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. 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.

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 $1 < i, j \leqslant n$. Clearly, to represent a Toeplitz matrix of size $(n+1) * (n+1)$ 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 (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 which contains one occurrence of each odd number.

$$odd \quad = \quad \{\!|1|\!\} \uplus \{\!|2|\!\} + odd$$

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

$$A + B \quad = \quad \{\!| a + b \mid a \leftarrow A; b \leftarrow B |\!\}.$$

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 *type constructor*, as a *container type* (Hoogendijk & de Moor, 2000), or as a *functor*[1]. An element of a parameterized type is called a *container*. Now, 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 sums and products lifted to functors, that is, $(F_1 \mid F_2)\ a = F_1\ a \mid F_2\ a$ and $(F_1 \times F_2)\ a = F_1\ a \times F_2\ a$. Comparing the two equations we see that $\{\!|1|\!\}$ corresponds to Id, the operation $(\uplus)$ corresponds to $(|)$, and $(+)$ corresponds to $(\times)$. This immediately implies that $Id \times Id$ 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, $Id \times Id$ corresponds to $\{\!|1|\!\} + \{\!|1|\!\} = \{\!|2|\!\}$ as it contains one container of size 2; the functor $Id \mid Id \times Id$ corresponds to $\{\!|1|\!\} \uplus \{\!|1|\!\} + \{\!|1|\!\} = \{\!|1, 2|\!\}$ 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. For concreteness, examples are given in the functional language Haskell 98 (Peyton Jones & Hughes, 1999).

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

The left upper corner of a Toeplitz matrix is represented by $Corner\ a$; 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 r c m we extend 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

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

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

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

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.

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

Note that we use the identifier *Toeplitz* both as a type and as a value constructor. The 4×4 Toeplitz matrix (a_{ij}) is now represented by

$$\mathit{Toeplitz}\ a_{11}\ (\mathit{Cons2}\ a_{21}\ a_{12}\ (\mathit{Cons2}\ a_{31}\ a_{13}\ (\mathit{Cons2}\ a_{41}\ a_{14}\ \mathit{Nil2}))).$$

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.

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

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

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

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 $K\ \mathit{Unit}$ corresponds to $\wr 0 \wr$. Unsurprisingly, *Nat* models the ubiquitous datatype of polymorphic lists.

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

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

$$\mathit{Toeplitz}\ a_{11}\ (\mathit{Cons}\ (a_{21}, a_{12})\ (\mathit{Cons}\ (a_{31}, a_{13})\ (\mathit{Cons}\ (a_{41}, a_{14})\ \mathit{Nil}))).$$

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.

$$power \quad = \quad \{\!|1|\!\} \uplus power * \{\!|2|\!\}$$

The corresponding functor equation is

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

from which we can easily derive the following datatype definition:

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

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 of the introduction on page 1 proceeds *top-down*. The type *Perfect* is an example of a so-called *nested datatype* (Bird & Meertens, 1998): the recursive call of *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 *Perfect Int* (\$\$ always refers to the expression in the preceding row and I abbreviates *Int*).

$$\begin{array}{rcl}
(((1,2),(3,4)),((5,6),(7,8))) & :: & (((I,I),(I,I)),((I,I),(I,I))) \\
Zero\ \$\$ & :: & Perfect\ (((I,I),(I,I)),((I,I),(I,I))) \\
Succ\ \$\$ & :: & Perfect\ ((I,I),(I,I)) \\
Succ\ \$\$ & :: & Perfect\ (I,I) \\
Succ\ \$\$ & :: & Perfect\ I
\end{array}$$

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 *Zero* the nested pair is turned into a leaf. Now, each application of *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 *Succ* and *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 $square = nat * nat$. 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.

$$\begin{array}{lcl}
nat & = & nat'\ \{\!|0|\!\} \\
nat'\ n & = & n \uplus nat'\ (\{\!|1|\!\} + n)
\end{array}$$

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.

