passed to our type-level *Apply*. If we try to do so, we end up with:

```
*Main> :type apply rot (u::Cube1)
```

```
apply rot (u :: Cube1) :: forall u f r b l d b1 ∘
  (Transforms u f r b l d,
  Apply (Cube u f r b l d → Cube u r b l f d) Cube1 b1) ⇒ b1
```

Instead, we need to create types to name each of these operations:

```
data Rotation
data Twist
data Flip
```

We defined *Apply* as a parameterized interface. Let's instantiate it for each of our transform types:

```
instance Apply Rotation (Cube u f r b l d) (Cube u r b l f d)
  where apply = ⊥
instance Apply Twist (Cube u f r b l d) (Cube f r u l d b)
  where apply = ⊥
instance Apply Flip   (Cube u f r b l d) (Cube d l b r f u)
  where apply = ⊥
```

With all these pieces in place, we can now apply our *Rotation* function to an example cube:

```
*Main> :type apply (u::Rotation) (u::Cube1)
```

apply (*u* :: *Rotation*) (*u* :: *Cube1*) :: *Cube B W G B G R*

Map and Filter

We can now create a function that recurses over a list and *Apply*s another function *f* to each element. This is the type-level equivalent of fthe *map* function from the Haskell Prelude.

```
class Map f xs zs | f xs → zs where
  map :: f → xs → zs
```

If we call *Map* over the empty list, we get an empty list:

```
instance Map f Nil Nil where map = ⊥
```

In the *Cons* case we use two type constraints: one to call *Apply* on the head element, and one to call *Map* on the tail. The result will simply join these two values. As we recurse down the list *x* ::: *xs*, we build the result *z* ::: *zs*:

```
instance (Apply f x z, Map f xs zs)
  ⇒ Map f (x ::: xs) (z ::: zs) where map = ⊥
```

Once again, we show the *map* function in action, by mapping the *Flip* operation on a list of two cubes:

```
*Main> :type map (u::Flip) (u:: Cube1 ::: (Cube2 ::: Nil))
```

$$\begin{array}{l} \mathit{map}\ (u :: \mathit{Flip})\ (u :: \mathit{Cube1} ::: (\mathit{Cube2} ::: \mathit{Nil})) :: \\ \quad (:::)\ (\mathit{Cube}\ R\ B\ G\ W\ G\ B)\ ((:::)\ (\mathit{Cube}\ R\ R\ W\ B\ G\ W)\ \mathit{Nil}) \end{array}$$

We can build a *Filter* function similarly:

$$\begin{array}{l} \textbf{class}\ \mathit{Filter}\ f\ \mathit{xs}\ \mathit{zs} \mid f\ \mathit{xs} \rightarrow \mathit{zs}\ \textbf{where} \\ \quad \mathit{filter} :: f \rightarrow \mathit{xs} \rightarrow \mathit{zs} \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Filter}\ f\ \mathit{Nil}\ \mathit{Nil}\ \textbf{where}\ \mathit{filter} = \bot \\ \textbf{instance}\ (\mathit{Apply}\ f\ x\ b, \mathit{Filter}\ f\ \mathit{xs}\ \mathit{ys}, \mathit{AppendIf}\ b\ x\ \mathit{ys}\ \mathit{zs}) \\ \quad \Rightarrow \mathit{Filter}\ f\ (x ::: \mathit{xs})\ \mathit{zs}\ \textbf{where}\ \mathit{filter} = \bot \end{array}$$

Here we have introduced a third constraint, *AppendIf*, which takes a boolean value b, a value x, and a list ys. The given value x is appended to ys only if b is *True*, otherwise ys is returned unaltered:

$$\begin{array}{l} \textbf{class}\ \mathit{AppendIf}\ b\ x\ \mathit{ys}\ \mathit{zs} \mid b\ x\ \mathit{ys} \rightarrow \mathit{zs} \\ \textbf{instance}\ \mathit{AppendIf}\ \mathit{True}\ x\ \mathit{ys}\ (x ::: \mathit{ys}) \\ \textbf{instance}\ \mathit{AppendIf}\ \mathit{False}\ x\ \mathit{ys}\ \mathit{ys} \end{array}$$

Hence, *Filter* recurses down a list $x ::: xs$, and builds the list zs by appending only those values of x for which $f\ x$ is *True*.

