Manufacturing datatypes

RALF HINZE
Institut für Informatik III, Universität Bonn,
Römerstraße 164, 53117 Bonn, Germany
(e-mail: ralf@informatik.uni-bonn.de)

Abstract

This article 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 (for example, square matrices) and implementations of several tree types (for example, Braun trees and 2-3 trees). Consider 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. 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 from these recursion equations datatype definitions that enforce the 'squareness' constraint. The transformation makes essential use of parameterized types.

Die ganze Zahl schuf der liebe Gott, alles Übrige ist Menschenwerk.
— Leopold Kronecker

1 Introduction

Many information structures are defined by certain size or structural constraints. Take, for instance, the class of perfectly balanced, binary leaf trees (Hinze, 2000a) (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 (Okasaki, 1999). 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 are not necessarily of the same length. In this article, we present a framework for designing 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 (Cormen et al., 1991) where a Toeplitz
matrix is an \( n \times n \) matrix \((a_{ij})\) such that \( a_{ij} = a_{i-1,j-1} \) for \( 1 < i, j \leq n \). Clearly, to represent a Toeplitz matrix of size \((n+1) \times (n+1)\) it suffices to store \( 2n + 1 \) 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 (see section 2). 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, that is, the multiset that contains one occurrence of each odd number.

\[
\text{odd} = \{1\} \uplus \text{odd} + \{2\}
\]

Here, \( \{n\} \) denotes the singleton multiset that contains \( n \) exactly once, ‘\( \uplus \)’ denotes multiset union and ‘\( + \)’ is addition lifted to multisets:

\[
A + B = \{a + b \mid a \leftarrow A; b \leftarrow B\}
\]

We agree upon the convention 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 \textit{type constructor}, as a \textit{container type} (Hoogendijk & de Moor, 2000), or as a \textit{functor}\(^1\). An element of a parameterized type is called a \textit{container}. Now, the equation above has the following counterpart in the world of functors:

\[
\text{Odd} = \text{Id} | \text{Odd} \times (\text{Id} \times \text{Id})
\]

Here, \( \text{Id} \) is the identity functor given by \( \text{Id} a = a \). Furthermore, ‘\( | \)’ and ‘\( \times \)’ denote sums and products lifted to functors, that is, \((F_1 | F_2) a = F_1 a | F_2 a \) and \((F_1 \times F_2) a = F_1 a \times F_2 a \). Comparing the two equations we see that \( \{1\} \) corresponds to \( \text{Id} \), the operation ‘\( \uplus \)’ corresponds to ‘\( | \)’, and ‘\( + \)’ corresponds to ‘\( \times \)’. This immediately implies that \( \text{Id} \times \text{Id} \) corresponds to \( \{1\} + \{1\} = \{2\} \). The relationship between multisets and functors is very tight: the functor corresponding to a multiset \( M \) contains, for each member of \( M \), a container of that size. For instance, \( \text{Id} \times \text{Id} \) corresponds to \( \{1\} + \{1\} = \{2\} \) as it contains one container of size 2; the functor \( \text{Id} | \text{Id} \times \text{Id} \) corresponds to \( \{1\} \uplus \{1\} + \{1\} = \{1, 2\} \) as it contains one container of size 1 and another 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. For concreteness, examples are given in the functional language Haskell 98 (Peyton Jones & Hughes, 1999).

\[
\text{data Toeplitz } a = \text{Corner } a | \text{Extend } (\text{Toeplitz } a) a a
\]

The left upper corner of a Toeplitz matrix is represented by \( \text{Corner } a \); using

\(^1\) Categorically speaking, a functor must satisfy additional conditions, see, for instance Bird and de Moor (1997). All the type constructors listed in this article are functors in the category-theoretical sense.
Extend m r c we extend the matrix m by an additional row and an additional
column, both of which are represented by single elements. For instance, the $4 \times 4$
Toeplitz matrix $(a_{ij})$ is represented by

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

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

$$
\text{odd} = \{1\} + \text{even}
$$

$$
\text{even} = \{0\} \uplus \{2\} + \text{even}
$$

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.

Using the natural numbers, we may define odd in terms of the fact that
each odd number is of the form $1 + n \times 2$ for some $n$.

$$
\text{odd} = \{1\} + \text{nat} \times \{2\}
$$

$$
\text{nat} = \{0\} \uplus \{1\} + \text{nat}
$$

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 ‘$\times$’ corresponds
to the composition of functors ‘·’ given by $(F_1 \cdot F_2) a = F_1 (F_2 a)$. The functor
equations derived from odd and nat are

$$
\text{Odd} = \text{Id} \times \text{Nat} \cdot (\text{Id} \times \text{Id})
$$

$$
\text{Nat} = \text{K Unit} \uplus \text{Id} \times \text{Nat}.
$$

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 $\{0\}$. Unsurprisingly, Nat models the ubiquitous datatype of lists.

Using the natural numbers, we may define odd in terms of the set of all even numbers.

$$
\text{odd} = \{1\} + \text{even}
$$

$$
\text{even} = \{0\} \uplus \{2\} + \text{even}
$$

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.

Using the natural numbers, we may define odd in terms of the fact that
each odd number is of the form $1 + n \times 2$ for some $n$.

$$
\text{odd} = \{1\} + \text{nat} \times \{2\}
$$

$$
\text{nat} = \{0\} \uplus \{1\} + \text{nat}
$$

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 ‘$\times$’ corresponds
to the composition of functors ‘·’ given by $(F_1 \cdot F_2) a = F_1 (F_2 a)$. The functor
equations derived from odd and nat are

$$
\text{Odd} = \text{Id} \times \text{Nat} \cdot (\text{Id} \times \text{Id})
$$

$$
\text{Nat} = \text{K Unit} \uplus \text{Id} \times \text{Nat}.
$$

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 $\{0\}$. Unsurprisingly, Nat models the ubiquitous datatype of lists.

$$
\text{data Toeplitz a} = \text{Toeplitz a (List2 a)}
$$

$$
\text{data List2 a} = \text{Nil2} \mid \text{Cons2 a a (List2 a)}
$$

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 ‘$\times$’ corresponds
to the composition of functors ‘·’ given by $(F_1 \cdot F_2) a = F_1 (F_2 a)$. The functor
equations derived from odd and nat are

$$
\text{Odd} = \text{Id} \times \text{Nat} \cdot (\text{Id} \times \text{Id})
$$

$$
\text{Nat} = \text{K Unit} \uplus \text{Id} \times \text{Nat}.
$$

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 $\{0\}$. Unsurprisingly, Nat models the ubiquitous datatype of lists.

$$
\text{data Toeplitz a} = \text{Toeplitz a (List a, a)}
$$

$$
\text{data List a} = \text{Nil} \mid \text{Cons a (List a)}
$$

Thus, to store an even number of elements we simply use a list of pairs. The $4 \times 4$
Toeplitz matrix $(a_{ij})$ now reads

$$
\text{Toeplitz a_{11} (Cons (a_{21}, a_{12}) (Cons (a_{31}, a_{13}) (Cons (a_{41}, a_{14}) Nil)))}.
$$
The last representation has the advantage of being modular: we can easily replace the list type 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.

\[
\text{power} = \{1\} \cup \text{power} \ast \{2\}
\]

The corresponding functor equation is

\[
\text{Power} = \text{Id} \mid \text{Power} \cdot (\text{Id} \times \text{Id}),
\]

from which we can easily derive the following datatype definition:

\[
data \text{Perfect } a = \text{Zero } a \mid \text{Succ } (\text{Perfect } (a, a)).
\]

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 of the introduction on page 493 proceeds top-down. The type \text{Perfect} is an example of a so-called nested datatype (Bird & Meertens, 1998): the recursive call of \text{Perfect} on the right-hand side is not a copy of the declared type on the left-hand side, that is, the type recursion is nested. It is revealing to have a closer look at the types. The table below illustrates the construction of an element of type \text{Perfect } \text{Int} \ ($$ always refers to the expression in the preceding row and \text{I} abbreviates \text{Int}$).

\[
\begin{align*}
(((1, 2), (3, 4)), ((5, 6), (7, 8))) & \quad :: \quad (((1, 1), (1, 1)), ((1, 1), (1, 1))) \\
\text{Zero } $$ & \quad :: \quad \text{Perfect } (((1, 1), (1, 1)), ((1, 1), (1, 1))) \\
\text{Succ } $$ & \quad :: \quad \text{Perfect } ((1, 1), (1, 1)) \\
\text{Succ } $$ & \quad :: \quad \text{Perfect } (1, 1) \\
\text{Succ } $$ & \quad :: \quad \text{Perfect } 1
\end{align*}
\]

We start with a pair of pairs of pairs of integers. Note that the type expression has the same size as the value expression. Using the constructor \text{Zero} the nested pair is turned into a leaf. Now, each application of \text{Succ} halves the size of the type expression. In each case the typechecker ensures that the elements are pairs of the same type. Note that the height of the perfect leaf tree is encoded in the prefix of \text{Succ} and \text{Zero} constructors.

