J. Functional Programming **8** (1): 83–88, January 1998. Printed in the United Kingdom

FUNCTIONAL PEARL

*Meertens number**

RICHARD S. BIRD
Programming Research Group, Oxford University,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

1 Introduction

Meertens number is a number with a very peculiar property. I had the idea in 1991, when I was invited to celebrate the occasion of Lambert Meertens' 25 years at the CWI, Amsterdam. Lambert has been a good friend and colleague for a number of years, and rather than bring the usual bottle of Glenlivet as a gift, I decided to give him a number.

The property concerns the prime factorisation of numbers. For example, $400 = 2^4 3^0 5^2$ and $432 = 2^4 3^3 5^0$. Suppose we define $g(n)$ by taking the decimal representation $d_1 d_2 \ldots d_k$ of n and setting $g(n) = 2^{d_1} 3^{d_2} \ldots p_k^{d_k}$, where p_k is the kth prime. Thus, $g(402) = 400$ and $g(430) = 432$. The numbers n and $g(n)$ are not always so close together as these two examples suggest; in fact, most of the time they are wildly different. The question is: do they ever coincide?

The function g is named after Kurt Gödel, who exploited the idea of coding a sequence of characters as a single number in his famous paper on the incompleteness of arithmetic. Here we are using the same device, coding a number by coding its decimal representation. If $n = g(n)$, then n is called a *Meertens* number.

The existence of a Meertens number is not at all obvious. The number has to be even, and a little mental calculation shows that it cannot have one, two or three digits. For example, any three digit number would have to end with a zero and so be both divisible and not divisible by five. On the other hand, there are some near misses as the numbers 402 and 430 show. Furthermore, as n increases the value $g(n)$ jumps about quite a lot, so there is no obvious reason why they should never coincide.

The problem is not specific to decimals of course. In base 2, each of the numbers 2, 6 and 10 is a Meertens number. For example, 6 in binary is 110 and $6 = 2^1 3^1 5^0$. The number 10 is also a Meertens number in base 3 since 10 in ternary is 101 and $10 = 2^1 3^0 5^1$. Such results are encouraging, but finding the Meertens number for mere bits and bytes is not enough. Mere tens, on the other hand, would be quite

* This pearl is adapted from material in *Introduction to Functional Programming, Second Edition* by Richard Bird, Prentice Hall, to be published in 1998. Permission to publish is gratefully acknowledged.

appropriate. Our aim in this pearl is to construct a program for finding Meertens numbers.

2 A simple approach

The most obvious method is to define

$$\begin{array}{lcl} \mathit{meertens} & :: & [\mathit{Integer}] \\ \mathit{meertens} & = & [n \mid n \leftarrow [1\,..\,], n = \mathit{godel}\ n] \\ \\ \mathit{godel} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{godel}\ n & = & \mathit{product}\ (\mathit{zipWith}\ (\uparrow)\ \mathit{primes}\ (\mathit{digits}\ n)) \end{array}$$

The function *digits* returns the decimal representation of a number and $(\uparrow)$ denotes exponentiation. The Haskell type *Integer* contains integers of arbitrary precision. The list *primes* of all primes can be defined in a number of ways, and we will not go into details.

The trouble with this program is that it is very slow. The godel number of each candidate is computed afresh and this takes a significant number of steps.

As preparation for a change in representation, we can rewrite the definition of *meertens* in the eqivalent form

$$\begin{array}{lcl} \mathit{meertens} & = & [n \mid (n, g) \leftarrow \mathit{map}\ \mathit{gn}\ [1\,..\,], n = g] \\ & & \mathbf{where}\ \mathit{gn}\ n = (n, \mathit{godel}\ n) \end{array}$$

3 Using trees

One way to avoid excessive recomputation is to use a tree structure for storing partial results. Each node in the tree, apart from the very first, has 10 subtrees, one for each digit. The first tree has 9 subtrees, since decimals begin with a nonzero digit. Each node in the tree is labelled with two integers, n and g. The decimal representation of n describes the path in the tree to the given node and $g = \mathit{godel}\ n$. The tree therefore has type *Tree*, where