List Comprehensions

Unfortunately we cannot directly mimic list comprehensions in the Haskell Type System, but we can translate the meaning of a given list comprehension using the type-level list functions that we have defined.

For example, building a list of the possible orientations of a cube involves appending a list of the possible applications of *flip*, so we will need to be able to map over a list and append the original list. The original list comprehension we are translating was:

```
orientations c
  = [c''' | ...,  c''' <- [c'', flip c'']]
```

We create a *MapAppend* class in order to compose *Map* and *ListConcat*:

$$\begin{array}{l} \textbf{class}\ \mathit{MapAppend}\ f\ \mathit{xs}\ \mathit{zs} \mid f\ \mathit{xs} \rightarrow \mathit{zs}\ \textbf{where} \\ \quad \mathit{mapAppend} :: f \rightarrow \mathit{xs} \rightarrow \mathit{zs} \end{array}$$

The *MapAppend* class has two instances:

$$\begin{array}{l}\textbf{instance}\ \mathit{MapAppend}\ f\ \mathit{Nil}\ \mathit{Nil}\ \textbf{where}\ \mathit{mapAppend} = \bot \\ \textbf{instance}\ (\mathit{Map}\ f\ \mathit{xs}\ \mathit{ys}, \mathit{ListConcat}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs}) \\ \quad \Rightarrow \mathit{MapAppend}\ f\ \mathit{xs}\ \mathit{zs} \qquad \textbf{where}\ \mathit{mapAppend} = \bot\end{array}$$

Further, we will need to be able to do the same twice for *twist*:

```
orientations c
  = [c''' | ...,  c'' <- [c', twist c', twist (twist c')], ...]
```

$$\begin{array}{l}\textbf{class}\ \mathit{MapAppend2}\ f\ \mathit{xs}\ \mathit{zs} \mid f\ \mathit{xs} \to \mathit{zs}\ \textbf{where} \\ \quad \mathit{mapAppend2} :: f \to \mathit{xs} \to \mathit{zs}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{MapAppend2}\ f\ \mathit{Nil}\ \mathit{Nil}\ \textbf{where}\ \mathit{mapAppend2} = \bot \\ \textbf{instance}\ (\mathit{Map}\ f\ \mathit{xs}\ \mathit{ys}, \mathit{MapAppend}\ f\ \mathit{ys}\ \mathit{ys}', \mathit{ListConcat}\ \mathit{xs}\ \mathit{ys}'\ \mathit{zs}) \\ \quad \Rightarrow \mathit{MapAppend2}\ f\ \mathit{xs}\ \mathit{zs} \qquad \textbf{where}\ \mathit{mapAppend2} = \bot\end{array}$$

and three times for *rot*:

```
orientations c
  = [c''' | c' <- [c, rot c, rot (rot c), rot (rot (rot c))], ...]
```

$$\begin{array}{l}\textbf{class}\ \mathit{MapAppend3}\ f\ \mathit{xs}\ \mathit{zs} \mid f\ \mathit{xs} \to \mathit{zs}\ \textbf{where} \\ \quad \mathit{mapAppend3} :: f \to \mathit{xs} \to \mathit{zs}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{MapAppend3}\ f\ \mathit{Nil}\ \mathit{Nil}\ \textbf{where}\ \mathit{mapAppend3} = \bot \\ \textbf{instance}\ (\mathit{Map}\ f\ \mathit{xs}\ \mathit{ys}, \mathit{MapAppend2}\ f\ \mathit{ys}\ \mathit{ys}', \mathit{ListConcat}\ \mathit{xs}\ \mathit{ys}'\ \mathit{zs}) \\ \quad \Rightarrow \mathit{MapAppend3}\ f\ \mathit{xs}\ \mathit{zs} \qquad \textbf{where}\ \mathit{mapAppend3} = \bot\end{array}$$