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

To see how the multiset of square numbers is constructed, let us unfold the definition of *square* a few steps:

$$\begin{array}{lcl} square'\ \wr 0 \wr & = & \wr 0 \wr \uplus square'\ \wr 1 \wr \\ & = & \wr 0 \wr \uplus (\wr 1 \wr \uplus square'\ \wr 2 \wr) \\ & = & \wr 0 \wr \uplus (\wr 1 \wr \uplus (\wr 4 \wr \uplus square'\ \wr 3 \wr)) \\ & = & \ldots\ . \end{array}$$

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 *square* and *square'*. From the functor equations

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

we can derive the following datatype declarations:

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

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

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

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. The following table shows the construction of $Succ^3\ (Zero\ m)$ in more detail ($f^n\ a$ means f applied n times to a):

$$\begin{array}{rcl} m & :: & Cons^3\ Nil\ (Cons^3\ Nil\ Int) \\ Zero\,\$\$ & :: & Square'\ (Cons^3\ Nil)\ Int \\ Succ\,\$\$ & :: & Square'\ (Cons^2\ Nil)\ Int \\ Succ\,\$\$ & :: & Square'\ (Cons\ Nil)\ Int \\ Succ\,\$\$ & :: & Square'\ Nil\ Int = Square\ Int. \end{array}$$

Roughly speaking, the outer applications of the value constructor *Cons* make sure that the inner lists have the same length and *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 $\wr T \wr$ 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 $M\ x$ is the number of occurrences of x in M. In order to give a meaning to recursion equations such as $M = \wr 1 \wr \uplus M$, we also allow elements to occur infinitely often. Thus, the set of multisets over T is given by $\wr T \wr = T \to \mathbb{N}_\infty$ where $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$. On $\wr T \wr$ we can define the usual pointwise order $M \mathrel{\dot\leqslant} N \iff \forall t \in T.\ M\ t \leqslant N\ t$ where $(\leqslant)$ is the standard order on the naturals additionally setting $n \leqslant \infty$. Then $(\wr T \wr, \mathrel{\dot\leqslant})$ forms a complete partial order.

In this article, 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, $\wr n \wr$ with $n \in \{0, 1\}$ denotes the singleton multiset that contains n exactly once, $(\uplus)$ denotes multiset union, $(+)$ and $(*)$ are addition and multiplication lifted to multisets: $A \otimes B = \wr\, a \otimes b \mid a \leftarrow A; b \leftarrow B \wr$ for $\otimes \in \{+, *\}$. Note that the operations $(\uplus) :: \wr T \wr \to \wr T \wr \to \wr T \wr$ and $(+), (*) :: \wr Nat \wr \to \wr Nat \wr \to \wr Nat \wr$ are continuous with respect to '$\mathrel{\dot\leqslant}$'. If the meaning can be resolved from the context, we abbreviate $\wr n \wr$ by n. Furthermore, we assume that

$$
\begin{array}{rcl}
\wr m \wr + \wr n \wr & = & \wr m + n \wr \\
\wr m \wr * \wr n \wr & = & \wr m * n \wr
\end{array}
\qquad
\begin{array}{rcl}
A \uplus (B \uplus C) & = & (A \uplus B) \uplus C \\
A \uplus B & = & B \uplus A
\end{array}
$$

$$
\begin{array}{rcl}
A + (B + C) & = & (A + B) + C \\
A + B & = & B + A
\end{array}
\qquad
\begin{array}{rcl}
A * (B * C) & = & (A * B) * C \\
a * b & = & b * a
\end{array}
$$

$$
\begin{array}{rcl}
0 + A & = & A \\
0 * a & = & 0
\end{array}
\qquad
\begin{array}{rcl}
1 * A & = & A \\
A * 1 & = & A
\end{array}
$$

$$
\begin{array}{rcl}
\varnothing \uplus A & = & A \\
\varnothing + A & = & \varnothing \\
\varnothing * A & = & \varnothing
\end{array}
\qquad
\begin{array}{rcl}
(A \uplus B) + C & = & A + C \uplus B + C \\
(A \uplus B) * C & = & A * C \uplus B * C \\
(A + B) * c & = & A * c + B * c
\end{array}
$$

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

Fig. 1. Laws of the operations.

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:

$$
\begin{array}{rcl}
M_1 & = & \wr 1 \wr \uplus M_1 \\
M_2 & = & \wr 1 \wr + M_2 \\
M_3 & = & \wr 1 \wr * M_3.
\end{array}
$$

The first equation defines the multiset that contains 1 infinitely often. Since we take the least fixpoint, M_2 and M_3 both define the empty multiset.