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Tree} & = & \mathit{Node}\ \mathit{Label}\ [\mathit{Tree}] \\ \mathbf{type}\ \mathit{Label} & = & (\mathit{Integer}, \mathit{Integer}) \end{array}$$

The idea is to build a tree *gtree* so that

$$\mathit{flatten}\ \mathit{gtree} \quad = \quad \mathit{map}\ \mathit{gn}\ [1\,..\,]$$

where *flatten* lists the labels of a tree in breadth-first order. The value *gtree* is defined by

$$\begin{array}{lcl} \mathit{gtree} & :: & \mathit{Tree} \\ \mathit{gtree} & = & \mathit{snip}\ (\mathit{mktree}\ \mathit{prps}\ (0, 1)) \end{array}$$

The function *snip* snips off the leftmost branch to remove decimals beginning with zero:

$$\begin{array}{lcl} \mathit{snip} & :: & \mathit{Tree} \rightarrow \mathit{Tree} \\ \mathit{snip}\ (\mathit{Node}\ x\ \mathit{ts}) & = & \mathit{Node}\ x\ (\mathit{tail}\ \mathit{ts}) \end{array}$$

The work of building the tree is relegated to the function *mktree*. This function takes an infinite list *prps* of the powers of each prime and a starting label of $(0, 1)$, which is correct since $g(0) = 1$. The list *prps* is defined by

$$\begin{aligned} prps &:: [[Integer]] \\ prps &= map\ powers\ primes \end{aligned}$$

The value of *powers p* is an infinite list $[1, p, p^2, \ldots]$ of powers of p:

$$\begin{aligned} powers &:: Integer \to [Integer] \\ powers\ p &= iterate\ (p\times)\ 1 \end{aligned}$$

The standard function *iterate* is defined by

$$\begin{aligned} iterate &:: (\alpha \to \alpha) \to \alpha \to [\alpha] \\ iterate\ f\ x &= x : iterate\ f\ (f\ x) \end{aligned}$$

Now we can define *mktree* by

$$\begin{aligned} mktree &:: [[Integer]] \to Label \to Tree \\ mktree\ (ps : pps)\ x &= Node\ x\ (map\ (mktree\ pps)\ (labels\ ps\ x)) \end{aligned}$$

where

$$\begin{aligned} labels &:: [Integer] \to Label \to [Label] \\ labels\ ps\ (n, g) &= zip\ (map\ (m+)\ digits)\ (map\ (g\times)\ ps) \\ &\quad \textbf{where}\ m = 10 \times n;\ \ digits = [0\ ..\ 9] \end{aligned}$$

The list *meertens* of Meertens' numbers can now be computed, in theory at least, by

$$\begin{aligned} meertens &= [n \mid (n, g) \leftarrow flatten\ gtree, n = g] \\ &\quad \textbf{where}\ gn\ n = (n, godel\ n) \end{aligned}$$

There is no theoretical difficulty in printing out the labels of an infinite tree in breadth-first order because the ordering proceeds level by level. However, there is a substantial practical difficulty: the computation rapidly runs out of space and grinds to a halt. Searching a tree in breadth-first order requires space proportional to the size of the tree, and the size of Meertens' tree grows exponentially. No evaluator, real or imagined, has sufficient resources to proceed very far with the computation.

