Manufacturing Datatypes

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany

ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract

This paper describes a general framework for designing purely functional datatypes that automatically satisfy given size or structural constraints. Using the framework we develop implementations of different matrix types (eg square matrices) and implementations of several tree types (eg Braun trees, 2-3 trees). Consider, for instance, representing square $n \times n$ matrices. The usual representation using lists of lists fails to meet the structural constraints: there is no way to ensure that the outer list and the inner lists have the same length. The main idea of our approach is to solve in a first step a related, but simpler problem, namely to generate the multiset of all square numbers. In order to describe this multiset we employ recursion equations involving finite multisets, multiset union, addition and multiplication lifted to multisets. In a second step we mechanically derive datatype definitions from these recursion equations, which enforce the 'squareness' constraint. The transformation makes essential use of polymorphic types.

1 Introduction

Many information structures are defined by certain size or structural constraints. Take, for instance, the class of perfectly balanced, binary leaf trees [10] (perfect leaf trees for short): a perfect leaf tree of height 0 is a leaf and a perfect leaf tree of height $h + 1$ is a node with two children, each of which is a perfect leaf tree of height h. How can we represent perfect leaf trees of arbitrary height such that the structural constraints are enforced? The usual recursive representation of binary leaf trees is apparently not very helpful since there is no way to ensure that the children of a node have the same height. As another example, consider square $n \times n$ matrices [14]. How do we represent square matrices such that the matrices are actually square? Again, the standard representation using lists of lists fails to meet the constraints: the outer list and the inner lists have not necessarily the same length. In this paper, we present a framework that allows to design representations of perfect leaf trees, square matrices, and many other information structures that automatically satisfy the given size or structural constraints.

Let us illustrate the main ideas by means of example. As a first example, we will devise a representation of *Toeplitz matrices* [6] where a Toeplitz matrix is an $n \times n$ matrix (a_{ij}) such that $a_{ij} = a_{i-1,j-1}$ for $1 < i, j \leqslant n$. Clearly, to represent a Toeplitz matrix of size $n + 1$ it suffices to store $2 * n + 1$ elements. Now, instead of designing a representation from scratch we first solve a related, but apparently simpler problem, namely, to generate the set of all odd numbers. Actually, we will work with multisets instead of sets for reasons to be explained later. In order to describe multisets of natural numbers we employ systems of recursion equations. The following system, for instance, specifies the multiset of all odd numbers, ie the multiset which contains one occurrence of each odd number.

$$odd \quad = \quad \wr 1 \wr \uplus \wr 2 \wr + odd$$

Here, $\wr n \wr$ denotes the singleton multiset that contains n exactly once, $(\uplus)$ denotes multiset union and $(+)$ is addition lifted to multisets: $A + B = \wr a + b \mid a \leftarrow A; b \leftarrow B \wr$. We agree upon that $(+)$ binds more tightly than $(\uplus)$. Now, how can we turn the equation into a sensible datatype definition for Toeplitz matrices? The first thing to note is that we are actually looking for a datatype that is parameterized by the type of matrix elements. Such a type is also known as a *type constructor* or as a *functor*[1]. An element of a parameterized type is called a *container*. The equation above has the following counterpart in the world of functors.

$$Odd \quad = \quad Id \mid (Id \times Id) \times Odd$$

Here, Id is the identity functor given by $Id\ a = a$. Furthermore, $(|)$ and $(\times)$ denote disjoint sums and products lifted to functors, ie $(F_1 \mid F_2)\ a = F_1\ a \mid F_2\ a$ and $(F_1 \times F_2)\ a = F_1\ a \times F_2\ a$. Comparing the two equations we see that $\wr 1 \wr$ corresponds to Id, $(\uplus)$ corresponds to $(|)$, and $(+)$ corresponds to $(\times)$. This immediately implies that $Id \times Id$ corresponds to $\wr 1 \wr + \wr 1 \wr = \wr 2 \wr$. The relationship is very tight: the functor corresponding to a multiset M contains, for each member of M, a container of that size. For instance, $Id \times Id$ corresponds to $\wr 1 \wr + \wr 1 \wr = \wr 2 \wr$ as it contains one container of size 2; $Id \mid Id \times Id$ corresponds to $\wr 1 \wr \uplus \wr 1 \wr + \wr 1 \wr = \wr 1, 2 \wr$ as it contains one container of size 1 and another one of size 2.

Functor equations are written in a compositional style. To derive a datatype declaration from a functor equation we simply rewrite it into an applicative form—additionally adding constructor names and possibly making cosmetic changes.[2]

$$\textbf{data}\ Toeplitz\ a \quad = \quad Corner\ a \mid Extend\ a\ a\ (Toeplitz\ a)$$

The left upper corner of a Toeplitz matrix is represented by $Corner\ a$; $Extend\ r\ c\ m$ extends the matrix m by an additional row and an additional column, both of which are represented by elements. For instance, the 4×4 Toeplitz matrix (a_{ij}) is represented by

$$Extend\ a_{41}\ a_{14}\ (Extend\ a_{31}\ a_{13}\ (Extend\ a_{21}\ a_{12}\ (Corner\ a_{11})))\ \ .$$

Of course, this is not the only implementation conceivable. Alternatively, we can define *odd* in terms of the set of all even numbers.

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

As innocent as this variation may look it has the advantage that the left upper corner can be accessed in constant time as opposed to linear time with the first representation.

$$\begin{aligned} \textbf{data}\ Toeplitz\ a &= Toeplitz\ a\ (List2\ a) \\ \textbf{data}\ List2\ a &= Nil2 \mid Cons2\ a\ a\ (List2\ a) \end{aligned}$$

Easier still, we may define *odd* in terms of the natural numbers using the fact that each odd number is of the form $1 + n * 2$ for some n.

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

[1] Categorically speaking, a functor must satisfy additional conditions, see [3]. All the type constructors listed in this paper are functors in the category-theoretical sense.

[2] Examples are given in the functional language Haskell 98 [15].