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 as easy as before. It is tempting to define \text{square} = \text{nat} \ast \text{nat}. However, this does not work since the resulting multiset contains products of arbitrary numbers. Incidentally, \text{nat} \ast \text{nat} is related to \text{List} \cdot \text{List}, the lists of lists implementation we already rejected. We must somehow arrange that \"\ast\" is only applied to singleton multisets. A trick to achieve this is to first rewrite the definition of \text{nat} into a tail-recursive form.

\[
\begin{align*}
\text{nat} & \quad = \quad \text{nat'} 0 \\
\text{nat'} n & \quad = \quad n \uplus \text{nat'} (1 + n)
\end{align*}
\]

The definition of \text{nat'} closely resembles the function \:: \text{Int} \to \text{[Int]} given by

\[
\text{from } :: \text{Int} \to \text{[Int]}
\]

Published online by Cambridge University Press
Manufacturing datatypes

497

from \( n = n : \text{from} \ (n + 1) \), which generates the infinite list of successive integers beginning with \( n \). Now, to obtain square numbers we simply replace \( n \) by \( n \times n \) in the second equation.

\[
\begin{align*}
\text{square} & = \text{square}' \ 0_j \\
\text{square}' n & = n \times n \uplus \text{square}' (1_j + n)
\end{align*}
\]

To see how the multiset of square numbers is constructed, let us unfold the definition of \( \text{square} \) a few steps:

\[
\begin{align*}
\text{square}' (0_j) & = (0_j \uplus \text{square}' (1_j)) \\
& = (0_j \uplus (1_j \uplus \text{square}' (2_j))) \\
& = (0_j \uplus (1_j \uplus (2_j \uplus \text{square}' (3_j)))) \\
& = \ldots .
\end{align*}
\]

Using this trick we are, in fact, able to enumerate the image of an arbitrary polynomial (whose coefficients are natural numbers). Even more interesting, this trick is not only applicable to lists but to other representations of sequences, as well. But, we are skipping ahead. For now, let us determine the datatypes corresponding to \( \text{square} \) and \( \text{square}' \). From the functor equations

\[
\begin{align*}
\text{Square} & = \text{Square}' (K \ \text{Unit}) \\
\text{Square}' f & = f \cdot f \mid \text{Square}' (\text{Id} \times f)
\end{align*}
\]

we can derive the following datatype declarations:

\[
\begin{align*}
\text{type} \ \text{Square} \ a & = \text{Square}' \ \text{Nil} \ a \\
\text{data} \ \text{Square}' \ t \ a & = \text{Zero} \ (t \ (t \ a)) \mid \text{Succ} \ (\text{Square}' \ (\text{Cons} \ t) \ a) \\
\text{data} \ \text{Nil} \ a & = \text{Nil} \\
\text{data} \ \text{Cons} \ t \ a & = \text{Cons} \ a \ (t \ a).
\end{align*}
\]

The type constructors \( \text{Nil} \) and \( \text{Cons} \ t \) correspond to \( K \ \text{Unit} \) and \( \text{Id} \times f \). As an aside, note that \( \text{Nil} \) and \( \text{Cons} \) are obtained by decomposing the \( \text{List} \) datatype into a base and into a recursive case (see section 5). Furthermore, note that \( \text{Square}' \) is not a functor but a higher-order functor as it takes functors to functors, that is, \( \text{Square}' \) is a type constructor of kind \((* \rightarrow *) \rightarrow (* \rightarrow *)\). Recall that the kind system of Haskell specifies the 'type' of a type constructor (Jones, 1995). The \('*'\) kind represents manifest types like \text{Bool} or \text{Int}. The kind \( \kappa_1 \rightarrow \kappa_2 \) represents type constructors that map type constructors of kind \( \kappa_1 \) to those of kind \( \kappa_2 \). The order of a kind is given by \( \text{order}(\kappa) = 0 \) and \( \text{order}(\kappa_1 \rightarrow \kappa_2) = \max \{1 + \text{order}(\kappa_1), \text{order}(\kappa_2)\} \). Thus, \( \text{Square}' \) has a kind of order 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.

\[
\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Cons} \ (\text{Cons} \ a_{11} \ (\text{Cons} \ a_{12} \ (\text{Cons} \ a_{13} \ \text{Nil})))) \\
(\text{Cons} \ (\text{Cons} \ a_{21} \ (\text{Cons} \ a_{22} \ (\text{Cons} \ a_{23} \ \text{Nil})))) \\
(\text{Cons} \ (\text{Cons} \ a_{31} \ (\text{Cons} \ a_{32} \ (\text{Cons} \ a_{33} \ \text{Nil})))))
\]

https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
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 \textit{Zero} and \textit{Succ} constructors. It is this prefix that takes care of the size constraints. The following table shows the construction of \textit{Succ}^3 (\textit{Zero} m) in more detail (\(f^n a\) means \(f\) applied \(n\) times to \(a\)):

\[
\begin{align*}
m &:: \text{Cons}^3 \text{ Nil} (\text{Cons}^3 \text{ Nil} \text{ Int}) \\
\text{Zero} \text{ SS} &:: \text{Square}' (\text{Cons}^3 \text{ Nil}) \text{ Int} \\
\text{Succ} \text{ SS} &:: \text{Square}' (\text{Cons}^2 \text{ Nil}) \text{ Int} \\
\text{Succ} \text{ SS} &:: \text{Square}' (\text{Cons} \text{ Nil}) \text{ Int} \\
\text{Succ} \text{ SS} &:: \text{Square'} \text{ Nil} \text{ Int} = \text{Square} \text{ Int}.
\end{align*}
\]

Roughly speaking, the outer applications of the value constructor \textit{Cons} make sure that the inner lists have the same length and \textit{Zero} checks that the inner lists have the same length as the outer list.

This completes the overview. The rest of the article 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. Section 5 shows how to adapt vector and matrix operations to the new representations. Section 6 reviews related work and points out a direction for future work. Finally, Appendix A lists the proofs of the theorems.

### 2 Multisets

A multiset of type \([T]\) is a collection of elements of type \(T\) that takes account of their number but not of their order. A multiset \(M\) can be modelled as a function into the natural numbers such that \(Mx\) is the number of occurrences of \(x\) in \(M\).

In order to give a meaning to recursion equations such as \(M = [1] \cup M\), we also allow elements to occur infinitely often. Thus, the set of multisets over \(T\) is given by \([T]\) where \(\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}\). On \([T]\) we can define the usual pointwise order \(M \sqsubseteq N \iff \forall t \in T. M t \leq N t\) where \(\leq\) is the standard order on the naturals additionally setting \(n \leq \infty\). Then \([T]\), \(\sqsubseteq\) forms a complete partial order.

In this article, we will only consider multisets formed according to the following grammar:

\[
M ::= \emptyset | [0] | [1] | (M \cup M) | (M + M) | (M \ast M).
\]

Here, \(\emptyset\) denotes the empty multiset, \([n]\) with \(n \in \{0, 1\}\) denotes the singleton multiset that contains \(n\) exactly once, \(\cup\) denotes multiset union, ‘+’ and ‘\(\ast\)’ are addition and multiplication lifted to multisets: \(A \oplus B = [a \oplus b] \iff A : b \oplus B\) for \(\oplus \in \{+, \ast\}\). Note that the operations \(\cup: [T] \rightarrow [T]\) and \((+, \ast): [\text{Nat}] \rightarrow [\text{Nat}] \rightarrow [\text{Nat}]\) are continuous with respect to \(\sqsubseteq\). If the meaning can be resolved from the context, we abbreviate \([n]\) by \(n\), where \([n]\ = [1] + [n-1]\) for \(n > 1\). Furthermore,
we assume that multiplication takes precedence over addition, which in turn takes precedence over multiset union.

Multisets are defined by higher-order recursion equations. Higher-order means that the equations may involve not only multisets, but also functions over multisets, functions over functions over multisets, etc. We will, however, only make use of 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. As an example, consider the following equations:

\[
\begin{align*}
\mathcal{M}_1 &= \mathcal{H}_1 \uplus \mathcal{M}_1 \\
\mathcal{M}_2 &= \mathcal{H}_1 \uplus \mathcal{M}_2 \\
\mathcal{M}_3 &= \mathcal{H}_1 \uplus \mathcal{M}_3.
\end{align*}
\]

The first equation defines the multiset that contains 1 infinitely often. Since we take the least fixpoint, \(\mathcal{M}_2\) and \(\mathcal{M}_3\) both define the empty multiset.

A multiset is called \textit{simple} iff it contains exactly one element. Simple multisets are denoted by lower case letters. A product \(A \times B\) is called \textit{admissible} iff \(B\) denotes a simple multiset. For instance, \(\text{nat} \times 2\) is admissible while \(\text{nat} \times \text{nat}\) is not. We will see in section 3 that only admissible products correspond to compositions of functors. That is, \(\text{nat} \times 2\) corresponds to \textit{Nat} \cdot (\text{Id} \times \text{Id})\) but \(\text{nat} \times \text{nat}\) does not correspond to \textit{Nat} \cdot \textit{Nat}. For that reason, we restrict ourselves to admissible products when defining multisets.

A multiset is called \textit{unique} iff each element occurs at most once. For instance, the multiset \(\text{pos}\) given by \(\text{pos} = 1 \uplus 1+\text{pos}\) is unique whereas \(\text{pos} = 1 \uplus \text{pos} + \text{pos}\) 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 \textit{and} for the corresponding...
operations on functors. This explains why, for instance, \( a \ast b = b \ast a \) 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 \( F \) and \( G \) each comprise only a single container. 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 continuous function \( h :: \{ T \} \rightarrow \{ T \} \) on multisets is said to be a homomorphism iff \( h \emptyset = \emptyset \) and \( h (A \uplus B) = h A \uplus h B \). For instance, \( h N = A + N \ast b \) is a homomorphism while \( g N = N + N \) is not. Let \( A \) be a multiset, let \( h_1, \ldots , h_n \) be homomorphisms, and let \( X \) be given by

\[
X = A \uplus h_1 X \uplus \cdots \uplus h_n X.
\]

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

\[
f Y = Y \uplus f (h_1 Y) \uplus \cdots \uplus f (h_n Y).
\]

The definition of \( f \) is called tail-recursive—think of \( \uplus \) as a conditional. Note that \( \text{nat}' \{ 0 \} \) is the tail-recursive variant of \( \text{nat} \). The correctness of the transformation is implied by the following theorem, whose proof is given in Appendix A.1.