A multiset is called *simple* iff it contains exactly one element. Simple multisets 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 restrict 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 both F and G only comprise 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 $h :: \{\!|\, T \,|\!\} \to \{\!|\, T \,|\!\}$ 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 A be a multiset, let $h_1, \ldots, h_n$ be homomorphisms, 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 $A = \{\!|\, 0 \,|\!\}$, $n = 1$, and $h_1\ N = \{\!|\, 1 \,|\!\} + N$. Now, the *tail-recursive variant* of X is $f\ A$ with f given by

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

The definition of f is called *tail-recursive*—think of $(\uplus)$ as a conditional. Note that $nat'\ \{\!|\, 0 \,|\!\}$ is the tail-recursive variant of *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 $f :: \{\!|\, T \,|\!\} \to \{\!|\, T \,|\!\}$ 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:

$$F \quad ::= \quad K\ \mathit{Void} \mid K\ \mathit{Unit} \mid \mathit{Id} \mid (F \mid F) \mid (F \times F) \mid (F \cdot F).$$

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 $\mathit{Id}\ a = a$ 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: $(F_1 \mid F_2)\ a = F_1\ a \mid F_2\ a$ 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:[2]

```
data Void
type Unit          = ()
data Either a1 a2  = Left a1 | Right a2
data (a1, a2)      = (a1, a2).
```

[2] Note that *Void* was defined in Haskell 1.4; it is, however, no longer part of Haskell 98.

The predefined types $Either\ a_1\ a_2$ and (a_1, a_2) implement 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*.

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. We could make this assumption explicit using strictness annotations

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

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.

M_1	M_2	$\varnothing$	$\{\!\{0\}\!\}$	$\{\!\{1\}\!\}$	$M_1 \uplus M_2$	$M_1 + M_2$	$M_1 * M_2$
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 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 $sum\langle F\rangle :: F\ Nat \to Nat$, which sums a structure of natural numbers. To make the relationship between multisets and functors precise we furthermore require the function $fan\langle F\rangle :: a \to \{\!\{F\ a\}\!\}$, 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 finite lists that contain the natural number 1 as the single element, that is, $fan\langle List\rangle\ 1 = \{\!\{Nil, Cons\ 1\ Nil, Cons\ 1\ (Cons\ 1\ Nil), Cons\ 1\ (Cons\ 1\ (Cons\ 1\ Nil)), \ldots\}\!\}$.

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 $\wr 1, 3 \wr \uplus (\wr 1, 3 \wr + \wr 1, 3 \wr) = \wr 1, 2, 3, 4, 4, 6 \wr$. In general, functor composition corresponds to the non-commutative multiset operation $(\circledast)$ given by

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

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 $(*)$ and $(\circledast)$ coincide only for admissible products, that is, if B contains only one container so that $b_1 = \cdots = b_a$.

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 a 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:

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

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:

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

Using this representation the vector $(0,1,2,3,4,5)$ is written somewhat lengthily 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 $\mathit{Vector'}$ is a type constructor of kind $(* \to *) \to (* \to *)$. In this case the 'higher-orderness' is, however, not required. Noting that the first argument of $\mathit{Vector'}$ is always applied to the second, we can transform $\mathit{Vector'}$ 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 $\mathit{Vector'}$ 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 $\mathit{Square'}$ defined in the introduction is not amenable to this transformation since the first argument of $\mathit{Square'}$ 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.

$$nat_2 \quad = \quad 0 \uplus nat_2 * 2 \uplus 1 + nat_2 * 2$$

Transforming the corresponding functor equation

$$Nat_2 \quad = \quad K\ \mathit{Unit} \mid Nat_2 \cdot (Id \times Id) \mid Id \times Nat_2 \cdot (Id \times Id)$$

into a Haskell datatype yields

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

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

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

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) on page 18. Note that the representation is not unique because of trailing zeros: the empty sequence, for example, can be represented by *End*, *Zero End*, *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:

$$\begin{aligned} nat_3 &= 0 \uplus pos_3 \\ pos_3 &= 1 \uplus pos_3 * 2 \uplus 1 + pos_3 * 2. \end{aligned}$$

More elegantly, one can define a *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*.

$$nat_4 = 0 \uplus 1 + nat_4 * 2 \uplus 2 + nat_4 * 2$$

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 nat_3 into a tail-recursive form.

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

The corresponding functor equations look puzzling.

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

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

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

These definitions assume that t is a simple functor, that is, a functor which contains exactly one container. 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

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

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

$$\begin{aligned}&\mathit{NonEmpty}\ (\mathit{One}\ (\mathit{Zero}\ (\mathit{Base}\ (\\ &\quad \mathit{Fork}\ (\mathit{Node}\ 0\ (\mathit{Id}\ 1)\ (\mathit{Id}\ 2))\ (\mathit{Node}\ 3\ (\mathit{Id}\ 4)\ (\mathit{Id}\ 5))))))).\end{aligned}$$

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 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 $\mathit{Fork}\ l\ r$ represents the vector $(b_1, c_1, \ldots, b_n, c_n)$ and $\mathit{Node}\ a\ l\ r$ 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 always for vector types we can 'firstify' the type definitions.

