MANUFACTURING DATATYPES

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Hompage: http://www.informatik.uni-bonn.de/~ralf

September, 1999

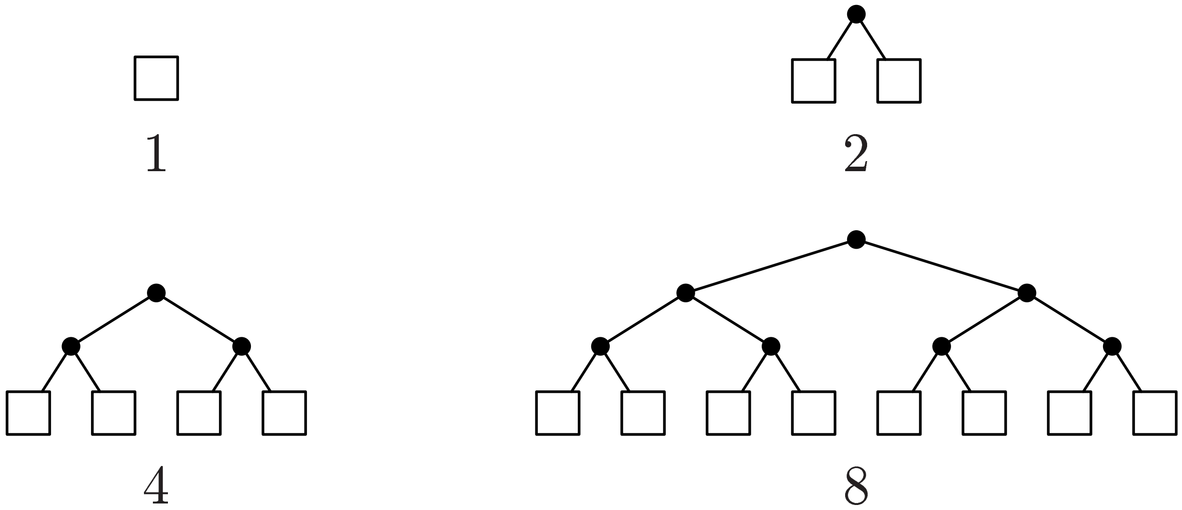
Summary

We describe a general framework for designing purely functional datatypes that automatically satisfy given size or structural constraints.

☞ The work is inspired by Chris Okasaki's recent paper "From Fast Exponentiation to Square Matrices: An Adventure in Types" presented at ICFP'99.

Problem #1

How can we represent perfect leaf trees in Haskell (or in Standard ML)?

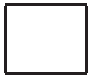


A non-solution:

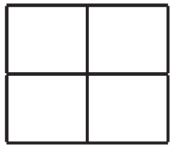
$$\boxed{\quad \mathbf{data}\ \mathit{Perfect}\ a \quad = \quad \mathit{Leaf}\ \ a \mid \mathit{Fork}\ (\mathit{Perfect}\ a)\ (\mathit{Perfect}\ a) \qquad \lightning\,. }$$

Problem #2

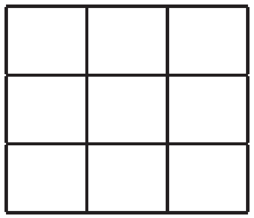
How can we represent square matrices?



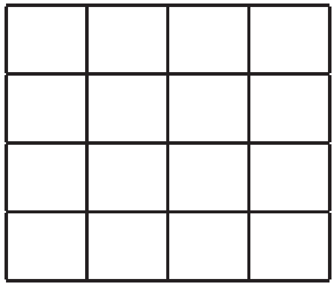
1



4



9



16

A non-solution:

$$\textbf{type}\ \mathit{Square}\ a \;=\; [[a]] \qquad \lightning\,.$$

Sketch of solution

First, solve a related, but simpler problem: generate the multiset of all natural numbers that are powers of 2 (for problem $\#1$).

$$power \quad = \quad \wr 1 \wr \uplus power * (\wr 1 \wr + \wr 1 \wr)$$

Second, transform this recursion equation into a functor equation.

$$Power \quad = \quad Id \mid Power \cdot (Id \times Id)$$

Third, rewrite the functor equation into a Haskell datatype declaration.

$$\mathbf{data}\ Perfect\ a \quad = \quad Zero\ a \mid Succ\ (Perfect\ (a, a))$$

☞ Step 2 and step 3 are entirely mechanical (except for the invention of nice constructor names).