The first equation makes use of the multiplication operation, which is defined analogously to $(+)$. To which operation on functors does multiplication correspond? We will see that under certain conditions to be spelled out later $(*)$ corresponds to the composition of functors $(\cdot)$ given by $(F_1 \cdot F_2)\ a = F_1\ (F_2\ a)$. The functor equations derived from *odd* and *nat* are

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

Here, $K\ t$ denotes the constant functor given by $K\ t\ a = t$ and *Unit* is the unit type containing a single element. Note that $K\ Unit$ corresponds to $\wr 0 \int$. Unsurprisingly, *Nat* models the ubiquitous datatype of polymorphic lists.

$$\begin{aligned} \mathbf{data}\ Toeplitz\ a &= Toeplitz\ a\ (List\ (a, a)) \\ \mathbf{data}\ List\ a &= Nil \mid Cons\ a\ (List\ a) \end{aligned}$$

Thus, to store an even number of elements we simply use a list of pairs. This representation has the advantage that the list type can be easily replaced by a more efficient sequence type.

Next, let us apply the technique to design a representation of perfect leaf trees. The related problem is simple: we have to generate the multiset of all powers of 2.

$$power = \wr 1 \int \uplus power * \wr 2 \int$$

The corresponding functor equation is

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

from which we can easily derive the following datatype definition.

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

Thus, a perfect leaf tree of height 0 is a leaf and a perfect leaf tree of height $h + 1$ is a perfect leaf tree of height h, whose leaves contain pairs of elements. Note that this definition proceeds *bottom-up* whereas the definition given in the beginning proceeds *top-down*. The type *Perfect* is an example for a so-called *nested datatype* [4]: the recursive call of *Perfect* on the right-hand side is not a copy of the declared type on the left-hand side, ie the type recursion is nested.

As the final example, let us tackle the problem of representing square matrices. We soon find that the related problem of generating the multiset of all square numbers is not quite as easy as before. One could be tempted to define $square = nat * nat$. However, this does not work since the resulting multiset contains products of arbitrary numbers. Incidentally, $nat * nat$ is related to $List \cdot List$, the lists of lists implementation we already rejected. We must somehow arrange that $(*)$ is only applied to singleton multisets. A trick to achieve this is to first rewrite the definition of *nat* into a *tail-recursive* form.

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

The definition of nat' closely resembles the function $from :: Int \to [Int]$ given by $from\ n = n : from\ (n + 1)$, which generates the infinite list of successive integers beginning with n. Now, to obtain square numbers we simply replace n by $n * n$ in the second equation.

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

Using this trick we are, in fact, able to enumerate the codomain of an arbitrary polynomial. Even more interesting, this trick is applicable to other representations of sequences, as well. But, we are skipping ahead. For now, let us determine the datatypes corresponding to *square* and *square'*. 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.

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

The type constructors *Nil* and *Cons t* correspond to $K\ \mathit{Unit}$ and $\mathit{Id} \times f$. As an aside, note that *Nil* and *Cons* are obtained by decomposing the *List* datatype into a base and into a recursive case. Furthermore, note that *Square'* is not a functor but a *higher-order functor* as it takes functors to functors. Accordingly, *Matrix'* is a type constructor of kind $(* \to *) \to (* \to *)$. Recall that the kind system of Haskell specifies the 'type' of a type constructor [12]. The '$*$' kind represents nullary constructors like *Bool* or *Int*. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. Though the type of square matrices looks daunting, it is comparatively easy to construct elements of that type. Here is a square matrix of size 3.

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

Perhaps surprisingly, we have essentially a list of lists! The only difference to the standard representation is that the size of the matrix is additionally encoded into a prefix of *Zero* and *Succ* constructors. It is this prefix that takes care of the size constraints.

This completes the overview. The rest of the paper is organized as follows. Section 2 introduces multisets and operations on multisets. Furthermore, we show how to transform equations into a tail-recursive form. Section 3 explains functors and makes the relationship between multisets and functors precise. A multitude of examples is presented in Section 4: among other things we study random-access lists, Braun trees, 2-3 trees, and square matrices. Finally, Section 5 reviews related work and points out directions for future work.

2 Multisets

A multiset of type $\wr a \wr$ is a collection of elements of type a that takes account of their number but not of their order. In this paper, we will only consider multisets formed according to the following grammar.

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

Here, $\varnothing$ denotes the empty multiset, $\wr n \wr$ denotes the singleton multiset that contains n exactly once, $(\uplus)$ denotes multiset union, $(+)$ and $(*)$ are addition and multiplication lifted to multisets, ie they are defined by $A \otimes B = \wr a \otimes b \mid a \leftarrow A; b \leftarrow B \wr$ for $\otimes \in \{+, *\}$. If the meaning can be resolved from the context, we abbreviate $\wr n \wr$ by n. Furthermore, we agree upon that multiplication takes precedence over addition, which in turn takes precedence over multiset union.

$$
\begin{array}{rcl@{\qquad}rcl}
\wr m \wr + \wr n \wr & = & \wr m + n \wr & A \uplus (B \uplus C) & = & (A \uplus B) \uplus C \\
\wr m \wr * \wr n \wr & = & \wr m * n \wr & A \uplus B & = & B \uplus A \\
\\
A + (B + C) & = & (A + B) + C & \varnothing \uplus A & = & A \\
A + B & = & B + A & \varnothing + A & = & \varnothing \\
0 + A & = & A & \varnothing * A & = & \varnothing \\
\\
A * (B * C) & = & (A * B) * C & (A \uplus B) + C & = & A + C \uplus B + C \\
a * b & = & b * a & (A \uplus B) * C & = & A * C \uplus B * C \\
1 * A & = & A & (A + B) * c & = & A * c + B * c \\
A * 1 & = & A & 0 * A & = & 0
\end{array}
$$

A, B, C are multisets $\qquad$ a, b, c are simple multisets $\qquad$ m, n are natural numbers

Figure 1: Laws of the operations.

Multisets are defined by *higher-order recursion equations*. Higher-order means that the equations may not only involve multisets, but also functions over multisets, function over functions over multisets etc. In this paper, we will, however, restrict ourselves to first-order equations. The exploration of higher-order kinds is the topic of future research. The meaning of higher-order recursion equations is given by the usual least fixpoints semantics.

A multiset is called *simple* iff it is either the empty multiset or a multiset containing a single element arbitrarily often. Simple multiset are denoted by lower case letters. A product $A * B$ is called *admissible* iff B denotes a simple multiset. For instance, $nat * 2$ is admissible while $nat * nat$ is not. We will see in Section 3 that only admissible products correspond to compositions of functors. That is, $nat * 2$ corresponds to $Nat \cdot (Id \times Id)$ but $nat * nat$ does not correspond to $Nat \cdot Nat$. For that reason, we confine ourselves to admissible products when defining multisets.