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

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 by repeatedly shifting a one:

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

[3] I have learned, however, that Hongwei Xi has independently discovered the same data structure.

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

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

that is, $nat'_6\ p$ 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

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

which assume that f is simple, into Haskell datatypes yields

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

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

$$\begin{array}{l} Zero\ (One\ (Fork\ (Id\ 0)\ (Id\ 1)) \\ \quad (One\ (Fork\ (Fork\ (Id\ 2)\ (Id\ 3))\ (Fork\ (Id\ 4)\ (Id\ 6)))\ End)). \end{array}$$

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.

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

This datatype implements *rightist right-perfect trees* or *RR-trees* (Dielissen & Kaldewaij, 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 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:

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

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

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

we can derive the following datatype definitions:

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

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:

$$Two\ (Two\ (Base\ (Bin\ (Bin\ (Id\ 0)\ 1\ (Id\ 2))\ 3\ (Bin\ (Id\ 4)\ 5\ (K\ ())))))).$$

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:

$$\begin{array}{lcl} heap_1 & = & heap_1'\ 0\ \varnothing\ 0 \\ heap_1'\ H\ p\ p' & = & H \uplus heap_1'\ (1+H+p \uplus 1+p'+H)\ p'\ (1+p'*2). \end{array}$$

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 $heap_1'\ [a+1,\ldots,a']\ a\ a'$ 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

$$\begin{array}{lcl} Heap_1 & = & Heap_1'\ (K\ Unit)\ (K\ Void)\ (K\ Unit) \\ Heap_1'\ h\ f\ f' & = & h \mid Heap_1'\ (Id \times h \times f \mid Id \times f' \times h)\ f'\ (Id \times f' \cdot (Id \times Id)) \end{array}$$

into Haskell datatypes yields

$$\begin{array}{lcl} \textbf{type}\ Heap & = & Heap'\ (K\ Unit)\ (K\ Void)\ (K\ Unit) \\ \textbf{data}\ Heap'\ h\ t\ t'\ a & = & Zero\ (h\ a) \\ & \mid & Succ\ (Heap'\ (Sum\ (Bin\ h\ t)\ (Bin\ t'\ h))\ t'\ (Node\ t')\ a). \end{array}$$

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

$$\begin{array}{l} Succ\ (Succ\ (Succ\ (Zero\ (right\ (Bin\ (Node\ 0\ (Node\ 1\ (K\ ())\ (K\ ()))\ (Node\ 2\ (K\ ())\ (K\ ())))\ 3\ (left\ (Bin\ (right\ (Bin\ (K\ ())\ 4\ (K\ ())))\ 5\ (K\ ()))))))))) \end{array}$$

Here, *left* a abbreviates $Sum\ (Left\ a)$ and *right* a abbreviates $Sum\ (Right\ a)$. Note that the term encodes both the height and the size of the heap. The prefix of *Succ* and *Zero* constructors represents the height. The sequence of *left* and *right* constructors encodes the size: if we replace *left* by 0, *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 $\lambda h \to heap'_1\ h\ p\ p'$ 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

$$\begin{array}{lcl} heap_2 & = & heap'_2\ 0\ 0\ 1 \\ heap'_2\ h\ p\ p' & = & h \uplus heap'_2\ (1+h+p)\ p'\ (1+p'*2) \\ & & \quad \uplus\ heap'_2\ (1+p'+h)\ p'\ (1+p'*2). \end{array}$$

Here are the corresponding functor equations

$$\begin{array}{lcl} Heap_2 & = & Heap'_2\ (K\ Unit)\ (K\ Unit)\ Id \\ Heap'_2\ h\ f\ f' & = & h \mid Heap'_2\ (Id \times h \times f)\ f'\ (Id \times f' \cdot (Id \times Id)) \\ & & \quad \mid Heap'_2\ (Id \times f' \times h)\ f'\ (Id \times f' \cdot (Id \times Id)) \end{array}$$

and finally the Haskell datatype declarations, which assume that t' is simple.

$$\begin{array}{lcl} \textbf{type}\ Heap & = & Heap'\ (K\ Unit)\ (K\ Unit)\ Id \\ \textbf{data}\ Heap'\ h\ t\ t'\ a & = & Base\ (h\ a) \\ & \mid & One\ (Heap'\ (Bin\ h\ t)\ t'\ (Node\ t')\ a) \\ & \mid & Two\ (Heap'\ (Bin\ t'\ h)\ t'\ (Node\ t')\ a) \end{array}$$