Orientations

The full list comprehension for generating all possible orientations of a cube builds upon all combinations of *rot*, *twist* and *flip*:

```
orientations c =
  [c''' | c' <- [c, rot c, rot (rot c), rot (rot (rot c))],
          c'' <- [c', twist c', twist (twist c')],
          c''' <- [c'', flip c'']]
```

We will implement Orientations as an *Apply*able type function. In turn, it is defined in terms of applications of *Rotation*, *Twist* and *Flip*, invoked via the various *MapAppend* functions:

```
data Orientations
instance (MapAppend Flip (c ::: Nil) fs, MapAppend2 Twist fs ts,
  MapAppend3 Rotation ts zs)
   ⇒ Apply Orientations c zs
  where apply = ⊥
```

For any *Cube*, this function generates the 24 possible orientations:

```
*Main> :type apply (u::Orientations) (u::Cube1)
```

```
apply (u :: Orientations) (u :: Cube1) ::
  (:::) (Cube B G W G B R) ((:::) (Cube R B G W G B)
((:::) (Cube G W B B R G) ((:::) (Cube B G R G B W)
((:::) (Cube W B G R G B) ((:::) (Cube G R B B W G)
((:::) (Cube B W G B G R) ((:::) (Cube R G W G B B)
((:::) (Cube G B B R W G) ((:::) (Cube B R G B G W)
((:::) (Cube W G R G B B) ((:::) (Cube G B B W R G)
((:::) (Cube B G B G W R) ((:::) (Cube R W G B G B)
((:::) (Cube G B R W B G) ((:::) (Cube B G B G R W)
((:::) (Cube W R G B G B) ((:::) (Cube G B W R B G)
((:::) (Cube B B G W G R) ((:::) (Cube R G B G W B)
((:::) (Cube G R W B B G) ((:::) (Cube B B G R G W)
((:::) (Cube W G B G R B) ((:::) (Cube G W R B B G)
Nil)))))))))))))))))))))))
```

Stacking Cubes

Given two cubes (*Cube u1 f1 r1 b1 l1 d1*) (*Cube u2 f2 r2 b2 l2 d2*), we now want to check that none of the corresponding visible faces are the same colour: the front sides *f1* and *f2* are not equal, and the right sides *r1* and *r2* are not equal, and so on. In short, we want to determine whether:

$$(f1 \neq f2) \wedge (r1 \neq r2) \wedge (b1 \neq b2) \wedge (l1 \neq l2)$$

In order to do this, we will first need to define the $\neq$ relation for all four colours. Given two cubes, we can then apply this relations to each pair of visible faces to get four boolean values. To check that all of these are True, we will construct a list of those four values, and then write a generic list function to check if all elements of a list are True.

Not Equal

To define the $\neq$ relation, we introduce a new class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{NE}\ x\ y\ b \mid x\ y \rightarrow b\ \textbf{where} \\
\quad ne :: x \rightarrow y \rightarrow b
\end{array}
$$

We are going to use *NE* to instantiate type comparisons for the faces of our cube. Recall that these faces are of the atomic types R, G, B, W, and we have not yet defined any relation between these atomic types.

$$
\begin{array}{ll}
\textbf{instance}\ \mathit{NE}\ R\ R\ \mathit{False} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ R\ G\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ R\ B\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ R\ W\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ G\ R\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ G\ G\ \mathit{False} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ G\ B\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ G\ W\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ B\ R\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ B\ G\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ B\ B\ \mathit{False} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ B\ W\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ W\ R\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ W\ G\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ W\ B\ \mathit{True} & \textbf{where}\ ne = \bot \\
\textbf{instance}\ \mathit{NE}\ W\ W\ \mathit{False} & \textbf{where}\ ne = \bot
\end{array}
$$

Note that our class *NE* is very different from the class *Eq* defined in Haskell:

```
*Main> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
        -- Imported from GHC.Base
...
```

Whereas *Eq* is used to describe types for which it is possible to compare **instances** for equality, *NE* directly compares types.