**Theorem 1**

Let \( X :: \{ T \}, A :: \{ T \}, \) and \( f :: \{ T \} \rightarrow \{ T \} \) 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:

\[
F ::= K \text{ Void} | K \text{ Unit} | \text{Id} | (F \mid F) | (F \times F) | (F \cdot F).
\]

Here, \( K t \) denotes the constant functor given by \( K t a = t \), \( \text{Void} \) is the empty type, and \( \text{Unit} \) is the unit type containing a single element. By \( \text{Id} \) we denote the identity functor given by \( \text{Id} a = a \) and \( F_1 \cdot F_2 \) denotes functor composition given by \( (F_1 \cdot F_2) a = F_1 (F_2 a) \). Sums and products are defined pointwise:

\[
(F_1 \mid F_2) a = F_1 a \mid F_2 a \quad \text{and} \quad (F_1 \times F_2) a = F_1 a \times F_2 a.
\]

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

```haskell
2 Note that Void was defined in Haskell 1.4; it is, however, no longer part of Haskell 98.
```
The predefined types \( \text{Either } a_1 \ a_2 \) and \((a_1, a_2)\) implement sums and products. The operations on functors are then defined by

\[
\begin{align*}
\text{newtype } \text{Id } a & = \text{Id } a \\
\text{newtype } K \ a \ b & = K \ a \\
\text{newtype } \text{Sum } t_1 \ t_2 \ a & = \text{Sum } (\text{Either } (t_1 \ a) (t_2 \ a)) \\
\text{newtype } \text{Prod } t_1 \ t_2 \ a & = \text{Prod } (t_1 \ a, t_2 \ a) \\
\text{newtype } \text{Comp } t_1 \ t_2 \ a & = \text{Comp } (t_1 (t_2 \ a)).
\end{align*}
\]

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 \( \text{Nil} \) instead of \( K \ Unit \) or \( \text{Cons } t \) instead of \( \text{Prod } \text{Id } t \).

Remark 1

We tacitly assume that we are working in a strict rather than in a lazy setting. Otherwise, the datatypes additionally contain unwanted elements such as infinite and partial elements. For example, in a lazy setting, \( \text{Unit} \) typically contains two elements: the desired unit element and \( \bot \). In the case of sum and product types we can avoid extra elements using strictness annotations

\[
\begin{align*}
data \ \text{Either } a_1 \ a_2 & = \text{Left } !a_1 \ | \ \text{Right } !a_2 \\
data (a_1, a_2) & = (!a_1, !a_2),
\end{align*}
\]

but we refrain from being that pedantic. As an aside, note that we do not use Standard ML for the examples because Standard ML has only first-order kinded datatypes.

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

\[
\begin{array}{cccccccc}
M_1 & M_2 & | & 0 & | & 1 & | & M_1 \uplus M_2 & | & M_1 + M_2 & | & M_1 \ast M_2 \\
F_1 & F_2 & | & K \ Void & | & K \ Unit & | & \text{Id} & | & F_1 \ | \ F_2 & | & F_1 \times F_2 & | & F_1 \cdot F_2 \\
\end{array}
\]

We say that \( F \) corresponds to \( M \) if \( F \) is obtained from \( M \) using this translation. In the rest of this section we will sketch the correctness of the translation (the proof can be found in Appendix A.2). 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 (Hinze, 2000b). Briefly, a polytypic function is one that is defined by induction on the structure of functor expressions. An example of such a function is \( \text{sum}(F) : F \ \text{Nat} \rightarrow \text{Nat} \), which sums a structure of natural numbers. To make the relationship between multisets and functors precise we furthermore require the function \( \text{fan}(F) : a \rightarrow \{ F \ a \} \), which generates the multiset of all structures of type \( F \ a \) from a given seed of type \( a \). For instance, \( \text{fan}(\text{List}) \ 1 = \{ \text{Nil}, \text{Cons} \ 1 \ \text{Nil}, \text{Cons} \ 1 (\text{Cons} \ 1 \ \text{Nil}), \text{Cons} \ 1 (\text{Cons} \ 1 (\text{Cons} \ 1 \ \text{Nil})) \ldots \} \).

https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
Theorem 2
If the functor $F$ corresponds to the multiset $M$ and if $M$’s definition only involves admissible products, then $M = \{\sum_{i} a \mid a \leftarrow \text{fan}(F)\}$.

The following example shows that it is necessary to restrict products to admissible products: if we compose the functors corresponding to $H_1$, $H_2$ and $H_1$, $3$, we obtain a functor that corresponds to $\{1, 3\} \uplus (\{1, 3\} + \{1, 3\}) = \{1, 2, 3, 4, 6\}$. In general, functor composition corresponds to the non-commutative multiset operation ‘$\uplus$’ given by

$$A \uplus B = \{b_1 + \cdots + b_n \mid a \leftarrow A; b_1 \leftarrow B; \ldots; b_n \leftarrow B\}.$$

In words, 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 ‘$\ast$’ and ‘$\uplus$’ coincide only for admissible products, that is, if $B$ contains only one container so that $b_1 = \cdots = b_n$.

4 Examples
In this section we apply the framework to generate efficient implementations of vectors (also known as lists or sequences or arrays) and matrices.

4.1 Lists
A vector or sequence type comprises 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:

$$\text{nat}_0 = 0 \uplus 1 + \text{nat}_0.$$ 

If we transform the corresponding functor equation

$$\text{Nat}_0 = K \text{Unit} | \text{Id} \times \text{Nat}_0$$

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

```haskell
data Vector a = Nil | Cons a (Vector a)
```

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

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

The tail-recursive variant of $\text{nat}_0$ is given by

$$\text{nat}_1 = \text{nat}_1' 0$$

$$\text{nat}_1' n = n \uplus \text{nat}_1' (1 + n).$$

From the functor equations

$$\text{Nat}_1 = \text{Nat}_1' (K \text{Unit})$$

$$\text{Nat}_1' f = f \mid \text{Nat}_1' (\text{Id} \times f)$$
we can derive the following datatype definitions:

\[
\begin{align*}
\text{type } \text{Vector} & = \text{Vector'} \text{Nil} \\
\text{data } \text{Vector'} \ t \ a & = \text{Zero} \ (t \ a) \ | \ \text{Succ} \ (\text{Vector'} \ (\text{Cons} \ t) \ a).
\end{align*}
\]

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

\[
\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Zero} \ ((\text{Cons} \ 0) \ (\text{Cons} \ 1) \ (\text{Cons} \ 2) \ (\text{Cons} \ 3) \ (\text{Cons} \ 4) \ (\text{Cons} \ 5) \ (\text{Nil}))))))))
\]

Fortunately, we can simplify the definitions slightly. Recall that \(\text{Vector'}\) is a type constructor of kind \(\ast \rightarrow \ast \rightarrow \ast\). However, in this case the 'higher-orderness' is not required. Noting that the first argument of \(\text{Vector'}\) is always applied to the second, we can transform \(\text{Vector'}\) into a first-order functor of kind \(\ast \rightarrow \ast \rightarrow \ast\).

\[
\begin{align*}
\text{type } \text{Vector} & = \text{Vector'} () \\
\text{data } \text{Vector'} \ t \ a & = \text{Zero} \ t \ | \ \text{Succ} \ (\text{Vector'} \ (a, t) \ a)
\end{align*}
\]

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

\[
\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Succ} \ (\text{Zero} \ ((\text{Cons} \ 0) \ (\text{Cons} \ 1) \ (\text{Cons} \ 2) \ (\text{Cons} \ 3) \ (\text{Cons} \ 4) \ (\text{Cons} \ 5) \ (\text{Nil}))))))))
\]

### 4.2 Random-access lists

The definition of \(\text{nat}_0\) 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.

\[
\text{nat}_2 \ = \ 0 \uplus \text{nat}_2 \uplus 2 \uplus 1 + \text{nat}_2 \uplus 2
\]

Transforming the corresponding functor equation

\[
\text{Nat}_2 \ = \ K \ \text{Unit} \ | \ \text{Nat}_2 \cdot (\text{Id} \times \text{Id}) \ | \ \text{Id} \times \text{Nat}_2 \cdot (\text{Id} \times \text{Id})
\]

into a Haskell datatype yields

\[
\begin{align*}
\text{data } \text{Vector} \ a & = \text{End} \ | \ \text{Zero} \ (\text{Vector} \ (a, a)) \ | \ \text{One} \ (\text{Vector} \ (a, a)).
\end{align*}
\]

Interestingly, this definition implements random-access lists (Okasaki, 1998), 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

\[
\text{Zero} \ (\text{One} \ (0, 1) \ (\text{One} \ ((2, 3), (4, 5)) \ \text{End})).
\]

The sequence of \(\text{Zero}\) and \(\text{One}\) constructors encodes the size of the vector in binary representation (with the least significant bit first). In this example we have \((011)_2 = 6\). The representation of a vector of size 11 is depicted in figure 2(a)—the
remaining examples in figure 2 will be discussed in the following few sections. Note
that the representation is not unique because of trailing zeros: the empty sequence,
for example, can be represented by \text{End}, \text{Zero End}, \text{Zero (Zero End)}, etc. There are
at least two ways to repair this defect. The following definition ensures that the last
digit is always a one:

\[
\text{nat}_3 = 0 \uplus \text{pos}_3
\]

\[
\text{pos}_3 = 1 \uplus \text{pos}_3 \ast 2 \uplus 1 + \text{pos}_3 \ast 2.
\]

More elegantly, one can define a \textit{zeroless representation} (Okasaki, 1998), which
employs the digits 1 and 2 instead of 0 and 1. We call this variant of the binary
number system the 1-2 system.

\[
\text{nat}_4 = 0 \uplus 1 + \text{nat}_4 \ast 2 \uplus 2 + \text{nat}_4 \ast 2
\]

This alternative has the further advantage that accessing the \(i\)-th element runs in \(O(\log i)\) time (Okasaki, 1998).