A multiset is called *unique* iff each element occurs at most once. For instance, the multiset pos given by $pos = 1 \uplus 1 + pos$ is unique whereas $pos = 1 \uplus pos + pos$ denotes a non-unique multiset. Note that the first definition corresponds to non-empty lists and the second to leaf trees. The ability to distinguish between unique and non-unique representations is the main reason for using multisets instead of sets.

The multiset operations satisfy a variety of laws listed in Figure 1. The laws have been chosen so that they hold both for multisets *and* for the corresponding operations on functors. This explains why, for instance, $a * b = b * a$ is restricted to simple multisets: the corresponding property on functors, $F \cdot G = G \cdot F$, does not hold in general. It is valid, however, if G only comprises containers of one size. Of course, for functors the equations state isomorphisms rather than equalities.

In the introduction we have transformed the recursive definition of the multiset of all natural numbers into a tail-recursive form. In the rest of this section we will study this transformation in more detail. A function $h :: \wr a \wr \to \wr a \wr$ on multisets is said to be a *homomorphism* iff $h\ \varnothing = \varnothing$ and $h\ (A \uplus B) = h\ A \uplus h\ B$. For instance, $h\ N = A + N * b$ is a homomorphism while $g\ N = N + N$ is not. Let $h_1, \ldots, h_n$ be homomorphisms, let A be a multiset, and let X be given by

$$X \quad = \quad A \uplus h_1\ X \uplus \cdots \uplus h_n\ X \ .$$

The definition of X is not tail-recursive as the recursive occurrences of X are nested inside function calls. Note that nat is an instance of this scheme with $A = \wr 0 \wr$, $n = 1$, and $h_1\ N = \wr 1 \wr + N$. Now, the *tail-recursive* *variant* of X is $f\ A$ with f given by

$$f\ N \quad = \quad N \uplus f\ (h_1\ N) \uplus \cdots \uplus f\ (h_n\ N) \ \ .$$

The definition of f is called *tail-recursive* for obvious reasons. Note that $nat'\ \wr 0 \wr$ is the tail-recursive variant of nat. The correctness of the transformation is implied by the following theorem.

Theorem 1 *Let $X :: \wr a \wr$, $A :: \wr a \wr$, and $f :: \wr a \wr \to \wr a \wr$ be given as above, then $X = f\ A$.*

3 Functors

In close analogy to multiset expressions we define the syntax of *functor expressions* by the following grammar.

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

Here, $K\ \ t$ denotes the constant functor given by $K\ \ t\ \ a\ =\ t$, *Void* is the empty type, and *Unit* is the unit type containing a single element. By *Id* we denote the identity functor given by $Id\ \ a = a$; $F_1\ \cdot\ F_2$ denotes functor composition given by $(F_1\ \cdot\ F_2)\ \ a = F_1\ \ (F_2\ \ a)$. Disjoint sums and products are defined pointwise: $(F_1 \mid F_2)\ a = F_1\ a \mid F_2\ a$ and $(F_1 \times F_2)\ a = F_1\ a \times F_2\ a$.

All these constructs can be easily defined in Haskell. First of all, we require the following type definitions.

$$\begin{array}{lcl}
\textbf{type}\ Unit & = & () \\
\textbf{data}\ Either\ a_1\ a_2 & = & Left\ a_1 \mid Right\ a_2 \\
\textbf{data}\ (a_1, a_2) & = & (a_1, a_2)
\end{array}$$

The predefined types $Either\ \ a_1\ \ a_2$ and (a_1, a_2) implement disjoint sums and products. The operations on functors are then defined by

$$\begin{array}{lcl}
\textbf{newtype}\ Id\ a & = & Id\ a \\
\textbf{newtype}\ K\ a\ b & = & K\ a \\
\textbf{newtype}\ Sum\ t_1\ t_2\ a & = & Sum\ (Either\ (t_1\ a)\ (t_2\ a)) \\
\textbf{newtype}\ Prod\ t_1\ t_2\ a & = & Prod\ (t_1\ a, t_2\ a) \\
\textbf{newtype}\ Comp\ t_1\ t_2\ a & = & Comp\ (t_1\ (t_2\ a))\ \ .
\end{array}$$

Using these type constructors it is straightforward to translate a functor equation into a Haskell datatype definition. For reasons of readability, we will often define special instances of the general schemes writing *Nil* instead of *K Unit* or *Cons t* instead of *Prod Id t*.

The translation of multisets into functors is given by the following table.

m_1	m_2	$\varnothing$	$\wr 0 \wr$	$\wr 1 \wr$	$m_1 \uplus m_2$	$m_1 + m_2$	$m_1 * m_2$
f_1	f_2	$K\ Void$	$K\ Unit$	Id	$f_1 \mid f_2$	$f_1 \times f_2$	$f_1 \cdot f_2$

We say that F *corresponds to* M if F is obtained from M using this translation. In the rest of this section we will briefly sketch the correctness of the translation. Informally, the functor corresponding to a multiset M contains, for each member of M, a container of that size. This statement can be made precise using the framework of polytypic programming [11]. Briefly, a polytypic function is one that is defined by induction on the structure of functor expressions. A simple example for a polytypic function is $sum\langle f\rangle :: f\ \ \mathbb{N} \to \mathbb{N}$, which sums a structure of natural numbers. To make the relationship between multisets and functors precise we furthermore require the function $fan\langle f\rangle :: a \to \wr f\ \ a \wr$, which generates the multiset of all structures of type $f\ \ a$ from a given seed of type a. For instance, $fan\langle List\rangle\ \ 1$ generates the multiset of all lists that contain 1 as the single element.

Theorem 2 *If the functor F corresponds to the multiset M and if M's definition only involves admissible products, then $M = \wr sum\langle F\rangle\ a \mid a \leftarrow fan\langle F\rangle\ 1 \wr$.*

The following example shows that it is necessary to restrict products to admissible products: if we compose the functors corresponding to $\wr 1, 2 \wr$ and $\wr 1, 3 \wr$, we obtain a functor that corresponds to $\wr 1, 2, 3, 4, 4, 6 \wr$. In general, functor composition corresponds to the multiset operation $(\circledast)$ given by

$$A \circledast B \quad = \quad \wr b_1 + \cdots + b_a \mid a \leftarrow A; b_1 \leftarrow B; \ldots ; b_a \leftarrow B \wr \ .$$

We take a container of type A and fill each of its slots with a container of type B. Summing the sizes of the B containers yields the overall size. The operations $(*)$ and $(\circledast)$ coincide only for admissible products, ie if the containers of type B all have equal size.