All

Now, we define a function *all* to check if all elements of a list are True.

$$
\begin{array}{l}
\textbf{class } \mathit{All}\ l\ b \mid l \to b \textbf{ where} \\
\quad \mathit{all} :: l \to b \\
\textbf{instance } \mathit{All}\ \mathit{Nil}\ \mathit{True} \textbf{ where } \mathit{all} = \bot \\
\textbf{instance } \mathit{All}\ (\mathit{False} ::: \mathit{xs})\ \mathit{False} \qquad\qquad \textbf{where } \mathit{all} = \bot \\
\textbf{instance } (\mathit{All}\ \mathit{xs}\ b) \Rightarrow \mathit{All}\ (\mathit{True} ::: \mathit{xs})\ b \textbf{ where } \mathit{all} = \bot
\end{array}
$$

The constraint $(\mathit{All}\ xs\ b) \Rightarrow \mathit{All}\ (\mathit{True}:::xs)\ b$ says that the output b of $\mathit{All}\ (\mathit{True}::: xs)\ b$ has the same value as the output b of $\mathit{All}\ xs\ b$; ie. that if the head of the list is *True*, then the value of *All* is determined by the rest of the list.

Once again, we can check that *all* behaves as we expect:

```
*Main> :type all (u::Nil)
```

$\mathit{all}\ (u :: \mathit{Nil}) :: \mathit{True}$

```
*Main> :type all (u:: False ::: Nil)
```

$\mathit{all}\ (u :: \mathit{False} ::: \mathit{Nil}) :: \mathit{False}$

```
*Main> :type all (u:: True ::: False ::: Nil)
```

$\mathit{all}\ (u :: \mathit{True} ::: \mathit{False} ::: \mathit{Nil}) :: \mathit{False}$

```
*Main> :type all (u:: True ::: True ::: True ::: Nil)
```

$\mathit{all}\ (u :: \mathit{True} ::: \mathit{True} ::: \mathit{True} ::: \mathit{Nil}) :: \mathit{True}$

Compatible

We can now write the compatibility check in the the Haskell Type System, that corresponds to the original `compatible` function:

```
visible [u, f, r, b, l, d] = [f, r, b, l]

compatible c c' =
  and [x /= x' | (x, x') <- zip (visible c) (visible c')]
```

We introduce a new *Compatible* class. It should check that no corresponding visible faces are the same colour.

$$\begin{array}{l}\textbf{class}\ \mathit{Compatible}\ c1\ c2\ b \mid c1\ c2 \to b\ \textbf{where}\\ \quad \mathit{compatible} :: c1 \to c2 \to b\end{array}$$

We will do this by evaluating the relationship *NE* for each pair of corresponding visible faces, giving four booleans *bF*, *bR*, *bB*, and *bL*. Whether or not the two *Cube*s are compatible is then determined by *All* (*bF* ::: *bR* ::: *bB* ::: *bL* ::: *Nil*).

$$\begin{array}{l}\textbf{instance}\ (\mathit{NE}\ f1\ f2\ bF, \mathit{NE}\ r1\ r2\ bR, \mathit{NE}\ b1\ b2\ bB, \mathit{NE}\ l1\ l2\ bL,\\ \quad \mathit{All}\ (bF ::: bR ::: bB ::: bL ::: \mathit{Nil})\ b)\\ \quad \Rightarrow \mathit{Compatible}\ (\mathit{Cube}\ u1\ f1\ r1\ b1\ l1\ d1)\ (\mathit{Cube}\ u2\ f2\ r2\ b2\ l2\ d2)\ b\\ \quad \textbf{where}\ \mathit{compatible} = \bot\end{array}$$