### 4.3 Fork-node trees

Now, let us transform \(\text{nat}_3\) into a tail-recursive form.

\[
\text{nat}_5 = 0 \uplus \text{pos}'_5 1
\]

\[
\text{pos}'_5 n = n \uplus \text{pos}'_5 (n \ast 2) \uplus \text{pos}'_5 (1 + n \ast 2)
\]

The corresponding functor equations look puzzling.

\[
\text{Nat}_5 = K \text{Unit} \mid \text{Pos}'_5 \text{Id}
\]

\[
\text{Pos}'_5 f = f \mid \text{Pos}'_5 (f \cdot (\text{Id} \times \text{Id})) \mid \text{Pos}'_5 (\text{Id} \times f \cdot (\text{Id} \times \text{Id}))
\]

Note that we may replace \(n \ast 2\) by \(2 \ast n\) in the definition of \(\text{pos}'_5\) if the function is
called with a simple multiset as in \(\text{pos}'_5 1\). In order to improve the readability of the
derived datatypes let us define idioms for \(2 \ast n = n + n\) and \(1 + 2 \ast n = 1 + n + n\):

\[
\text{data Fork} t a = \text{Fork} (t a) (t a)
\]

\[
\text{data Node} t a = \text{Node} a (t a) (t a).
\]

These definitions assume that \(t\) is a simple functor, that is, a functor which contains
exactly one container. The alternative definitions \textbf{newtype} \(\text{Fork}' t a = \text{Fork}' (t (a,a))\)
and \textbf{data} \(\text{Node}' t a = \text{Node}' a (t (a,a))\), which correspond to \(n \ast 2\) and \(1 + n \ast 2\),
work for arbitrary functors but are more awkward to use. Building upon \(\text{Fork}\) and
\(\text{Node}\), the Haskell datatypes read

\[
\text{data Vector} a = \text{Empty} \mid \text{NonEmpty} (\text{Vector}' \text{Id} a)
\]

\[
\text{data Vector}' t a = \text{Base} (t a)
\]

A vector of size \(n\) is represented by a complete binary tree of height \(\lceil \log_2 n \rceil + 1\).
A node in the \(i\)-th level of this tree is labelled with an element iff the \(i\)-th digit
Fig. 2. Different representations of a vector with 11 elements. Note that ‘$\mathsf{D}$’ represents a leaf (an element of $\mathsf{Id}$), ‘$\mathsf{O}$’ an unlabelled node (an element of $\mathsf{Id} \times \mathsf{Id}$, $\mathsf{Fork}$ $t_1 \mathsf{t}_2$, or $\mathsf{Prod}$ $t_1 \mathsf{t}_2$), and ‘$\mathsf{L}$’ a labelled node (an element of $\mathsf{Node}$ $t_1$ or $\mathsf{Bin}$ $t_1 \mathsf{t}_2$).

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

$$\text{NonEmpty} \left( \text{One} \left( \text{Zero} \left( \text{Base} \left( \text{Fork} \left( \text{Node} \left( \text{Id} \ 1 \right) \left( \text{Id} \ 2 \right) \right) \left( \text{Node} \left( \text{Id} \ 4 \right) \left( \text{Id} \ 5 \right) \right) \right) \right) \right) \right).$$

Again, the size of the vector is encoded into the prefix of constructors: replacing $\text{NonEmpty}$ and $\text{One}$ by 1 and $\text{Zero}$ 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.

\(^3\) I have learned, however, that Hongwei Xi has independently discovered the same data structure.
The vector elements are stored in left-to-right preorder in the example above: 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 \( \text{Fork } l \ r \) represents \((b_1,c_1,\ldots,b_n,c_n)\) and \( \text{Node } a \ l \ r \) 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 (see section 5.2).

As usual for vector types we can ‘firstify’ the type definitions.

\[
\begin{align*}
\text{data } & \text{Vector } a &= \text{Empty } | \text{NonEmpty } (\text{Vector’ } a \ a) \\
\text{data } & \text{Vector’ } t \ a &= \text{Base } t \\
& & | \text{Zero } (\text{Vector’ } (t,t) \ a) \\
& & | \text{One } (\text{Vector’ } (a,t,t) \ a)
\end{align*}
\]

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

\[
\text{NonEmpty } (\text{One } (\text{Zero } (\text{Base } ((0,1,2),(3,4,5)))))
\]

Finally, let us remark that the tail-recursive variant of \(\text{nat}_4\), 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 \(\text{nat}_2\) is based on the fact that all natural numbers can be generated by shifting \((n \times 2)\) and setting the least significant bit \((1 + n \times 2)\). The following definition sets bits by repeatedly shifting a one:

\[
\begin{align*}
\text{nat}_6 &= \text{nat}_6 1 \\
\text{nat}_6 p &= 0 \uplus \text{nat}_6 (p \times 2) \uplus p + \text{nat}_6 (p \times 2).
\end{align*}
\]

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

\[
\text{nat}_2 \times p = \text{nat}_6 p,
\]

that is, \(\text{nat}_6 p\) generates all multiples of \(p\). In the \(i\)-th level of recursion the parameter of \(\text{nat}_6\) equals \(p \times 2^i\) if the initial call was \(\text{nat}_6 p\). Now, transforming the corresponding functor equations

\[
\begin{align*}
\text{Nat}_6 &= \text{Nat}_6 \text{Id} \\
\text{Nat}_6 f &= f \uplus \text{Nat}_6 (f \times f) \uplus f \times \text{Nat}_6 (f \times f),
\end{align*}
\]

which assume that \(f\) is simple, into Haskell datatypes yields

\[
\begin{align*}
\text{type } & \text{Vector } &= \text{Vector’ } \text{Id} \\
\text{data } & \text{Vector’ } t \ a &= \text{End} \\
& & | \text{Zero } (\text{Vector’ } (\text{Fork } t \ a)) \\
& & | \text{One } (t \ a) (\text{Vector’ } (\text{Fork } t \ a)).
\end{align*}
\]
This datatype implements higher-order random-access lists (Hinze, 1998). 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 Hinze (1998). The vector (0, 1, 2, 3, 4, 5) is represented by

Zero (One (Fork (Id 0) (Id 1))
(One (Fork (Id 2) (Id 3)) (Fork (Id 4) (Id 6)) End)).

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

nat 7 = nat′ 0 1
nat′ n p = n ⊎ nat′ n (p * 2) ⊎ nat′ (n + p) (p * 2)

The function nat′ is related to nat 2 by

n + nat 2 * p = nat′ n p.

Assuming that p is simple we get the following functor equations:

Nat 7 = Nat′ (K Unit) Id
Nat′ f p = f | Nat′ f (p × p) | Nat′ (f × p) (p × p),

from which we can easily derive the datatype definitions below.

This datatype implements rightist right-perfect trees or RR-trees (Dielissen & Kalde-waij, 1995) where the offsprings of the nodes on the left spine form a sequence of perfect leaf trees of increasing height (assuming that we traverse the spine top-down starting at the root). Note that if we change Prod t p to 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.

Even (Odd (Odd (Base (Prod (Prod (K ()), Prod (Id 0, Id 1)),
Prod (Prod (Id 2, Id 3), Prod (Id 4, Id 5)))))),

Reading the constructors Even and Odd as digits (least significant bit 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 (Braun & Rem, 1983). Braun trees are node-oriented trees, that is, the inner nodes are labelled with elements. They 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

https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
one element larger. In other words, a Braun tree of size $2 \cdot n + 1$ has two children of size $n$ and a Braun tree of size $2 \cdot n + 2$ has a left child of size $n + 1$ and a right child of size $n$. This motivates the following definition:

\[
\text{braun} = \text{braun'} 0 1 \\
\text{braun'} n n' = n \sqcup \text{braun'} (1 + n + n) (1 + n' + n) \\
\sqcup \text{braun'} (1 + n' + n) (1 + n + n').
\]

A similar decomposition is used in Okasaki (1997) to initialize a Braun tree. The arguments of $\text{braun'}$ are always two successive natural numbers. From the corresponding functor equations

\[
\text{Braun} = \text{Braun'} (K \text{ Unit}) \text{ Id} \\
\text{Braun'} f f' = f | \text{Braun'} (\text{Id} \times f \times f') (\text{Id} \times f' \times f) \\
| \text{Braun'} (\text{Id} \times f' \times f') (\text{Id} \times f' \times f')
\]

we can derive the following datatype definitions:

\[
\text{data} \text{ Bin} \ t_1 \ t_2 \ a = \text{ Bin} \ a (t_1 \ a) (t_2 \ a) \\
\text{type} \text{ Braun} = \text{Braun'} (K \ \text{ Unit}) \text{ Id} \\
\text{data} \text{ Braun'} \ t \ t' \ a = \text{ Base} (t \ a) \\
| \text{ One} (\text{Braun'} (\text{Bin} \ t \ t) (\text{Bin} \ t' \ t) \ a) \\
| \text{ Two} (\text{Braun'} (\text{Bin} \ t' \ t) (\text{Bin} \ t' \ t') \ a).
\]

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

\[
\text{Two} (\text{Two} (\text{Base} (\text{Bin} 3 (\text{Bin} 1 (\text{Id} 0) (\text{Id} 2)) (\text{Bin} 5 (\text{Id} 4) (K ()))))).
\]

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

### 4.6 Left-complete trees

Braun trees are a popular data structure for implementing heaps (also known as priority queues). Another common heap data structure is the left-complete tree (heap for short), which has the leaves on the lowest level in the leftmost possible positions. These trees underlie, for instance, heapsort (Williams, 1964).