4 Examples

In this section we apply the framework to generate efficient implementations of vectors (aka lists or sequences or arrays) and matrices.

4.1 Lists

A vector or a sequence type contains containers of arbitrary size. The problem related to designing a sequence type is, of course, to generate the multiset of all natural numbers. Different ways to describe this set correspond to different implementations of vectors. Perhaps surprisingly, there is an abundance of ways to solve this problem. In the introduction we already encountered the most direct solution:

$$nat_0 \quad = \quad 0 \uplus 1 + nat_0 \ .$$

If we transform the corresponding functor equation

$$Nat_0 \quad = \quad K\ Unit \mid Id \times Nat_0$$

into a Haskell datatype, we obtain the ubiquitous datatype of polymorphic lists.

$$\textbf{data}\ Vector\ a \quad = \quad Nil \mid Cons\ a\ (Vector\ a)$$

As an example, the list representation of the vector $(0, 1, 2, 3, 4, 5)$ is

$$Cons\ 0\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ (Cons\ 4\ (Cons\ 5\ Nil))))) \ .$$

The tail-recursive variant of nat_0 is given by

$$\begin{aligned} nat_1 \quad &= \quad nat_1'\ 0 \\ nat_1'\ n \quad &= \quad n \uplus nat_1'\ (1 + n) \ . \end{aligned}$$

From the functor equations

$$\begin{aligned} Nat_1 \quad &= \quad Nat_1'\ (K\ Unit) \\ Nat_1'\ f \quad &= \quad f \mid Nat_1'\ (Id \times f) \end{aligned}$$

we can derive the following datatype definitions.

$$\begin{aligned} \textbf{type}\ Vector \quad &= \quad Vector'\ Nil \\ \textbf{data}\ Vector'\ t\ a \quad &= \quad Zero\ (t\ a) \mid Succ\ (Vector'\ (Cons\ t)\ a) \end{aligned}$$

Using this representation the vector $(0, 1, 2, 3, 4, 5)$ is written somewhat lengthy as

$$\begin{array}{l} \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\\ \quad \mathit{Cons}\ 0\ (\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ (\mathit{Cons}\ 5\ \mathit{Nil}))))))))))))\ . \end{array}$$

Fortunately, we can simplify the definitions slightly. Recall that Vector' is a type of kind $(* \to *) \to (* \to *)$. In this case the 'higher-orderness' is, however, not required. Noting that the first argument of Vector' is always applied to the second we can transform Vector' into a first-order functor of kind $* \to * \to *$.

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

The two variants of Vector' are related by $\mathit{Vector}'_{ho}\ t\ a = \mathit{Vector}'_{fo}\ (t\ a)\ a$ and $\mathit{Vector}'_{fo}\ t\ a = \mathit{Vector}'_{ho}\ (K\ t)\ a$. Note that the type Matrix' defined in the introduction is not amenable to this transformation since the first argument of Matrix' is used at different instances. Using the first-order definition $(0, 1, 2, 3, 4, 5)$ is represented by

$$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (0, (1, (2, (3, (4, (5, ())))))))))))))\ .$$

4.2 Random-access lists

The definition of nat_0 is based on the unary representation of the natural numbers: a natural number is either zero or the successor of a natural number. Of course, we can also base the definition on the binary number system: a natural number is either zero, even, or odd.

$$\mathit{nat}_2 \quad = \quad 0 \uplus \mathit{nat}_2 * 2 \uplus 1 + \mathit{nat}_2 * 2$$

Transforming the corresponding functor equation

$$\mathit{Nat}_2 \quad = \quad K\ \mathit{Unit} \mid \mathit{Nat}_2 \cdot (\mathit{Id} \times \mathit{Id}) \mid \mathit{Id} \times \mathit{Nat}_2 \cdot (\mathit{Id} \times \mathit{Id})$$

into a Haskell datatype yields

$$\textbf{data}\ \mathit{Vector}\ a \quad = \quad \mathit{Null} \mid \mathit{Zero}\ (\mathit{Vector}\ (a, a)) \mid \mathit{One}\ a\ (\mathit{Vector}\ (a, a))\ .$$

Interestingly, this definition implements *random-access lists* [13], which support logarithmic access to individual vector elements. A random-access list is basically a sequence of perfect leaf trees of increasing height. The vector $(0, 1, 2, 3, 4, 5)$, for instance, is represented by

$$\mathit{Zero}\ (\mathit{One}\ (0, 1)\ (\mathit{One}\ ((2, 3), (4, 5))\ \mathit{Null}))\ .$$

The sequence of *Zero* and *One* constructors encodes the size of the vector in binary representation (with the least significant bit first): we have $(011)_2 = 6$. The representation of a vector of size 11 is depicted in Figure 2(a). Note that the representation is not unique because of leading zeros: the empty sequence, for example, can be represented by *Null*, *Zero Null*, *Zero* (*Zero Null*) etc. There are at least two ways to repair this defect. The following definition ensures that the leading digit is always a one.

$$\begin{array}{lcl} \mathit{nat}_3 & = & 0 \uplus \mathit{pos}_3 \\ \mathit{pos}_3 & = & 1 \uplus \mathit{pos}_3 * 2 \uplus 1 + \mathit{pos}_3 * 2 \end{array}$$

More elegantly, one can define a *zeroless representation* [13], which employs the digits 1 and 2 instead of 0 and 1. We call this variant of the binary number system *1-2 system*.

$$\mathit{nat}_4 \quad = \quad 0 \uplus 1 + \mathit{nat}_4 * 2 \uplus 2 + \mathit{nat}_4 * 2$$

This alternative has the further advantage that accessing the i-th element runs in $O(\log i)$ time [13].