A completely red cube is obviously compatible with a completely blue cube:

```
*Main> :type compatible (u::Cube R R R R R R) (u::Cube B B B B B B)
```

compatible (*u* :: *Cube R R R R R R*) (*u* :: *Cube B B B B B B*) :: *True*

whereas if we paint their front sides green then they are no longer compatible:

```
*Main> :type compatible (u::Cube R R G R R R) (u::Cube B B G B B B)
```

compatible (*u* :: *Cube R R G R R R*) (*u* :: *Cube B B G B B B*) :: *False*

Allowed

The above *Compatible* class checks a cube for compatibility with another single cube. In the puzzle, a cube needs to be compatible with all the other cubes in the pile.

```
allowed c cs = and [compatible c c' | c' <- cs]
```

We write a class to check for compatibility with each of a list of cubes. This class generalizes *Compatible* over lists:

$$\begin{array}{l}\textbf{class}\ \mathit{Allowed}\ c\ cs\ b \mid c\ cs \to b\ \textbf{where}\\ \quad \mathit{allowed} :: c \to cs \to b\\ \textbf{instance}\ \mathit{Allowed}\ c\ \mathit{Nil}\ \mathit{True}\ \textbf{where}\ \mathit{allowed} = \bot\\ \textbf{instance}\ (\mathit{Compatible}\ c\ y\ b1, \mathit{Allowed}\ c\ ys\ b2, \mathit{And}\ b1\ b2\ b)\\ \quad \Rightarrow \mathit{Allowed}\ c\ (y ::: ys)\ b\ \textbf{where}\ \mathit{allowed} = \bot\end{array}$$

Sure enough, we can now test the *allowed* function:

```
*Main> :type allowed (u::CubeRed) (u:: CubeBlue ::: CubeRed ::: Nil)
```

allowed (*u* :: *CubeRed*) (*u* :: *CubeBlue* ::: *CubeRed* ::: *Nil*) :: *False*

Solution

We are now ready to tackle the implementation of *solutions*:

```
solutions [] = [[]]
solutions (c:cs) = [c' : cs' | cs' <- solutions cs,
                                c' <- orientations c,
                                allowed c' cs']
```

We will create a corresponding class *Solutions*, which takes a list of *Cube*s as input, and returns a list of possible solutions, where each solution is a list of *Cube*s in allowed orientations.

```
class Solutions cs ss | cs → ss where
  solutions :: cs → ss
```

The base case for *Solutions* is the list containing the empty list, [[]], which we represent by *Nil* ::: *Nil*. The recursive step considers all orientations of the topmost *Cube*:

```
instance Solutions Nil (Nil ::: Nil)
  where solutions = ⊥
instance (Solutions cs sols, Apply Orientations c os,
  AllowedCombinations os sols zs)
  ⇒ Solutions (c ::: cs) zs
  where solutions = ⊥
```

The *AllowedCombinations* class recurses across the solutions so far, checking each against the given orientations.

```
class AllowedCombinations os sols zs | os sols → zs
instance AllowedCombinations os Nil Nil
instance (AllowedCombinations os sols as, MatchingOrientations os s bs,
  ListConcat as bs zs)
  ⇒ AllowedCombinations os (s ::: sols) zs
```

Finally, the *MatchingOrientations* class recurses across the orientations of the new cube, checking each against a particular solution *sol*.

```
class MatchingOrientations os sol zs | os sol → zs
instance MatchingOrientations Nil sol Nil
instance (MatchingOrientations os sol as,
  Allowed o sol b, AppendIf b (o ::: sol) as zs)
  ⇒ MatchingOrientations (o ::: os) sol zs
```

If the orientation is allowed, then the combination o is added to the existing solutions sol, by forming the type $o ::: sol$.

Note that we have not been able to make use of the previously defined *Filter* because it requires a one-argument *Apply*able function. As we lack currying, we are unable to construct such a function on the fly. However, we can make use of the more generic *AppendIf* (defined on page 32) to handle the filtering constraint.