Multisets

Multiset expressions are formed according to the following grammar.

$$M \quad ::= \quad \varnothing \mid \wr 0 \wr \mid \wr 1 \wr \mid (M \uplus M) \mid (M + M) \mid (M * M)$$

Example. $\wr 2 \wr$ is shorthand for $\wr 1 \wr + \wr 1 \wr$.

$$\begin{aligned} nat &= \wr 0 \wr \uplus \wr 1 \wr + nat \\ odd &= \wr 1 \wr + nat * \wr 2 \wr \end{aligned}$$

So, multisets are defined by recursion equations (possibly higher-order).

A multiset is called *simple* iff it contains at most one element (arbitrarily often). A product $A * B$ is called *admissible* iff B denotes a simple multiset.

Functors

Functor expressions are given by the following grammar.

$$F \quad ::= \quad K\ Void \mid K\ Unit \mid Id \mid (F \mid F) \mid (F \times F) \mid (F \cdot F)$$

Translation of multisets into functors.

$$\begin{array}{cccccc} \varnothing & \wr 0 \int & \wr 1 \int & (\uplus) & (+) & (*) \\ K\ Void & K\ Unit & Id & (\mid) & (\times) & (\cdot) \end{array}$$

Example.

$$\begin{array}{lcl} Nat & = & K\ Unit \mid Id \times Nat \\ Odd & = & Id \times Nat \cdot (Id \times Id) \end{array}$$

Correctness

Proposition. The functor corresponding to a multiset M contains, for each member of M, a container of that size.

☞ This holds if M's definition only involves admissible products.

Datatypes

Functor equations are written in a compositional or 'point-free' style while $\mathbf{data}$ declarations are written in an applicative or 'pointwise' style.

To transform a functor equation into a $\mathbf{data}$ declaration apply both sides to a type variable, say, a and simplify the result using

$$
\begin{array}{lclclcl}
K\ t\ a & = & t & \qquad & (F_1 \mid F_2)\ a & = & F_1\ a \mid F_2\ a \\
\mathit{Id}\ a & = & a & & (F_1 \times F_2)\ a & = & F_1\ a \times F_2\ a \\
 & & & & (F_1 \cdot F_2)\ a & = & F_1\ (F_2\ a)\ .
\end{array}
$$

Example. Note the addition of fancy constructor names.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\mathbf{data}\ \mathit{Odd}\ a & = & \mathit{Odd}\ a\ (\mathit{List}\ (a, a))
\end{array}
$$

Square matrices

Task: generate the set of all square numbers.

A non-solution:

$$\boxed{\quad square \quad = \quad nat * nat \qquad ↯\,. \quad}$$

Corresponds roughly to the lists of lists representation.

Square matrices

Step 1: define the multiset of all natural numbers.

$$nat \quad = \quad \wr 0 \int \uplus \wr 1 \int + nat$$

Step 2: transform this definition into a *tail-recursive* form.

$$\begin{aligned} nat \quad &= \quad nat' \wr 0 \int \\ nat'\ n \quad &= \quad \boxed{n} \uplus nat' (\wr 1 \int + n) \end{aligned}$$

Step 3: replace $\boxed{n}$ by $\boxed{n * n}$.

$$\begin{aligned} square \quad &= \quad square' \wr 0 \int \\ square'\ n \quad &= \quad \boxed{n * n} \uplus square' (\wr 1 \int + n) \end{aligned}$$

NB. The correctness of the preceding steps is easy to establish.