The following characterization of heaps, which is due to Ross Paterson (personal communication), proceeds by induction on the height: a heap of height 0 is an empty tree, a heap of height $h + 1$ is either a node whose left subtree is a heap of height $h$ and whose right subtree is a perfect tree of height $h - 1$ or a node whose left subtree is a perfect tree of height $h$ and whose right subtree is a heap of height $h$. Note that both heaps and perfect trees like Braun trees are node-oriented. Furthermore, note that by definition there are no perfect trees of height $-1$ (so there is only one heap of height 1). The characterization suggests the following tail-recursive definition of
heaps:

\[
\begin{align*}
heap_1 &= heap'_1 0 \not\in 0 \\
heap'_1 H \ p \ p' &= H \uplus heap'_1(1 + H + p \uplus 1 + p' + H) \ p'(1 + p' * 2).
\end{align*}
\]

The first argument of \(heap_1\) comprises heaps of height \(h\), the second argument is a perfect tree of height \(h - 1\), and the third is a perfect tree of height \(h\). In other words, the \(n\)-th recursive call equals \(heap'_1\) \{a + 1, \ldots, a' + 1\} \(a'\) where \(a = 2^{n-1} - 1\) and \(a' = 2^n - 1\). Interestingly, this is the first definition where \(\uplus\) is used in an argument position. It is worth noting, however, that though the first argument of \(heap'_1\) is not simple, \(heap_1\) denotes a unique multiset. Now, transforming the corresponding functor equations

\[
\begin{align*}
Heap_1 &= Heap'_1 (K \ Unit) (K \ Void) (K \ Unit) \\
Heap'_1 h \ f \ f' &= h \mid Heap'_1(Id \times h \times f \mid Id \times f' \times h) f'(Id \times f' \cdot (Id \times Id))
\end{align*}
\]

into Haskell datatypes yields

\[
\begin{align*}
type \ Heap &= Heap' (K \ Unit) (K \ Void) (K \ Unit) \\
data \ Heap' h \ t \ t' a &= \text{Zero} (h \ a) \\
&\mid \text{Succ} (Heap' (\text{Sum} (\text{Bin} h \ t) (\text{Bin} t' h)) t' (\text{Node} t') a).
\end{align*}
\]

The representation of the vector \((0,1,2,3,4,5)\) is rather lengthy.

\[
\begin{align*}
\text{Succ} (\text{Succ} (\text{Zero} (\text{right} (\text{Bin} 3 (\text{Node} 0 (\text{Node} 1 (K)))) (K)))) \\
&\quad (\text{Node} 2 (K)) ((K))) \\
&\quad (\text{left} (\text{Bin} 5 (\text{right} (\text{Bin} 4 (K)))) (K))) \\
&\quad (K)))
\end{align*}
\]

Here, \(\text{left} a\) abbreviates \(\text{Sum} (\text{Left} a)\) and \(\text{right} a\) abbreviates \(\text{Sum} (\text{Right} a)\). Note that the term encodes both the height and the size of the heap. The prefix of \(\text{Succ}\) and \(\text{Zero}\) constructors represents the height. The sequence of \(\text{left}\) and \(\text{right}\) constructors encodes the size: if we replace \(\text{left}\) by 0, \(\text{right}\) by 1, remove the trailing 1 and add a leading 1, then we obtain the binary decomposition of the size (most significant bit first). Figure 3(a) shows a heap containing 11 elements. The representation of heaps is unwieldy mainly because \(\uplus\) is used in an argument position. Fortunately, we can simplify the definition of \(heap'_1\) by noting that \(h \rightarrow heap_1 h p p'\) for fixed \(p\) and \(p'\) is a homomorphism (this can be shown by a simple fixpoint induction). Pushing \(\uplus\) to the outside and simplifying the base case we obtain

\[
\begin{align*}
heap_2 &= heap'_2 0 \not\in 0 \ 1 \\
heap'_2 h \ p \ p' &= h \uplus heap'_2(1 + h + p) p'(1 + p' * 2) \\
&\quad \uplus heap'_2(1 + p' + h) p'(1 + p' * 2).
\end{align*}
\]

Here are the corresponding functor equations

\[
\begin{align*}
Heap_2 &= Heap'_2 (K \ Unit) (K \ Unit) Id \\
Heap'_2 h \ f \ f' &= h \mid Heap'_2(Id \times h \times f) f'(Id \times f' \cdot (Id \times Id)) \\
&\quad \mid Heap'_2(Id \times f' \times h) f'(Id \times f' \cdot (Id \times Id))
\end{align*}
\]
and finally the Haskell datatype declarations, which assume that \( t' \) is simple.

\[
\begin{align*}
\text{type } & \text{Heap } = \text{Heap'} (K \text{ Unit}) (K \text{ Unit}) Id \\
\text{data } & \text{Heap'} \ h \ t \ t' \ a = \text{Base} (h \ a) \\
& \quad \mid \text{One} (\text{Heap'} (\text{Bin} h t) t' (\text{Node} t') a) \\
& \quad \mid \text{Two} (\text{Heap'} (\text{Bin} t' h) t' (\text{Node} t') a)
\end{align*}
\]

Perhaps surprisingly, the transformation yields a representation that is based on the 1-2 number system (least significant bit first). For background information the interested reader is referred to Hinze (1999), which explains the relationship between heaps and different number systems in more detail. The representation of the vector \((0, 1, 2, 3, 4, 5)\) is much more compact now.

\[
\text{Two} (\text{Two} (\text{Base} (\text{Bin} 3 (\text{Node} 0 (Id 1) (Id 2)) (\text{Bin} 5 (Id 4) (K ()))))))
\]

Figure 3(b) displays a heap comprising 11 elements.

### 4.7 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 (Snyder, 1977). For that reason, popular search tree schemes such as 2-3 trees (Aho et al., 1983), red-black trees (Guibas & Sedgewick, 1978), and AVL-trees (Adel’son-Vel’skiĭ & Landis, 1962) 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 modelling 2-3 trees as follows:

$$
\text{tree23} = \text{tree23}' 0
$$
$$
\text{tree23}' N = N \uplus \text{tree23}' (N + 1 + N \uplus N + 1 + N + 1 + N).
$$

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

$$
\text{Tree23} = \text{Tree23}' (K \text{Unit})
$$
$$
\text{Tree23}' F = F \uplus \text{Tree23}' (F \times \text{Id} \times F \uplus F \times \text{Id} \times F \times \text{Id} \times F)
$$

give rise to the following datatype definitions:

```
type Tree23 a = Tree23' Nil a
data Tree23' t a = Zero (t a) | Succ (Tree23' (Node23 t) a)
data Node23 t a = Node2 (t a) a (t a) | Node3 (t a) a (t a) a (t a).
```

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

$$
\text{Succ} (\text{Succ} (\text{Zero} (\text{Node3} (\text{Node Nil 0 Nil 1 Nil} 2 (\text{Node2 Nil 3 Nil}) 4 (\text{Node2 Nil 5 Nil}))))).
$$

Algorithms for insertion and deletion are described in Hinze (1998).

### 4.8 Square and rectangular 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)

$$
X = f a
$$
$$
f y = y \uplus f (h_1 y) \uplus \cdots \uplus f (h_n y)
$$

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

$$
\text{square} = \text{square}' a
$$
$$
\text{square}' y = y * y \uplus \text{square}' (h_1 y) \uplus \cdots \uplus \text{square}' (h_n y).
$$

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

https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
exception of 2-3 trees. As a concrete example, here is an implementation of square matrices based on fork-node trees.

```
data Square a = Empty | NonEmpty (Square' Id a)
data Square' t a = Base (t (t a))
   | Zero (Square' (Fork t) a)
   | One (Square' (Node t) a)
```

The representation of a $6 \times 6$ matrix is shown in figure 4.

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

$$rect = rect' a$$
$$rect' y = \text{nat} \ast y \uplus rect' (h_1 y) \uplus \cdots \uplus rect' (h_n y).$$

Alternatively, one may use the following scheme:

$$rect = rect' a a$$
$$rect' y z = y \ast z \uplus rect' (h_1 y) (h_1 z) \uplus \cdots \uplus rect' (h_1 y) (h_n z) \uplus \cdots \uplus rect' (h_n y) (h_1 z) \uplus \cdots \uplus rect' (h_n y) (h_n z).$$

This representation requires more constructors than the first one ($n^2 + 1$ instead of $n + 1$). On the positive side, it can easily be generalized to higher dimensions.
Many of the Haskell datatype declarations we have seen so far are unusual, some even mind-boggling. The question naturally arises how the standard vector and matrix operations can be adapted to these new representations.

Fortunately, several functions can be generated automatically using the technique of polytypic programming (Hinze, 2000b). A useful polytypic function is the so-called mapping function \( \text{map}(F) \) \( \forall a \ b \ (a \rightarrow b) \rightarrow (F \ a \rightarrow F \ b) \), which applies a given function to each element of type \( a \) in a given container of type \( F \ a \). The mapping function can be used, for instance, to multiply a vector or a matrix by a scalar: \( \text{map}(F) \ (\text{times} \ c) \) where \( \text{times} \ a \ b = a \times b \). A related function is the zipping function \( \text{zip}(F) \) \( \forall a \ b \ c \ (a \rightarrow b \rightarrow c) \rightarrow (F \ a \rightarrow F \ b \rightarrow F \ c) \), which takes two containers of the same shape and combines them into a single container. Zipping with ‘+’, for instance, implements vector and matrix addition. Further examples for polytypic functions include equality and comparison functions, pretty printers (such as Haskell’s \text{show} function), and parsers (such as Haskell’s \text{read} function).