Finally, we can solve the puzzle for the given cubes:

```
type Cubes = (Cube1 ::: Cube2 ::: Cube3 ::: Cube4 ::: Nil)
```

```
*Main> :type solutions (u::Cubes)
```

```
solutions (u :: Cubes) ::
(:::) ((:::) (Cube G B B R W G) ((:::) (Cube R G R W B W)
        ((:::) (Cube R W G B R R) ((:::) (Cube W R W G G B) Nil))))
((:::) ((:::) (Cube G B B W R G) ((:::) (Cube W R G B W R)
        ((:::) (Cube R G W R B R) ((:::) (Cube B W R G G W) Nil))))
((:::) ((:::) (Cube G W B B R G) ((:::) (Cube R B G R W W)
        ((:::) (Cube R R W G B R) ((:::) (Cube W G R W G B) Nil))))
((:::) ((:::) (Cube G R B B W G) ((:::) (Cube W W R G B R)
        ((:::) (Cube R B G W R R) ((:::) (Cube B G W R G W) Nil))))
((:::) ((:::) (Cube G R W B B G) ((:::) (Cube R W B G R W)
        ((:::) (Cube R B R W G R) ((:::) (Cube W G G R W B) Nil))))
((:::) ((:::) (Cube G W R B B G) ((:::) (Cube W B W R G R)
        ((:::) (Cube R R B G W R) ((:::) (Cube B G G W R W) Nil))))
((:::) ((:::) (Cube G B R W B G) ((:::) (Cube R R W B G W)
        ((:::) (Cube R G B R W R) ((:::) (Cube W W G G R B) Nil))))
((:::) ((:::) (Cube G B W R B G) ((:::) (Cube W G B W R R)
        ((:::) (Cube R W R B G R) ((:::) (Cube B R G G W W) Nil))))
Nil)))))))
```

Changing the order of backtracking steps changes the order of solutions. For comparison, here is the solution generated by the pure Haskell version:

```
[["GBWRBG","WGBWRR","RWRBGR","BRGGWW"],
 ["GBRWBG","RRWBGW","RGBRWR","WWGGRB"],
 ["GWRBBG","WBWRGR","RRBGWR","BGGWRW"],
 ["GBBRWG","RGRWBW","RWGBRR","WRWGGB"],
 ["GRBBWG","WWRGBR","RBGWRR","BGWRGW"],
 ["GWBBRG","RBGRWW","RRWGBR","WGRWGB"],
 ["GBBWRG","WRGBWR","RGWRBR","BWRGGW"],
 ["GRWBBG","RWBGRW","RBRWGR","WGGRWB"]]
```

Using types in Haskell

Now that we've seen how to use the Haskell Type System to solve a puzzle, let's review what it was actually designed for: ensuring that your programs make sense. The syntax of the Haskell Type System lets you tell the compiler what your program must or must not do, while also giving you enough flexibility to implement useful shared and extensible interfaces.

Let's review the features of the Haskell Type System from a Haskell programming perspective.

Keywords

The most straightforward way to define your own types is with the keyword **data**. You can use this to define simple types or to introduce structured records.

Type aliases are a handy form of abbreviation; for example the Prelude provides **type** *Rational* = *Ratio Integer*, to make code that manipulates rational numbers easier to read and write; under the hood, code that works with *Rational*s is actually dealing with the *Ratio Integer* type. This is different from **newtype** which declares an entirely new type that cannot be used in the same contexts.

Parameterized type-classes, introduced by the keyword **class**, are used to provide common interfaces for existing types. These interfaces are composed of **methods**, which may be functions or constant values. An **instance** of a type-class simply provides an implementation of the class methods. You can also provide default methods in a **class** declaration, which are used unless overridden in an **instance**.

Haskell98 features

Parameterized types are used to wrap types in general interfaces. You use this technique when you want to ensure that you can't accidentally use two variants of a parameterized type in the same call; the Haskell Type System ensures that they are incompatible. You can also use this to create tainted versions of existing types, in order to quarantine data that has come from untrusted sources.