The obvious solution is to restrict the height of the tree and to switch to depth-first search. We have to make the tree finite if we want to use depth-first search, because one cannot traverse an infinite Meertens' tree in depth-first order. If we build a tree of height k, then we can search for Meertens' numbers n in the range $1 \leq n < 10^k$. The revised definition is

$$\begin{aligned} meertens &:: Int \to [Integer] \\ meertens\ k &= [n \mid (n, g) \leftarrow flatten\ (gtree\ k), n = g] \\ &\quad \textbf{where}\ gn\ n = (n, godel\ n) \end{aligned}$$

This time, *flatten* lists the labels of a tree in depth-first order:

$$flatten\ (Node\ x\ ts) = x : concat\ (map\ flatten\ ts)$$

The function *gtree* returns trees of given height:

$$\begin{aligned} gtree &:: Int \rightarrow Tree \\ gtree\ k &= snip\ (mktree\ prps\ (0,1)) \\ &\quad \textbf{where}\ prps = map\ powers\ (take\ k\ primes) \end{aligned}$$

The function *mktree* is virtually the same as before:

$$\begin{aligned} mktree &:: [[Integer]] \rightarrow Label \rightarrow Tree \\ mktree\ [\,]\ x &= Node\ x\ [\,] \\ mktree\ (ps : pps)\ x &= Node\ x\ (map\ (mktree\ pps)\ (labels\ ps\ x)) \end{aligned}$$

Since we have replaced breadth-first by depth-first search, it follows that *meertens* is no longer guaranteed to produce numbers in increasing order.

4 Optimisation

The revised definition is faster but still very slow. There is another optimisation that can usefully be applied. There is no point in generating any subtree with a label (n, g) in which $g \geq 10^k$ since, by construction, all labels (n, g) in the tree have $n < 10^k$. To incorporate this test without carrying around an extra parameter, we will make the functions *mktree* and *labels* local to the definition of *gtree*:

$$\begin{aligned} & gtree\ k \\ & \quad = snip\ (mktree\ prps\ (0,1)) \\ & \qquad \textbf{where} \\ & \qquad prps = map\ powers\ (take\ k\ primes) \\ & \qquad mktree\ [\,]\ x = Node\ x\ [\,] \\ & \qquad mktree\ (ps : pps)\ x = Node\ x\ (map\ (mktree\ pps)\ (labels\ ps\ x)) \\ & \qquad labels\ ps\ (n,g) = zip\ (map\ (m+)\ ds)\ (chop\ (map\ (g\times)\ ps)) \\ & \qquad\qquad \textbf{where}\ m = 10 \times n \\ & \qquad chop = takeWhile (< 10^k) \end{aligned}$$

Even with these optimisations the search is still fairly slow. Apart from carefully studying the problem to see what number theory can be applied, there is one more technique for improving matters and we consider this next.

5 Deforestation

Deforestation is the general name for a programming technique that removes intermediate datatypes from a program. The idea is to get rid of the tree we have so carefully constructed. The saving will not be huge but it will be significant. If the resulting program can be sped up by, say, a factor of two, then the optimisation is worthwhile. (Surely by now the reader will have suspected that the first Meertens number is not a small one.)

Like all sound techniques for optimisation, deforestation is carried out simply by program calculation. The theme of the calculation is to combine functions that produce and consume elements of an intermediate datatype. We begin by calculating, for $k > 0$, that