4.3 Fork-node trees

Now, let us transform nat_3 into a tail-recursive form.

$$
\begin{aligned}
nat_5 &= 0 \uplus pos'_5\ 1 \\
pos'_5\ n &= n \uplus pos'_5\ (n * 2) \uplus pos'_5\ (1 + n * 2)
\end{aligned}
$$

Note that we may replace $n * 2$ by $2 * n = n + n$ if pos'_5 is called with a simple multiset as in $pos'_5\ 1$. The corresponding functor equations look puzzling.

$$
\begin{aligned}
Nat_5 &= K\ Unit \mid Pos'_5\ Id \\
Pos'_5\ f &= f \mid Pos'_5\ (f \cdot (Id \times Id)) \mid Pos'_5\ (Id \times f \cdot (Id \times Id))
\end{aligned}
$$

In order to improve the readability of the derived datatypes let us define idioms for $2 * n = n + n$ and $1 + 2 * n = 1 + n + n$.

$$
\begin{aligned}
\mathbf{data}\ Fork\ t\ a &= Fork\ (t\ a)\ (t\ a) \\
\mathbf{data}\ Node\ t\ a &= Node\ a\ (t\ a)\ (t\ a)
\end{aligned}
$$

These definitions assume that t is a simple functor. The alternative definitions **newtype** $Fork\ t\ a = Fork\ (t\ (a, a))$ and **data** $Node\ t\ a = Node\ a\ (t\ (a, a))$, which correspond to $n * 2$ and $1 + n * 2$, work for arbitrary functors but are more awkward to use. Building upon *Fork* and *Node* the Haskell datatypes read

$$
\begin{aligned}
\mathbf{data}\ Vector\ a &= Empty \mid NonEmpty\ (Vector'\ Id\ a) \\
\mathbf{data}\ Vector'\ t\ a &= Base\ (t\ a) \\
&\mid\ Zero\ (Vector'\ (Fork\ t)\ a) \\
&\mid\ One\ (Vector'\ (Node\ t)\ a)\ .
\end{aligned}
$$

A vector of size n is represented by a complete binary tree of height $\lfloor \log_2 n \rfloor + 1$. A node in the i-th level of this tree is labelled with an element iff the i-th digit in the binary decomposition of n is one. The lowest level, which corresponds to a leading one, always contains elements. To the best of the author's knowledge this data structure, which we baptize *fork-node trees* for want of a better name, has not been described elsewhere.[3] Our running example, the vector $(0, 1, 2, 3, 4, 5)$, is represented by

$$
NonEmpty\ (One\ (Zero\ (Base\ (Fork\ (Node\ 0\ (Id\ 1)\ (Id\ 2))\ (Node\ 3\ (Id\ 4)\ (Id\ 5))))))\ .
$$

Again, the size of the vector is encoded into the prefix of constructors: replacing *NonEmpty* and *One* by 1 and *Zero* by 0 yields the binary decomposition of the size *with the most significant bit first*. Figure 2(b) shows a sample vector of 11 elements. The vector elements are stored in left-to-right preorder: if the tree has a root, it contains the first element; the elements in the left tree precede the elements in the right tree. This layout is, however, by no means compelling. Alternatively, one can interleave the elements of the left and the right subtree: if l represents the vector $(b_1, \ldots, b_n)$ and r represents $(c_1, \ldots, c_n)$, then $Fork\ l\ r$ represents the vector $(b_1, c_1, \ldots, b_n, c_n)$ and $Node\ a\ l\ r$ represents $(a, b_1, c_1, \ldots, b_n, c_n)$. This choice facilitates the extension of a vector at the front and also slightly simplifies accessing a vector element.

As always for vector types we can 'firstify' the type definitions.

$$
\begin{aligned}
\mathbf{data}\ Vector\ a &= Empty \mid NonEmpty\ (Vector'\ a\ a) \\
\mathbf{data}\ Vector'\ t\ a &= Base\ t \\
&\mid\ Zero\ (Vector'\ (t, t)\ a) \\
&\mid\ One\ (Vector'\ (a, t, t)\ a)
\end{aligned}
$$

[3] Since this paper was written, I have learned that Hongwei Xi has independently discovered the same data structure.

The representation of $(0,1,2,3,4,5)$ now consists of nested pairs and triples.

$$\mathit{NonEmpty}\ (\mathit{One}\ (\mathit{Zero}\ (\mathit{Base}\ ((0,1,2),(3,4,5)))))$$

Finally, let us remark that the tail-recursive variant of nat_4, which is based on the 1-2 system, yields a similar tree shape: a node on the i-th level contains d elements where d is the i-th digit in the 1-2 decomposition of the vector's size.

4.4 Rightist right-perfect trees

The definition of nat_2 is based on the fact that all natural numbers can be generated by shifting $(n * 2)$ and setting the least significant bit $(1+n*2)$. The following definition sets bits at arbitrary positions by repeatedly shifting a one.

$$\begin{array}{lcl} nat_6 & = & nat'_6\ 1 \\ nat'_6\ p & = & 0 \uplus nat'_6\ (p*2) \uplus p + nat'_6\ (p*2) \end{array}$$

Of course, the two definitions are not unrelated, we have

$$nat_2 * p \quad = \quad nat'_6\ p\ \ ,$$

ie $nat'_6\ p$ generates all multiples of p. In the i-th level of recursion the parameter of nat'_6 equals $p * 2^i$ if the initial call was $nat'_6\ p$. Now, transforming the corresponding functor equations, which assume that f is simple,

$$\begin{array}{lcl} Nat_6 & = & Nat'_6\ Id \\ Nat'_6\ f & = & f \mid Nat'_6\ (f \times f) \mid f \times Nat'_6\ (f \times f) \end{array}$$

into Haskell datatypes yields

$$\begin{array}{lcl} \textbf{type}\ \mathit{Vector} & = & \mathit{Vector}'\ \mathit{Id} \\ \textbf{data}\ \mathit{Vector}'\ t\ a & = & \mathit{Null} \\ & \mid & \mathit{Zero}\ (\mathit{Vector}'\ (\mathit{Fork}\ t)\ a) \\ & \mid & \mathit{One}\ (t\ a)\ (\mathit{Vector}'\ (\mathit{Fork}\ t)\ a)\ \ . \end{array}$$