Type constraints, introduced by ⇒, are often used to provide sensible preconditions. For example, the Haskell Prelude declares the type-class *Real* with the constraint **class** (*Num a*, *Ord a*) ⇒ *Real a*, which expresses that you can only try to implement the interface for *Real* for types which are numeric and have an ordering; the Haskell Type System first ensures that that you have already implemented *Num* and *Ord*.

When reading Haskell code, you will notice that type constraints can occur in data and function definitions, not just in class instances.

Type constraints can also be used more creatively to encode post-conditions, if these conditions are encoded into the function types. For example, Kyselyov's Dependently Typed Append [2], implements a list append with the assurance that the length of the output list is the sum of the lengths of the input list.

Extensions

Some of the language features used in this tutorial are not part of the Haskell98 specification, but are widely supported by Haskell compilers. When building with GHC, these extensions can be enabled with the options `-fglasgow-exts` [3].

The simplest of these extensions is the use of **data** types with no constructors, such as

> **data** *Red*
> **data** *Blue*

As these types can only contain the value ⊥, they are not much use on their own. However, they can be used to construct other, more complex types, and can be used as **phantom types** [4].

Multi-parameter type-classes allow you to specify an interface which can behave differently for each combination of "input" types. This is usually used together with **functional dependencies** [5], which make it easier for the type-checker to infer type relationships, and make the code easier to read. We enable the GHC option `-fundecidable-instances` [3] to allow general recursion at the type-level.

GHC allows empty class declarations (with a warning), so the definitions in this tutorial which specify **where** *something* = ⊥ are actually unnecessary. However, we left them in in order to retain some semblance of connection to conventional Haskell programming.

Conclusion

We have seen how to use the Haskell Type System as a programming language to solve a given problem. Of course, its real purpose is not to solve problems directly, but to express constraints on the behaviour of a system in Haskell. The type level features we have seen allow us to express the pre- and post-conditions of functions. By expressing these constraints in the type system, the compiler statically verifies that the produced code operates as required, without the need for run-time checks.

We reiterate that familiarity with the syntax of the Haskell Type System is a prerequisite for general Haskell programming. With a solid understanding of type and type-class parameterization, type constraints and dependencies, you should be well on your way to understanding the interfaces of interesting and useful types.

This tutorial introduced the most widely used type-level features in general Haskell programming, extending Haskell98 with multi-parameter type-classes and undecidable instances. If you wish to go further with type-level programming there are many interesting extensions to the Haskell Type System [3], and more advanced type systems research explores topics like program verification and proof carrying code. [6, 7].

Please enjoy hacking in the Haskell Type System, and wield it wisely in your program designs.

Acknowledgements

Thanks to Simon Horman, Patryk Zadarnowski, Shane Stephens and Wouter Swierstra for their feedback on drafts of this article.

About the author

Conrad Parker (kfish on #haskell) is a PhD student at Kyoto University. Originally from Sydney, he graduated from UNSW with degrees in Mathematics and Computer Engineering. Hobbies include computer music and Japanese language.

References

[1] Richard Bird and Philip Wadler. **Introduction to Functional Programming**. Prentice Hall (1998).

[2] Oleg Kiselyov. Dependently typed append. `http://okmij.org/ftp/Haskell/dependently-typed-append.lhs`.

[3] Ghc User's Guide: Type system extensions. `http://www.haskell.org/ghc/docs/latest/html/users_guide/type-extensions.html`.

[4] HaskellWiki: Phantom type. `http://haskell.org/haskellwiki/Phantom_type`.

[5] Thomas Hallgren. Fun with functional dependencies (2001). `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

[6] HaskellWiki: Type systems. `http://haskell.org/haskellwiki/Research_papers/Type_systems`.

[7] Oleg Kiselyov. Haskell programming: Attractive types. `http://okmij.org/ftp/Haskell/types.html`.