In the rest of this section we explain how to implement non-polytypic operations in a fairly systematic manner. For concreteness, we use vector and matrix representations based on fork-node trees. Recall that a fork-node tree is a complete binary tree, where the nodes in each level are either consistently labelled or consistently unlabelled. The bottom-up definition of fork-node trees given in section 4.3 captures these structural constraints. As usual, the straightforward top-down definition

\[
\text{data Tree } a = \text{Id } a \mid \text{Fork } (\text{Tree } a) (\text{Tree } a) \mid \text{Node } a (\text{Tree } a) (\text{Tree } a)
\]

fails to meet the requirements. Interestingly though, both definitions are closely related. Consider the definitions of the subsidiary types that are used in \text{Vector} and \text{Vector’}:

\[
\begin{align*}
\text{newtype Id } a &= \text{Id } a \\
\text{data Fork } \text{tree } a &= \text{Fork } (\text{tree } a) (\text{tree } a) \\
\text{data Node } \text{tree } a &= \text{Node } a (\text{tree } a) (\text{tree } a)
\end{align*}
\]

These types can be seen as decompositions of the regular type constructor \text{Tree}: instead of a single type consisting of three constructors we have three types each consisting of a single constructor, where the recursive type calls are turned into type arguments. Of course, this is not a coincidence since \text{Vector} and \text{Vector’} result from an application of Theorem 1: \text{Id} corresponds to the multiset \( A \) and \text{Fork} and \text{Node} correspond to the homomorphisms \( h_i \). Generally, every datatype that stems from the transformation into a tail-recursive form incorporates a similar decomposition.

Now, the basic idea for implementing operations on fork-node trees is to let ‘function follow type’. We start by defining the required operation on regular trees, that is, on elements of type \text{Tree}. This is usually the creative part. Then we decompose this function into a base and two recursive cases mirroring the decomposition of \text{Tree} into \text{Id}, \text{Fork}, and \text{Node}. Finally, we write ‘wrapping code’ for \text{Vector} and
R. Hinze

Vector'. The development in this section is essentially based on Hinze (1998), which shows how to adapt search tree algorithms to nested representations of search trees.

5.1 Vector Traversal

Let us illustrate the basic idea with a simple example: collecting the elements of a vector from left to right in a list. The type of the traversal function is given by

\[
\text{type } \text{Flatten } t = \forall a. t \to [a].
\]

Note that \text{Flatten} is parameterized by the vector type, which is of kind \( * \to * \). Furthermore, note that this type declaration is not legal Haskell 98 since it introduces a polymorphic type on the right-hand side. Type synonyms like \text{Flatten} will, however, prove to be very useful.

We assume that the vector elements are stored as follows: if \( l \) represents the vector \((b_1, \ldots, b_n)\) and \( r \) represents \((c_1, \ldots, c_n)\), then \text{Fork } l \ r \ represents the vector \((b_1, c_1, \ldots, b_n, c_n)\) and \text{Node } a \ l \ r \ represents \((a, b_1, c_1, \ldots, b_n, c_n)\). Translating this description into Haskell yields the desired traversal function:

\[
\begin{align*}
\text{flatten} & \quad \mathbin:: \quad \text{Flatten Tree} \\
\text{flatten } (\text{Id } a) & = [a] \\
\text{flatten } (\text{Fork } l \ r) & = \text{interleave } (\text{flatten } l) (\text{flatten } r) \\
\text{flatten } (\text{Node } a \ l \ r) & = a : \text{interleave } (\text{flatten } l) (\text{flatten } r) \\
\text{interleave} & \quad \mathbin:: \quad \forall a. [a] \to [a] \\
\text{interleave } [] \mathbin:: \quad as_2 & = as_2 \\
\text{interleave } (a : as_1) \mathbin:: \quad as_2 & = a : \text{interleave } as_2 as_1.
\end{align*}
\]

Note that \text{flatten} is defined by structural recursion over the datatype \text{Tree}, that is, \text{flatten} is a so-called \textit{catamorphism} (Meijer et al., 1991). This renders it easy to decompose the function.

\[
\begin{align*}
\text{flattenId} & \quad \mathbin:: \quad \text{Flatten Id} \\
\text{flattenId } (\text{Id } a) & = [a] \\
\text{flattenFork} & \quad \mathbin:: \quad \forall t. \text{Flatten } t \to \text{Flatten } (\text{Fork } t) \\
\text{flattenFork } flt \mathbin:: \quad (\text{Fork } l \ r) & = \text{interleave } (flt \ l) (flt \ r) \\
\text{flattenNode} & \quad \mathbin:: \quad \forall t. \text{Flatten } t \to \text{Flatten } (\text{Node } t) \\
\text{flattenNode } flt \mathbin:: \quad (\text{Node } a \ l \ r) & = a : \text{interleave } (flt \ l) (flt \ r)
\end{align*}
\]

The recursive calls are turned into function arguments mirroring the transformations on the type level. Again, we make use of an extension to Haskell’s type system: both \text{flattenFork} and \text{flattenNode} have so-called \textit{rank-2 type signatures} (Leivant, 1983),
that is, they operate on polymorphic functions. Now, in defining the traversal function for `Vector` we follow again the structure of the datatype definition.

```haskell
flattenVector :: Flatten Vector
flattenVector Empty = []
flattenVector (NonEmpty v) = flattenVector' flattenId v
flattenVector' :: \t. Flatten t \rightarrow Flatten (Vector' t)
flattenVector' flt (Base v) = flt v
flattenVector' flt (Zero v) = flattenVector' (flattenFork flt) v
flattenVector' flt (One v) = flattenVector' (flattenNode flt) v
```

While recursing `flattenVector'` constructs a tailor-made traversal function, which is eventually applied in the base case. As an aside, note that `flatten` and `flattenVector` take $O(n \log n)$ time. The running time can be improved to $O(n)$ using a variant of the bottom-up traversal described in Hinze (2000a).

### 5.2 Vector indexing

The particular layout of vector elements was chosen to simplify the definition of the indexing function. Since the elements of the left and the right subtree are intertwined, the least significant bit of the index indicates the subtree in which the indexed element is located. On vectors of type `Tree` the indexing function takes the following form:

```haskell
type Sub t = \forall a. Int \rightarrow t a \rightarrow Maybe a
sub = Sub Tree
sub n (Id a) =
  | n ≡ 0 = Just a
  | otherwise = Nothing
sub n (Fork l r) =
  | n `mod` 2 ≡ 0 = sub (n `div` 2) l
  | otherwise = sub (n `div` 2) r
sub n (Node a l r) =
  | n ≡ 0 = Just a
  | m `mod` 2 ≡ 0 = sub (m `div` 2) l
  | otherwise = sub (m `div` 2) r
where m = n - 1.
```

If the index $n$ is out of range, `sub n v` returns `Nothing`. Otherwise, it yields `Just a` where $a$ is the requested element. Again, it is straightforward to decompose the

---

4 In this case we can turn the rank-2 type signatures into ordinary rank-1 type signatures:

```haskell
flattenFork :: \t. \forall a. (t a \rightarrow [a]) \rightarrow (Fork t a \rightarrow [a])
flattenNode :: \t. \forall a. (t a \rightarrow [a]) \rightarrow (Node t a \rightarrow [a]).
```

This transformation, that is, simplifying $(\forall a. t) \rightarrow (\forall a. u)$ to $\forall a. t \rightarrow u$ works as long as the decomposed function does not rely on polymorphic recursion (Mycroft, 1984).
function into the three cases.

\[
\begin{align*}
\text{subId} &: \text{Sub Id} \\
\text{subId } n \ (\text{Id } a) &= \begin{cases} 
0 & \text{Just } a \\
\text{otherwise} & \text{Nothing}
\end{cases} \\
\text{subFork} &: \forall t. \text{Sub } t \to \text{Sub } (\text{Fork } t) \\
\text{subFork } sub t n \ (\text{Fork } l \ r) &= \begin{cases} 
\text{mod } 2 = 0 & \text{sub } (\text{div } 2) l \\
\text{otherwise} & \text{sub } (\text{div } 2) r
\end{cases} \\
\text{subNode} &: \forall t. \text{Sub } t \to \text{Sub } (\text{Node } t) \\
\text{subNode } sub t n \ (\text{Node } a \ l \ r) &= \begin{cases} 
0 & \text{Just } a \\
\text{otherwise} &= \text{sub } (\text{div } 2) r \\
\text{where } m &= n - 1
\end{cases}
\end{align*}
\]

The 'wrapping code' for vector indexing is very similar to that for vector traversal.

\[
\begin{align*}
\text{subVector} &: \text{Sub Vector} \\
\text{subVector } n \ \text{Empty} &= \text{Nothing} \\
\text{subVector } n \ (\text{NonEmpty } v) &= \text{subVector'} \text{subId } n \ v \\
\text{subVector'} &: \forall t. \text{Sub } t \to \text{SubVector'} t \\
\text{subVector'} \text{subt } n \ (\text{Base } m) &= \text{subt } n \ v \\
\text{subVector'} \text{subt } n \ (\text{Zero } v) &= \text{subVector'} \ (\text{subFork } sub) \ n \ v \\
\text{subVector'} \text{subt } n \ (\text{One } v) &= \text{subVector'} \ (\text{subNode } sub) \ n \ v
\end{align*}
\]

In general, all catamorphisms on Tree can be adapted along these lines.