$$\begin{array}{ll}
 & (\mathit{flatten} \cdot \mathit{gtree})\ k \\
= & \quad \{\text{definition of } \mathit{gtree}\text{, with } \mathit{prps} = \mathit{map\ powers}\ (\mathit{take}\ k\ \mathit{primes})\} \\
 & (\mathit{flatten} \cdot \mathit{snip} \cdot \mathit{mktree\ prps})\ (0,1) \\
= & \quad \{\text{definition of } \mathit{mktree}\text{, with } \mathit{ps} : \mathit{pps} = \mathit{prps}\ (\text{since } k > 0)\} \\
 & (\mathit{flatten} \cdot \mathit{snip} \cdot \mathit{Node}\ (0,1) \cdot \mathit{map}\ (\mathit{mktree\ pps}) \cdot \mathit{labels\ ps})\ (0,1) \\
= & \quad \{\text{since } \mathit{snip} \cdot \mathit{Node}\ x = \mathit{Node}\ x \cdot \mathit{tail}\} \\
 & (\mathit{flatten} \cdot \mathit{Node}\ (0,1) \cdot \mathit{tail} \cdot \mathit{map}\ (\mathit{mktree\ pps}) \cdot \mathit{labels\ ps})\ (0,1) \\
= & \quad \{\text{since } \mathit{tail} \cdot \mathit{map}\ f = \mathit{map}\ f \cdot \mathit{tail}\} \\
 & (\mathit{flatten} \cdot \mathit{Node}\ (0,1) \cdot \mathit{map}\ (\mathit{mktree\ pps}) \cdot \mathit{tail} \cdot \mathit{labels\ ps})\ (0,1) \\
= & \quad \{\text{since } \mathit{flatten} \cdot \mathit{Node}\ x = (x\ :) \cdot \mathit{concat} \cdot \mathit{map\ flatten}\} \\
 & (((0,1)\ :) \cdot \mathit{concat} \cdot \mathit{map\ flatten} \cdot \mathit{map}\ (\mathit{mktree\ pps}) \cdot \mathit{tail} \cdot \mathit{labels\ ps})\ (0,1) \\
= & \quad \{\text{since } \mathit{map} \text{ is a functor}\} \\
 & (((0,1)\ :) \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{flatten} \cdot \mathit{mktree\ pps}) \cdot \mathit{tail} \cdot \mathit{labels\ ps})\ (0,1)
\end{array}$$

The resulting expression suggests introducing the function *search*, defined by

$$\mathit{search\ pps} \quad = \quad \mathit{flatten} \cdot \mathit{mktree\ pps}$$

It is this function that captures the idea of deforestation. The building and flattening of a tree is combined in a single function.

The aim now is to get a recursive definition of *search*. The simplification of *search* [ ] is straightforward, and that of *search* (*ps* : *pps*) *x* is similar to that of *flatten* · *gtree*. The result is

$$\begin{array}{lcl}
\mathit{search}\ [\,]\ x & = & [x] \\
\mathit{search}\ (\mathit{ps} : \mathit{pps})\ x & = & x : \mathit{concat}\ (\mathit{map}\ (\mathit{search\ ps})\ (\mathit{labels\ ps}\ x))
\end{array}$$

Here is the final program, in which we suppose $k > 0$:

$$\begin{array}{l}
\mathit{meertens}\ k = [n \mid (n,g) \leftarrow \mathit{candidates}\ (0,1), n = g] \\
\quad \begin{array}{lcl}
\textbf{where} & & \\
\mathit{candidates} & = & \mathit{concat} \cdot \mathit{map}\ (\mathit{search\ pps}) \cdot \mathit{tail} \cdot \mathit{labels\ ps} \\
\mathit{ps} : \mathit{pps} & = & \mathit{map\ powers}\ (\mathit{take}\ k\ \mathit{primes}) \\
\mathit{search}\ [\,]\ x & = & [x] \\
\mathit{search}\ (\mathit{ps} : \mathit{pps})\ x & = & x : \mathit{concat}\ (\mathit{map}\ (\mathit{search\ pps})\ (\mathit{labels\ ps}\ x)) \\
\mathit{labels\ ps}\ (n,g) & = & \mathit{zip}\ (\mathit{map}\ (m+)\ \mathit{digits})\ (\mathit{chop}\ (\mathit{map}\ (g\times)\ \mathit{ps})) \\
 & & \textbf{where}\ m = 10 \times n \\
\mathit{chop} & = & \mathit{takeWhile}\ (< 10^k) \\
\mathit{digits} & = & [0\,..\,9]
\end{array}
\end{array}$$

6 A result

The program above was run under Hugs for Windows 3.1 on a small and ancient laptop with 22K of heap space:

```
? meertens 8
[81312000]
(1783302 reductions, 4393345 cells, 189 garbage collections)
```

As far as I am aware, 81312000 is the only known Meertens number. As I said to Lambert, handle it carefully and keep it cool, for it decomposes rather easily.

Acknowledgement

I would like to thank Geraint Jones for a number of discussions about Meertens number, and my wife Norma for lending me her laptop.