Now, from the functor equations

$$
\begin{array}{lcl}
\mathit{Square} & = & \mathit{Square}'\ (K\ \ \mathit{Unit}) \\
\mathit{Square}'\ f & = & f \cdot f \mid \mathit{Square}'\ (\mathit{Id} \times f)
\end{array}
$$

we can derive the following datatype declarations.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Matrix}\ a & = & \mathit{Matrix}'\ \mathit{Nil}\ a \\
\textbf{data}\ \mathit{Matrix}'\ t\ a & = & \mathit{Zero}\ (t\ (t\ a)) \mid \mathit{Succ}\ (\mathit{Matrix}'\ (\mathit{Cons}\ t)\ a) \\
\textbf{data}\ \mathit{Nil}\ a & = & \mathit{Nil} \\
\textbf{data}\ \mathit{Cons}\ t\ a & = & \mathit{Cons}\ a\ (t\ a)
\end{array}
$$

Here is a square matrix of size 3.

$$\begin{array}{l}
\quad Succ\ (Succ\ (Succ\ (Zero \\
\qquad (Cons\ (Cons\ a_{11}\ (Cons\ a_{12}\ (Cons\ a_{13}\ Nil))) \\
\qquad (Cons\ (Cons\ a_{21}\ (Cons\ a_{22}\ (Cons\ a_{23}\ Nil))) \\
\qquad (Cons\ (Cons\ a_{31}\ (Cons\ a_{32}\ (Cons\ a_{33}\ Nil))) \\
\qquad (Nil)))))))
\end{array}$$

We have essentially a list of lists!

Square matrices with logarithmic access

☞ *Idea:* use a better representation of vectors.

Step 1: base nat on the binary number system.

$$nat \quad = \quad 0 \uplus nat * 2 \uplus 1 + nat * 2$$

NB. This definition corresponds to *random-access lists*.

Step 2: eliminate leading zeros to ensure that the representation is unique (this is merely a cosmetic change).

$$\begin{array}{lcl} nat & = & 0 \uplus pos \\ pos & = & 1 \uplus pos * 2 \uplus 1 + pos * 2 \end{array}$$

Step 3: transform these definitions into a tail-recursive form.

$$\begin{array}{lcl} nat & = & 0 \uplus pos'\ 1 \\ pos'\ n & = & \boxed{n} \uplus pos'\ (2 * n) \uplus pos'\ (1 + 2 * n) \end{array}$$

Step 4: replace $\boxed{n}$ by $\boxed{n * n}$.

$$\begin{array}{lcl} square & = & 0 \uplus square'\ 1 \\ square'\ n & = & \boxed{n * n} \uplus square'\ (2 * n) \uplus square'\ (1 + 2 * n) \end{array}$$