### 5.3 Matrix indexing

Building upon the functions of the previous section we can quite easily implement matrix indexing. Recall that the type of square matrices differs from the vector type only in the base case where we have \(\text{Base } (t \ (t \ a))\) instead of \(\text{Base } (t \ a)\). Consequently, we only have to change one equation.

\[
\begin{align*}
\text{type } \text{SubMatrix } m &= \forall a. (\text{Int}, \text{Int}) \to m a \to \text{Maybe } a \\
\text{subSquare} &: \text{SubMatrix Square} \\
\text{subSquare } (i, j) \ \text{Empty} &= \text{Nothing} \\
\text{subSquare } (i, j) \ (\text{NonEmpty } m) &= \text{subSquare'} \text{subId } (i, j) \ m \\
\text{subSquare'} &: \forall t. \text{Sub } t \to \text{SubMatrix } (\text{Square'} t) \\
\text{subSquare'} \text{subt } (i, j) \ (\text{Base } m) &= \text{subt } i \ m \gg \text{subt } j \\
\text{subSquare'} \text{subt } (i, j) \ (\text{Zero } m) &= \text{subSquare'} \ (\text{subFork } sub) (i, j) \ m \\
\text{subSquare'} \text{subt } (i, j) \ (\text{One } m) &= \text{subSquare'} \ (\text{subNode } sub) (i, j) \ m
\end{align*}
\]

Note that the call \(\text{subt } n \ v\) has been replaced by \(\text{subt } i \ m \gg \text{subt } j\). A remark is appropriate: the so-called monadic application \(\gg\) takes care of 'out of range'
exceptions. If \( \text{subt} \ i \ m \) yields \( \text{Nothing} \), then \( \text{Nothing} \) is propagated; if it returns \( \text{Just} \ v \), then \( \text{subt} \ j \) is applied to \( v \).

### 5.4 Vector creation

Roughly speaking, a catamorphism is a function that consumes a data structure. Its dual, an anamorphism, produces a data structure. While both classes are equally important, anamorphisms are slightly harder to adapt. To illustrate the additional work involved let us implement an operation that creates a vector of a given length. For simplicity, we assume that the length is given as a bit list with the least significant bit first.

\[
\text{type Make } t = \forall a. a \rightarrow t a
\]

\[
\text{make } :: [\text{Bit}] \rightarrow \text{Make Tree} \quad \text{-- LSB first}
\]

\[
\text{make } [1] a = \text{Id } a
\]

\[
\text{make } (0 : bs) a = \text{Fork (make } bs \ a) (\text{make } a)
\]

\[
\text{make } (1 : bs) a = \text{Node } a (\text{make } bs a) (\text{make } bs a)
\]

The call \( \text{make} \) \( (\text{bits } n) \ a \) creates a vector containing \( n \) copies of \( a \). Now, since \( \text{make} \) performs a case analysis on the bit list, we cannot simply decompose this function. Clearly, the size information must be used while creating the prefix of \( \text{Zero} \) and \( \text{One} \) constructors. Consequently, the size parameter is no longer necessary when the actual tree is constructed, which motivates the following definitions:

\[
\text{makeId } :: \text{Make Id}
\]

\[
\text{makeId } a = \text{Id } a
\]

\[
\text{makeFork } :: \forall t. \text{Make } t \rightarrow \text{Make (Fork } t)
\]

\[
\text{makeFork } \text{maket } a = \text{Fork (maket } a) (\text{maket } a)
\]

\[
\text{makeNode } :: \forall t. \text{Make } t \rightarrow \text{Make (Node } t)
\]

\[
\text{makeNode } \text{maket } a = \text{Node } a (\text{maket } a) (\text{maket } a).
\]

The case analysis \( \text{makeVector}^\prime \) performs is similar to that of \( \text{make} \) except that we now assume that the bit list has the most significant bit first. This change of order is necessary since the creation function \( \text{maket} \) is constructed bottom-up.

\[
\text{makeVector } :: [\text{Bit}] \rightarrow \text{Make Vector} \quad \text{-- MSB first}
\]

\[
\text{makeVector } [] a = \text{Empty}
\]

\[
\text{makeVector } (1 : bs) a = \text{NonEmpty (makeVector}^\prime bs \text{maket } a)
\]

\[
\text{makeVector}^\prime :: [\text{Bit}] \rightarrow \forall t. \text{Make } t \rightarrow \text{Make (Vector}^\prime t)
\]

\[
\text{makeVector}^\prime [] \text{maket } a = \text{Base (maket } a)
\]

\[
\text{makeVector}^\prime (0 : bs) \text{maket } a = \text{Zero (makeVector}^\prime bs (\text{makeFork } \text{maket } a))
\]

\[
\text{makeVector}^\prime (1 : bs) \text{maket } a = \text{One (makeVector}^\prime bs (\text{makeNode } \text{maket } a))
\]

In general, implementing anamorphisms is more involved since top-down algorithms must be adapted to the bottom-up representation.
6 Related and future work

This work is inspired by a recent paper of Okasaki (1999), who derives representations of square matrices from exponentiation algorithms. He shows, in particular, that the tail-recursive version of the fast exponentiation algorithm 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 significant to most significant 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 the textbook by Okasaki (1998). 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 are able to capture structural invariants in a more straightforward way have been described by Zenger (1997; 1998) and Xi (1999). Using the indexed types of Zenger one can, for instance, parameterize vectors and matrices by their size. Size compatibility is then statically ensured by the type checker. Xi achieves the same effect using dependent datatypes. In his system, de Caml, the type of perfect leaf trees, for instance, is declared as follows:

```
datatype 'a perfect with nat = Leaf (0) of 'a | {n : nat} Fork (n + 1) of 'a perfect (n) * 'a perfect (n).
```

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.

A direction for future work suggests itself. It remains to investigate the expressiveness of the framework and of higher-order kinded types in general. Which class of multisets can be described using higher-order recursion equations? It appears that recursion equations of order \( n + 1 \) are more expressive than recursion equations of order \( n \). Observe, for instance, that one cannot define the multiset of square numbers using zeroth-order equations, where the unknowns range only over multisets. Finally, are there data structures of practical interest that cannot be captured by second-order kinded types?

Acknowledgements

I am grateful to Iliano Cervesato, Lambert Meertens, John O'Donnell, Chris Okasaki, and two anonymous referees for many valuable comments.
A Proofs

A.1 Proof of Theorem 1

Recall the definitions of $X$ and $f$, 

$$
X = A \uplus \biguplus_{i \in N} h_i X \\
Y = f Y = Y \uplus \biguplus_{i \in N} f(h_i Y),
$$

where $N$ is some finite index set, $A$ is an arbitrary multiset, and the $h_i$ are homomorphisms, that is, they satisfy $h_i \emptyset = \emptyset$ and $h_i (A \uplus B) = h_i A \uplus h_i B$. We show $X = f A$ by fixpoint induction. To this end define $\mathcal{X}$ and $\mathcal{F}$

$$
\mathcal{X} X = A \uplus \biguplus_{i \in N} h_i X \\
\mathcal{F} f Y = Y \uplus \biguplus_{i \in N} f(h_i Y)
$$

and $\mathcal{X}_n$ and $\mathcal{F}_n$

$$
\begin{align*}
\mathcal{X}_0 &= \emptyset \\
\mathcal{X}_{n+1} &= \mathcal{X}_n \\
\mathcal{F}_0 &= \lambda Y \rightarrow \emptyset \\
\mathcal{F}_{n+1} &= \mathcal{F}_n
\end{align*}
$$

Now, in order to prove $\bigsqcup_{n \geq 0} \mathcal{X}_n = (\bigsqcup_{n \geq 0} \mathcal{F}_n) A$ we establish

$$
\forall n \in \mathbb{N}. \mathcal{X}_n = \mathcal{F}_n A
$$

by induction over the natural numbers.

- **Case $n = 0$:**

  $$
  \begin{align*}
  \mathcal{X}_0 &= \emptyset \\
  &= \{ \text{ definition } \mathcal{X}_0 \} \\
  &= \emptyset
  \\
  \mathcal{F}_0 &= \lambda Y \rightarrow \emptyset \\
  &= \{ \text{ definition } \mathcal{F}_0 \} \\
  &= \mathcal{F}_0 A.
  \end{align*}
  $$

- **Case $n = m + 1$:**

  $$
  \begin{align*}
  \mathcal{X}_{m+1} &= \{ \text{ definition } \mathcal{X}_{m+1} \text{ and } \mathcal{X} \} \\
  &= A \uplus \biguplus_{i \in N} h_i \mathcal{X}_m \\
  &= \{ \text{ ex hypothesi } \} \\
  &= A \uplus \biguplus_{i \in N} h_i (\mathcal{F}_m A) \\
  &= \{ \text{ proof obligation, see below } \} \\
  &= A \uplus \biguplus_{i \in N} \mathcal{F}_m (h_i A) \\
  &= \{ \text{ definition } \mathcal{F}_{m+1} \text{ and } \mathcal{F} \} \\
  &= \mathcal{F}_{m+1} A.
  \end{align*}
  $$

https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
It remains to show

$$\forall n \in \mathbb{N}. \forall Y. \biguplus_{i \in N} h_i (\mathcal{F}_n Y) = \biguplus_{i \in N} \mathcal{F}_n (h_i Y).$$

Again we proceed by simple induction.