This datatype implements *higher-order random-access lists* [9]. If we 'firstify' the type constructor Vector', we obtain the first-order variant as defined in Section 4.2. For a discussion of the tradeoffs we refer the interested reader to [9]. The vector $(0,1,2,3,4,5)$ is represented by

$$\mathit{Zero}\ (\mathit{One}\ (\mathit{Fork}\ (\mathit{Id}\ 0)\ (\mathit{Id}\ 1))\ (\mathit{One}\ (\mathit{Fork}\ (\mathit{Fork}\ (\mathit{Id}\ 2)\ (\mathit{Id}\ 3))\ (\mathit{Fork}\ (\mathit{Id}\ 4)\ (\mathit{Id}\ 6)))\ \mathit{Null}))\ \ .$$

Interestingly, using a slight generalization of Theorem 1 we can transform nat'_6 into a tail-recursive form, as well.

$$\begin{array}{lcl} nat_7 & = & nat'_7\ 0\ 1 \\ nat'_7\ n\ p & = & n \uplus nat'_7\ n\ (p*2) \uplus nat'_7\ (n+p)\ (p*2) \end{array}$$

The function nat'_7 is related to nat_2 by

$$n + nat_2 * p \quad = \quad nat'_7\ n\ p\ \ .$$

Assuming that p is simple we get the following functor equations

$$\begin{array}{lcl} Nat_7 & = & Nat'_7\ (K\ Unit)\ Id \\ Nat'_7\ f\ p & = & f \mid Nat'_7\ f\ (p \times p) \mid Nat'_7\ (f \times p)\ (p \times p)\ \ , \end{array}$$

from which we can easily derive the datatype definitions below.

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Vector} & = & \mathit{Vector}'\ (K\ \mathit{Unit})\ \mathit{Id} \\
\textbf{data}\ \mathit{Vector}'\ t\ p\ a & = & \mathit{Base}\ (t\ a) \\
 & | & \mathit{Even}\ (\mathit{Vector}'\ t\ (\mathit{Prod}\ p\ p)\ a) \\
 & | & \mathit{Odd}\ (\mathit{Vector}'\ (\mathit{Prod}\ t\ p)\ (\mathit{Prod}\ p\ p)\ a)
\end{array}$$

This datatype implements *rightist right-perfect trees* or *RR-trees* [7] where the offsprings of the nodes on the left spine form a sequence of perfect leaf trees of decreasing height. Note that if we change $\mathit{Prod}\ t\ p$ to $\mathit{Prod}\ p\ t$ in the last line, we obtain *leftist left-perfect trees*. Here is the vector $(0, 1, 2, 3, 4, 5)$ written as an RR-tree.

$$\begin{array}{l}
\mathit{Even}\ (\mathit{Odd}\ (\mathit{Odd}\ (\mathit{Base}\ (\mathit{Prod}\ (\mathit{Prod}\ (K\ (), \mathit{Prod}\ (\mathit{Id}\ 0, \mathit{Id}\ 1)), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Prod}\ (\mathit{Prod}\ (\mathit{Id}\ 2, \mathit{Id}\ 3), \mathit{Prod}\ (\mathit{Id}\ 4, \mathit{Id}\ 5)))))))
\end{array}$$

Reading the constructors *Even* and *Odd* as digits (LSB first) gives the size of the vector. A sample vector of size 11 is shown in Figure 2(c). The 'firstification' of Vector' is left as an exercise to the reader.

4.5 Braun trees

Let us apply the framework to design a representation of *Braun trees* [5]. Braun trees are node-oriented trees, which are characterized by the following balance condition: for all subtrees, the size of the left subtree is either exactly the size of the right subtree, or one element larger. In other words, a Braun tree of size $2 * n + 1$ has two children of size n and a Braun tree of size $2 * n + 2$ has a left child of size $n + 1$ and a right child of size n. This motivates the following definition.

$$\begin{array}{lcl}
\mathit{braun} & = & \mathit{braun}'\ 0\ 1 \\
\mathit{braun}'\ n\ n' & = & n \uplus \mathit{braun}'\ (n + 1 + n)\ (n' + 1 + n) \\
 & & \ \ \uplus\ \mathit{braun}'\ (n' + 1 + n)\ (n' + 1 + n')
\end{array}$$

The arguments of braun' are always two successive natural numbers. From the corresponding functor equations

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

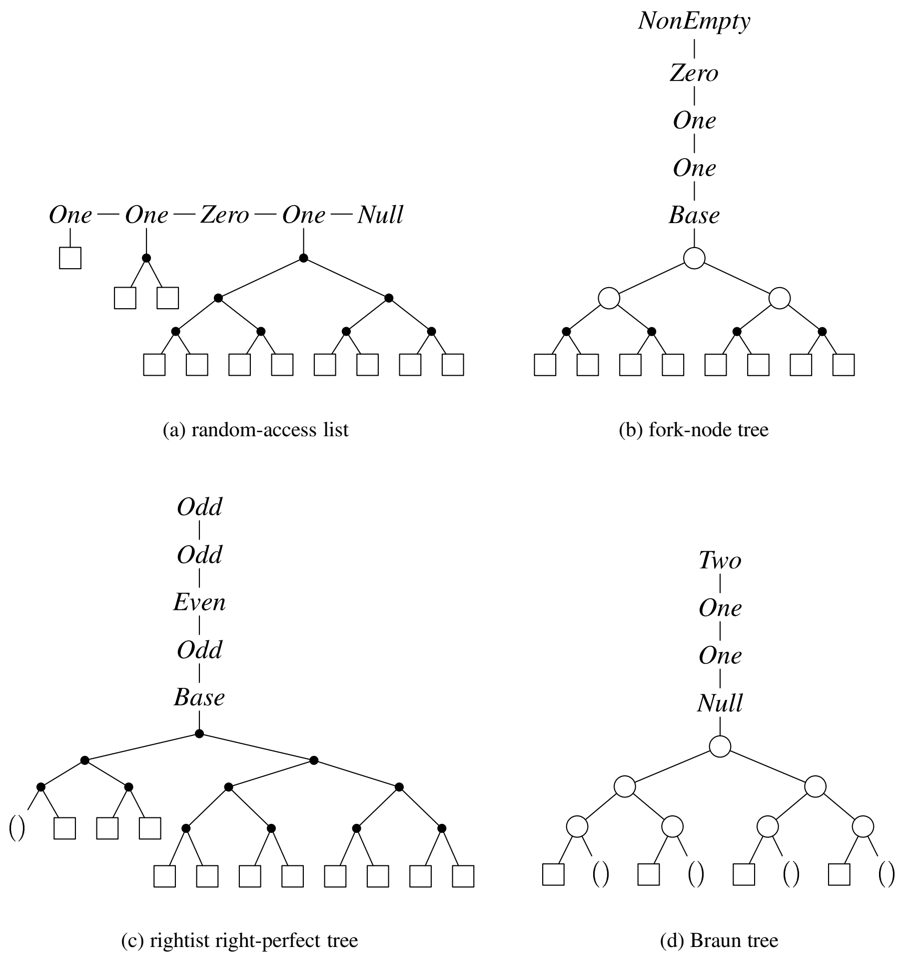
we can derive the following datatype definitions.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Bin}\ t_1\ t_2\ a & = & \mathit{Bin}\ (t_1\ a)\ a\ (t_2\ a) \\
\textbf{type}\ \mathit{Braun} & = & \mathit{Braun}'\ (K\ \mathit{Unit})\ \mathit{Id} \\
\textbf{data}\ \mathit{Braun}'\ t\ t'\ a & = & \mathit{Null}\ (t\ a) \\
 & | & \mathit{One}\ (\mathit{Braun}'\ (\mathit{Bin}\ t\ t)\ (\mathit{Bin}\ t'\ t)\ a) \\
 & | & \mathit{Two}\ (\mathit{Braun}'\ (\mathit{Bin}\ t'\ t)\ (\mathit{Bin}\ t'\ t')\ a)
\end{array}$$