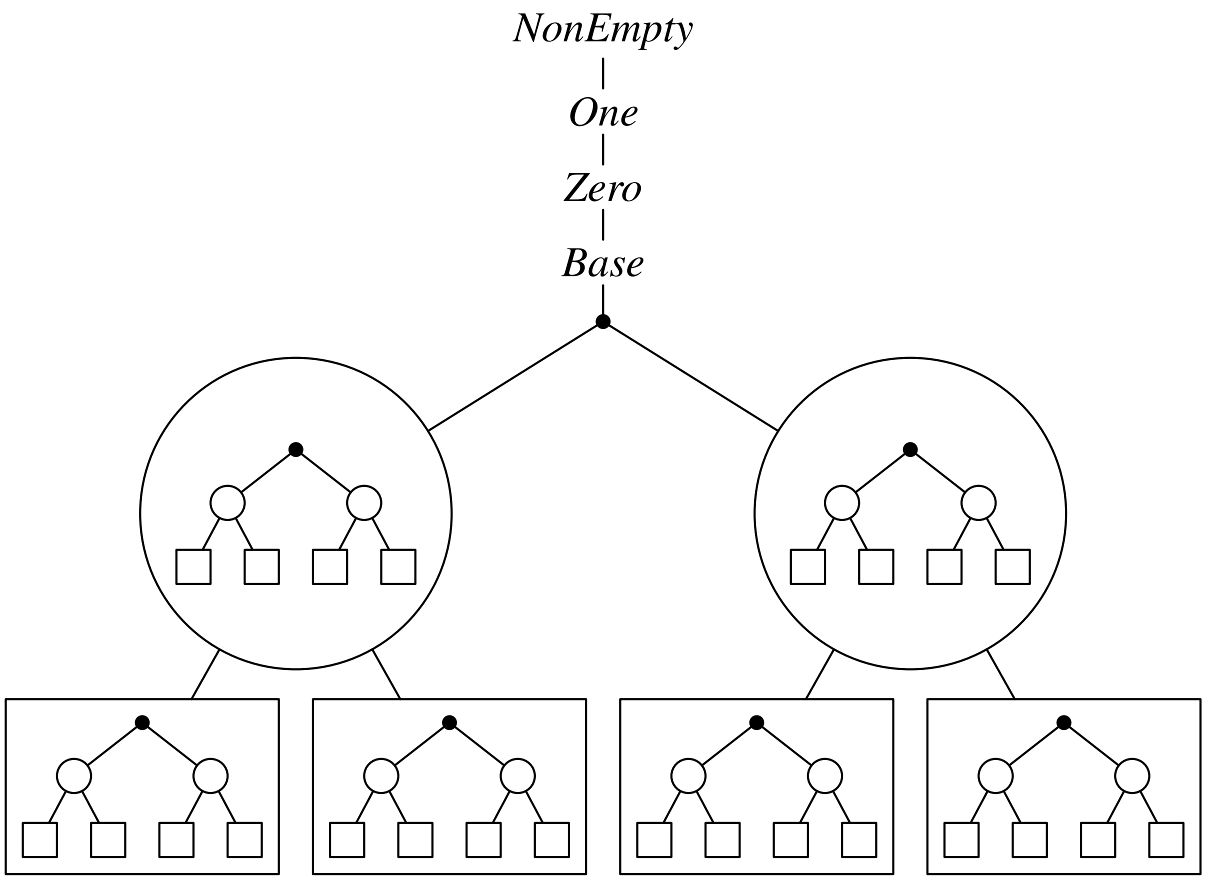
Now, from the functor equations

$$
\begin{array}{lcl}
\mathit{Square} & = & K\ \mathit{Unit} \mid \mathit{Square}'\ \mathit{Id} \\
\mathit{Square}'\ f & = & f \cdot f \mid \mathit{Square}'\ ((\mathit{Id} \times \mathit{Id}) \cdot f) \\
 & & \quad\ \mid \mathit{Square}'\ (\mathit{Id} \times (\mathit{Id} \times \mathit{Id}) \cdot f)
\end{array}
$$

we can derive the following datatype declarations.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Fork}\ t\ a & = & \mathit{Fork}\ (t\ a)\ (t\ a) \\
\mathbf{data}\ \mathit{Node}\ t\ a & = & \mathit{Node}\ a\ (t\ a)\ (t\ a) \\
\mathbf{data}\ \mathit{Matrix}\ a & = & \mathit{Empty} \mid \mathit{NonEmpty}\ (\mathit{Matrix}'\ \mathit{Id}\ a) \\
\mathbf{data}\ \mathit{Matrix}'\ t\ a & = & \mathit{Base}\ (t\ (t\ a)) \\
 & \mid & \mathit{Zero}\ (\mathit{Matrix}'\ (\mathit{Fork}\ t)\ a) \\
 & \mid & \mathit{One}\ (\mathit{Matrix}'\ (\mathit{Node}\ t)\ a)
\end{array}
$$

Here is a 6×6 matrix (note that $6 = (110)_2$).



Résumé

- A small framework for enforcing structural constraints.
- Has been used to model:
  - perfect trees
  - random-access lists
  - rightist right-perfect trees
  - left-complete trees
  - Braun trees
  - triangular matrices
  - square matrices
  - 2-3 trees
  - factorial trees `;-)`.
- *But*, defining functions on these datatypes is non-trivial.
  - type classes help
  - generic programming helps