- **Case** $n = 0$:

$$\biguplus_{i \in N} h_i (\mathcal{F}_0 Y)$$

$$= \{ \text{definition } \mathcal{F}_0 \}$$

$$\biguplus_{i \in N} \emptyset$$

$$= \{ \text{the } h_i \text{ are homomorphisms} \}$$

$$\biguplus_{i \in N} \emptyset$$

$$= \{ \text{definition } \mathcal{F}_0 \}$$

$$\biguplus_{i \in N} \mathcal{F}_0 (h_i Y).$$

- **Case** $n = m + 1$: the proof makes essential use of the following two properties of the multiset union:

$$\biguplus_{i \in I} (A_i \uplus B_i) = \biguplus_{i \in I} A_i \uplus \biguplus_{i \in I} B_i \quad \biguplus_{i \in I} \biguplus_{j \in J} A_{i,j} = \biguplus_{j \in J} \biguplus_{i \in I} A_{i,j}.$$

Note that the *associativity law* on the left is a special case of the second law called *interchanging the order of union*. Now we reason as follows:

$$\biguplus_{i \in N} h_i (\mathcal{F}_{m+1} Y)$$

$$= \{ \text{definition } \mathcal{F}_{m+1} \text{ and } \mathcal{F} \}$$

$$\biguplus_{i \in N} h_i (Y \uplus \biguplus_{k \in N} \mathcal{F}_m (h_k Y))$$

$$= \{ \text{the } h_i \text{ are homomorphisms} \}$$

$$\biguplus_{i \in N} (h_i Y \uplus \biguplus_{k \in N} h_i (\mathcal{F}_m (h_k Y))))$$

$$= \{ \text{associativity} \}$$

$$\biguplus_{i \in N} h_i Y \uplus \biguplus_{j \in N} \biguplus_{k \in N} h_j (\mathcal{F}_m (h_k Y))$$

$$= \{ \text{interchanging the order of union} \}$$

$$\biguplus_{i \in N} h_i Y \uplus \biguplus_{k \in N} \biguplus_{j \in N} h_j (\mathcal{F}_m (h_k Y))$$

$$= \{ \text{ex hypothesi} \}$$

$$\biguplus_{i \in N} h_i Y \uplus \biguplus_{k \in N} \mathcal{F}_m (h_j (h_k Y))$$

$$= \{ \text{associativity} \}$$

https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
Manufacturing datatypes

\[
\bigcup_{i \in \mathbb{N}} (h_i Y \uplus \bigcup_{j \in \mathbb{N}} \mathcal{F}_m (h_j (h_i Y))) = \{ \text{definition } \mathcal{F}_{m+1} \text{ and } \mathcal{F} \} \bigcup_{i \in \mathbb{N}} \mathcal{F}_{m+1} (h_i Y).
\]

A.2 Proof of Theorem 2

The theorem and its proof rely heavily on the framework of polytypic programming. The basic idea of polytypism is briefly sketched below. For a more thorough treatment the interested reader is referred to Hinze (2000b; 2000c).

The Haskell Prelude defines the function \( \text{sum} \), which computes the sum of a finite list of numbers. Summing up a container of numbers makes, of course, sense for arbitrary functors, of which \( \text{List} \) is only a very popular instance. Unfortunately, in Haskell the programmer must define \( \text{sum} \) for each datatype from scratch, typically, by induction on the structure of values. Using a polytypic definition \( \text{sum} \) can be defined once and for all times. The basic idea is to recurse on the structure of types.

Writing the type argument in angle brackets we may define

\[
\text{sum}(F) :: F \text{ Nat} \to \text{Nat}
\]

\[
\text{sum}(K \text{ Unit}) \ x = 0
\]

\[
\text{sum}(\text{Id}) \ x = x
\]

\[
\text{sum}(F_1 \mid F_2) \ x = \text{case } x \text{ of } \{ \text{Left } x_1 \to \text{sum}(F_1) \ x_1; \text{Right } x_2 \to \text{sum}(F_2) \ x_2 \}
\]

\[
\text{sum}(F_1 \times F_2) \ x = \text{sum}(F_1) (\text{fst } x) + \text{sum}(F_2) (\text{snd } x).
\]

A polytypic function is uniquely defined by giving cases for the constant functor \( K \) \( \text{Unit} \), for the identity functor, for sums, and for products. The case for functor composition can be automatically derived from this information. Furthermore, a polytypic function must be strict with respect to its type argument, for example, \( \text{sum}(K \text{ Void}) = \bot \) where \( K \text{ Void} \) is the ‘bottom functor’ and \( \bot \) is the bottom element of \( \text{Void} \to \text{Nat} \).

The second polytypic function we require is \( \text{fan}(F) \), which generates the multiset of all structures of type \( F \ a \) from a given seed of type \( a \).

\[
\text{fan}(F) :: \forall a. a \to \{ F \ a \}
\]

\[
\text{fan}(K \text{ Unit}) \ x = \{()\}
\]

\[
\text{fan}(\text{Id}) \ x = \{x\}
\]

\[
\text{fan}(F_1 \mid F_2) \ x = \{\text{Left } x_1 | x_1 \leftarrow \text{fan}(F_1) \ x_1 \} \uplus \{\text{Right } x_2 | x_2 \leftarrow \text{fan}(F_2) \ x_2\}
\]

\[
\text{fan}(F_1 \times F_2) \ x = \{\{x_1, x_2 | x_1 \leftarrow \text{fan}(F_1) \ x_1; x_2 \leftarrow \text{fan}(F_2) \ x_2\}
\]

Note that \( \text{fan}(K \text{ Void}) = \lambda x \to \emptyset \) by strictness (\( \emptyset \) is the least element of \( \{ F \ a \} \)).

Finally, we define

\[
\text{Size}(F) :: \{\text{Nat}\}
\]

\[
\text{Size}(F) = \{\text{sum}(F) \ a | a \leftarrow \text{fan}(F) \ 1\},
\]

which determines the multiset of all container sizes of \( F \). Given this definition we must prove that if the functor \( F \) corresponds to the multiset \( M \) and if \( M \)’s definition
only involves admissible products, then \( M = \text{Size}(F) \). The following equations can be shown using straightforward calculations:

\[
\begin{align*}
\text{Size}(K \text{ Void}) &= \emptyset \quad \text{(A 1)} \\
\text{Size}(K \text{ Unit}) &= \{0\} \quad \text{(A 2)} \\
\text{Size}(Id) &= \{1\} \quad \text{(A 3)} \\
\text{Size}(F_1 \mid F_2) &= \text{Size}(F_1) \uplus \text{Size}(F_2) \quad \text{(A 4)} \\
\text{Size}(F_1 \times F_2) &= \text{Size}(F_1) + \text{Size}(F_2). \quad \text{(A 5)}
\end{align*}
\]

It remains to establish

\[
\text{Size}(F \cdot G) = \text{Size}(F) \ast \text{Size}(G) \quad \text{if Size}(G) \text{ is simple.} \quad \text{(A 6)}
\]

We proceed by fixpoint induction over the structure of \( F \)—this proof principle is detailed in Hinze (2000c).

- **Case** \( F = K \text{ Void} \):

\[
\begin{align*}
\text{Size}(K \text{ Void} \cdot G) &= \{ K \text{ Void} \cdot F = K \text{ Void } \} \\
\text{Size}(K \text{ Void}) &= \{ \text{Eq. (A 1)} \} \\
\emptyset &= \{ \emptyset \ast A = \emptyset \} \\
\emptyset \ast \text{Size}(G) &= \{ \text{Eq. (A 1)} \} \\
\text{Size}(K \text{ Void}) \ast \text{Size}(G).
\end{align*}
\]

- **Case** \( F = K \text{ Unit} \):

\[
\begin{align*}
\text{Size}(K \text{ Unit} \cdot G) &= \{ K \text{ Unit} \cdot F = K \text{ Unit } \} \\
\text{Size}(K \text{ Unit}) &= \{ \text{Eq. (A 2)} \} \\
\{0\} &= \{ 0 \ast a = 0 \ast a \text{ and Size}(G) \text{ is simple } \} \\
\{0\} \ast \text{Size}(G) &= \{ \text{Eq. (A 2)} \} \\
\text{Size}(K \text{ Unit}) \ast \text{Size}(G).
\end{align*}
\]

- **Case** \( F = Id \):

\[
\begin{align*}
\text{Size}(Id \cdot G) &= \{ Id \cdot F = F \} \\
\text{Size}(G)
\end{align*}
\]
Manufacturing datatypes

\[ \{ \{1\} \ast A = A \} \]

\[ \{1\} \ast \text{Size}(G) \]

\[ \{ \text{Eq. (A 3)} \} \]

\[ \text{Size}(\text{Id}) \ast \text{Size}(G). \]

- **Case** \( F = F_1 \mid F_2 \):

\[ \text{Size}(\langle F_1 \mid F_2 \rangle) \cdot G \]

\[ = \{ (F \mid G) \cdot H = F \cdot H \mid G \cdot H \} \]

\[ \text{Size}(F_1 \cdot G \mid F_2 \cdot G) \]

\[ = \{ \text{Eq. (A 4)} \} \]

\[ \text{Size}(F_1 \cdot G) \cup \text{Size}(F_2 \cdot G) \]

\[ = \{ \text{ex hypothesi} \} \]

\[ \text{Size}(F_1) \ast \text{Size}(G) \cup \text{Size}(F_2) \ast \text{Size}(G) \]

\[ = \{ (A \cup B) \ast C = A \ast C \cup B \ast C \} \]

\[ (\text{Size}(F_1) \cup \text{Size}(F_2)) \ast \text{Size}(G) \]

\[ = \{ \text{Eq. (A 4)} \} \]

\[ \text{Size}(F_1 \mid F_2) \ast \text{Size}(G). \]

- **Case** \( F = F_1 \times F_2 \): analogous using \((F \times G) \cdot H = F \cdot H \times G \cdot H\) and \((A + B) \ast c = A \ast c + B \ast c\).

Note that the cases \( F = \text{K Unit} \) and \( F = F_1 \times F_2 \) require the multiset \( \text{Size}(G) \) to be simple. Finally, Theorem 2 follows by a straightforward fixpoint induction using Eq. (A 1)–(A 6).

References


https://doi.org/10.1017/S095679680100404X Published online by Cambridge University Press
R. Hinze