Interestingly, Braun trees are based on the 1-2 number system (MSB first). The vector $(0, 1, 2, 3, 4, 5)$, for instance, is represented as follows.

$$\mathit{Two}\ (\mathit{Two}\ (\mathit{Null}\ (\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Id}\ 0)\ 1\ (\mathit{Id}\ 2))\ 3\ (\mathit{Bin}\ (\mathit{Id}\ 4)\ 5\ (K\ ())))))$$

Figure 2(d) displays the representation of a vector of 11 elements. R. Paterson has described a similar implementation (personal communication).



(a) random-access list

(b) fork-node tree

(c) rightist right-perfect tree

(d) Braun tree

Figure 2: Different representations of a vector with 11 elements. Note that '□' represents a leaf (an element of *Id*), '•' an unlabelled node (an element of $Id \times Id$, *Fork* t, or *Prod* t_1 t_2), and '○' a labelled node (an element of *Node* t or *Bin* t_1 t_2).

4.6 2-3 trees

Up to now we have mainly considered unique representations where the shape of a data structure is completely determined by the number of elements it contains. Interestingly, unique representations are not well-suited for implementing search trees: one can prove a lower bound of $\Omega(\sqrt{n})$ for insertion and deletion in this case [16]. For that reason, popular search tree schemes such as 2-3 trees [2], red-black trees [8], or AVL-trees [1] are always based on non-unique representations. Let us consider how to implement, say, 2-3 trees. The other search tree schemes can be handled in an analogous fashion. The definition of 2-3 trees is similar to that of perfect leaf trees: a 2-3 tree of height 0 is a leaf and a 2-3 tree of height $h + 1$ is a node with either two or three children, each of which is a 2-3 tree of height h. This similarity suggests to model 2-3 trees as follows.

$$\begin{aligned} \mathit{tree23} &= \mathit{tree23}'\ 0 \\ \mathit{tree23}'\ N &= N \uplus \mathit{tree23}'\ (N + 1 + N \uplus N + 1 + N + 1 + N) \end{aligned}$$

Note that contrary to previous definitions the parameter of the auxiliary function does not range over simple sets. The corresponding functor equations

$$\begin{aligned} \mathit{Tree23} &= \mathit{Tree23}'\ (K\ \mathit{Unit}) \\ \mathit{Tree23}'\ F &= F \mid \mathit{Tree23}'\ (F \times \mathit{Id} \times F \mid F \times \mathit{Id} \times F \times \mathit{Id} \times F) \end{aligned}$$

give rise to the following datatype definitions.

$$\begin{aligned} \textbf{type}\ \mathit{Tree23}\ a &= \mathit{Tree23}'\ \mathit{Nil}\ a \\ \textbf{data}\ \mathit{Tree23}'\ t\ a &= \mathit{Zero}\ (t\ a) \mid \mathit{Succ}\ (\mathit{Tree23}'\ (\mathit{Node23}\ t)\ a) \\ \textbf{data}\ \mathit{Node23}\ t\ a &= \mathit{Node2}\ (t\ a)\ a\ (t\ a) \mid \mathit{Node3}\ (t\ a)\ a\ (t\ a)\ a\ (t\ a) \end{aligned}$$

The vector $(0, 1, 2, 3, 4, 5)$ has three different representations; one alternative is

$$\begin{aligned} &\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\mathit{Node3}\ (\mathit{Node3}\ \mathit{Nil}\ 0\ \mathit{Nil}\ 1\ \mathit{Nil})\ 2\ (\mathit{Node2}\ \mathit{Nil}\ 3\ \mathit{Nil}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad 4\ (\mathit{Node2}\ \mathit{Nil}\ 5\ \mathit{Nil})))) \ . \end{aligned}$$

Algorithms for insertion and deletion are described in [9].

4.7 Matrices

Let us finally design representations of square matrices and rectangular matrices. In the introduction we have already discussed the central idea: we take a tail-recursive definition of the natural numbers (or of the positive numbers)

$$\begin{aligned} X &= f\ a \\ f\ n &= n \uplus f\ (h_1\ n) \uplus \cdots \uplus f\ (h_n\ n) \end{aligned}$$

and replace n by $n * n$ in the second equation:

$$\begin{aligned} \mathit{square} &= \mathit{square}'\ a \\ \mathit{square}'\ n &= n * n \uplus \mathit{square}'\ (h_1\ n) \uplus \cdots \uplus \mathit{square}'\ (h_n\ n) \ . \end{aligned}$$

This transformation works provided a is a simple multiset and the h_i preserve simplicity. These conditions hold for all of the examples above with the notable exception of 2-3 trees. As a concrete example, here is an