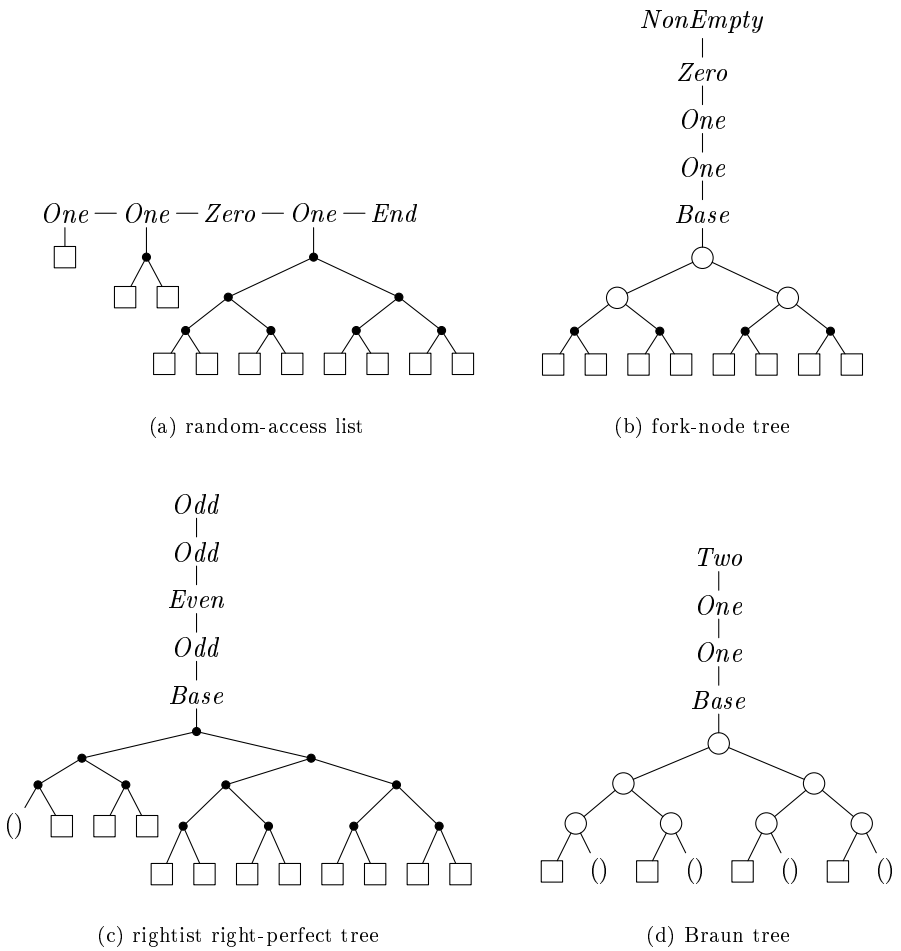
Perhaps surprisingly, the transformation yields a representation that is based on the 1-2 number system (least significant bit first). For background information the interested reader is referred to Hinze (1999), which explains the relationship between heaps and different number systems in more detail.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

$$Two\ (Two\ (Base\ (Bin\ (Node\ 0\ (Id\ 1)\ (Id\ 2))\ 3\ (Bin\ (Id\ 4)\ 5\ (K\ ())))))$$

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), or 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



(a) random-access list (b) fork-node tree

(c) rightist right-perfect tree (d) Braun tree

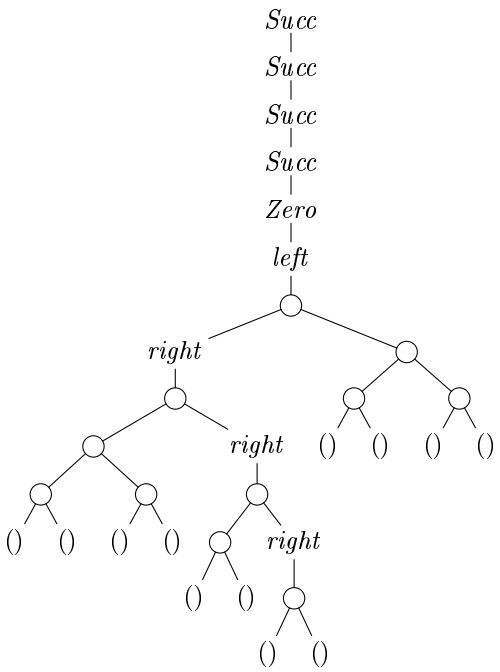
Fig. 2. Different representations of a vector with 11 elements. Note that '□' represents a leaf (an element of Id), '•' an unlabelled node (an element of $Id \times Id$, $Fork\ t$, or $Prod\ t_1\ t_2$), and '○' a labelled node (an element of $Node\ t$ or $Bin\ t_1\ t_2$).