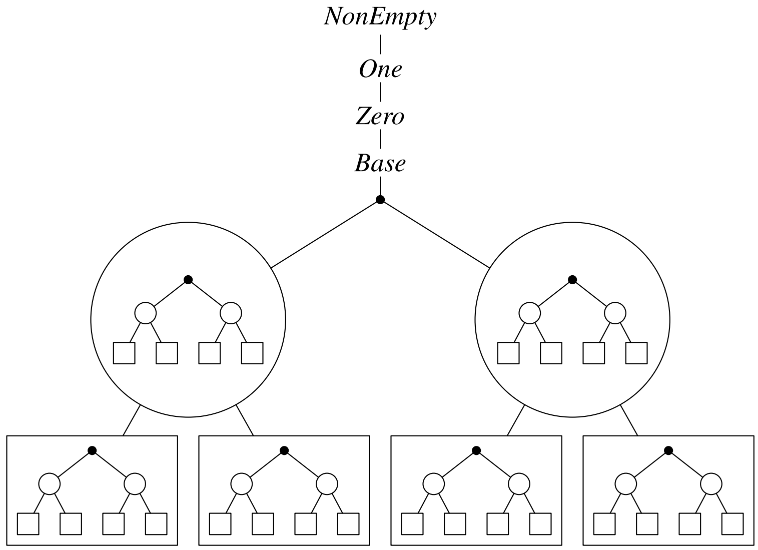


Figure 3: The representation of a 6×6 matrix based on fork-node trees.

implementation of square matrices based on fork-node trees.

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

The representation of a 6×6 matrix is shown in Figure 3.

Rectangular matrices are equally easy to implement. In this case we replace n by $\mathit{nat} * n$ in the second equation:

$$
\begin{array}{lcl}
\mathit{rect} & = & \mathit{rect}'\ a \\
\mathit{rect}'\ n & = & \mathit{nat} * n \uplus \mathit{rect}'\ (h_1\ n) \uplus \cdots \uplus \mathit{rect}'\ (h_n\ n) \quad .
\end{array}
$$

Alternatively, one may use the following scheme.

$$
\begin{array}{lcl}
\mathit{rect} & = & \mathit{rect}'\ a\ a \\
\mathit{rect}'\ m\ n & = & m * n \uplus \mathit{rect}'\ (h_1\ m)\ (h_1\ n) \uplus \cdots \uplus \mathit{rect}'\ (h_1\ m)\ (h_n\ n) \\
 & & \quad \uplus \cdots \\
 & & \quad \uplus\ \mathit{rect}'\ (h_n\ m)\ (h_1\ n) \uplus \cdots \uplus \mathit{rect}'\ (h_n\ m)\ (h_n\ n)
\end{array}
$$

This representation requires more constructors than the first one ($n^2 + 1$ instead of $n + 1$). On the positive side, it can be easily generalized to higher dimensions.

5 Related and future work

This work is inspired by a recent paper of C. Okasaki [14], who derives representations of square matrices from exponentiation algorithms. He shows, in particular, that the tail-recursive version of the fast exponentiation gives rise to an implementation based on rightist right-perfect trees. Interestingly, the simpler implementation based on fork-node trees is not mentioned. The reason is probably that fast exponentiation algorithms typically process the bits from least to most significant bit while fork-node trees and Braun trees are based on the reverse order. The relationship between number systems and data structures is explained at great length in [13]. The development in Section 3 can be seen as putting this design principle on a formal basis.

Extensions to the Hindley-Milner type system that allow to capture structural invariants in a more straightforward way have been described by C. Zenger [18, 19] and H. Xi [17]—the latter paper also appears in the proceedings of this workshop. Using the *indexed types* of C. Zenger one can, for instance, parameterize vectors and matrices by their size. Size compatibility is then statically ensured by the type checker. H. Xi achieves the same effect using dependent datatypes. In his system, *de Caml*, the type of perfect leaf trees is, for instance, declared as follows.

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{datatype}\ `a\ \mathit{perfect}\ \textbf{with}\ \mathit{nat}} \\ = & \mathit{Leaf}\ (0)\ \textbf{of}\ `a \\ \mid & \{n : \mathit{nat}\}\mathit{Fork}\ (n+1)\ \textbf{of}\ `a\ \mathit{perfect}\ (n) * `a\ \mathit{perfect}\ (n) \end{array}$$

This definition is essentially a transliteration of the top-down definition of perfect leaf trees given in the introduction. A practical advantage of dependent types is that standard regular datatypes and functions on these types can be adapted with little or no change. Often it suffices to annotate datatype declarations and type signatures with appropriate size constraints.

Directions for future work suggest themselves. It remains to adapt the standard vector and matrix algorithms to the new representations. Some preparatory work has been done in this respect. In [9] the author shows how to adapt search tree algorithms to nested representations of search trees using constructor classes. It is conceivable that this approach can be applied to matrix algorithms, as well. Furthermore, many functions like *map*, *listify*, *sum* etc can be generated automatically using the technique of polytypic programming [11]. On the theoretical side, it would be interesting to investigate the expressiveness of the framework and of higher-order polymorphic types in general. Which class of multisets can be described using higher-order recursion equations? For instance, it appears to be impossible to specify the multisets of all prime numbers. Do higher-order kinds increase the expressiveness?

Acknowledgements

I am grateful to Iliano Cervesato, Lambert Meertens, and John O'Donnell for many valuable comments.

References

[1] G.M. Adel'son-Vel'skiĭ and Y.M. Landis. An algorithm for the organization of information. *Doklady Akademiia Nauk SSSR*, 146:263–266, 1962. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[3] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall Europe, London, 1997.

[4] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.

[5] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology, 1983.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1991.

[7] Victor J. Dielissen and Anne Kaldewaij. A simple, efficient, and flexible implementation of flexible arrays. In *Third International Conference on Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany*, volume 947 of *Lecture Notes in Computer Science*, pages 232–241. Springer-Verlag, July 1995.

[8] Leo J. Guibas and Robert Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[9] Ralf Hinze. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, December 1998.

[10] Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 1999. To appear.

[11] Ralf Hinze. Polytypic functions over nested datatypes (extended abstract). In *3rd Latin-American Conference on Functional Programming (CLaPF'99)*, March 1999.

[12] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[14] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, Paris, France*, 1999. To appear.

[15] Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999.

[16] Lawrence Snyder. On uniquely represented data structures (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence*, pages 142–146, Long Beach, Ca., USA, October 1977. IEEE Computer Society Press.

[17] Hongwei Xi. Dependently typed data structures. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France*, September 1999.

[18] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, November 1997.

[19] Christoph Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, 1998.