similarity suggests modelling 2-3 trees as follows:

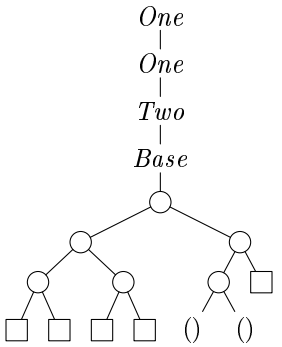
$$\begin{aligned} \mathit{tree23} &= \mathit{tree23'}\ 0 \\ \mathit{tree23'}\ N &= N \uplus \mathit{tree23'}\ (N + 1 + N \uplus N + 1 + N + 1 + N). \end{aligned}$$

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

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



(a) left-complete tree ($Heap_1$)



(b) left-complete tree ($Heap_2$)

Fig. 3. Different representations of a vector with 11 elements (continued).

give rise to the following datatype definitions:

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

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

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

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)

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

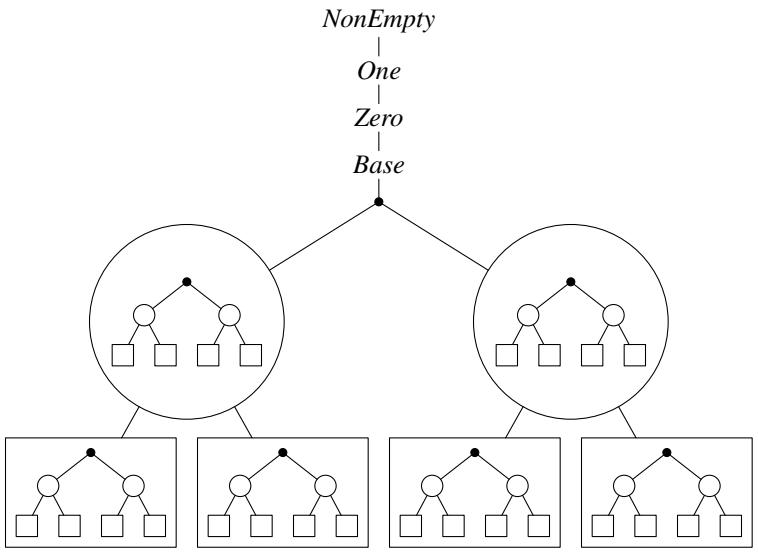


Fig. 4. The representation of a 6×6 matrix based on fork-node trees.

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

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

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 implementation of square matrices based on fork-node trees.

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

The representation of a 6×6 matrix is shown in Figure 4.

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

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

Alternatively, one may use the following scheme:

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

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 Vector and matrix operations

Many of the Haskell datatype declarations we have seen so far are unusual, some are 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* $map\langle F\rangle :: \forall a\ b.\ (a \to b) \to (F\ a \to F\ b)$, 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: $map\langle F\rangle\ (times\ c)$ where $times\ a\ b = a * b$. A related function is the *zipping function* $zip\langle F\rangle :: \forall a\ b\ c.\ (a \to b \to c) \to (F\ a \to F\ b \to F\ c)$, 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 *show* function), and parsers (such as Haskell's *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

$$\textbf{data}\ \mathit{Tree}\ a \quad = \quad \mathit{Id}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

fails to meet the requirements. Interestingly though, both definitions are closely related. Consider the definitions of the subsidiary types that are used in *Vector* and *Vector′*.

$$\begin{aligned}
&\textbf{newtype}\ \mathit{Id}\ a &&= \quad \mathit{Id}\ a\\
&\textbf{data}\ \mathit{Fork}\ \mathit{tree}\ a &&= \quad \mathit{Fork}\ (\mathit{tree}\ a)\ (\mathit{tree}\ a)\\
&\textbf{data}\ \mathit{Node}\ \mathit{tree}\ a &&= \quad \mathit{Node}\ a\ (\mathit{tree}\ a)\ (\mathit{tree}\ a)
\end{aligned}$$

These types can be seen as decompositions of the regular type constructor *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 *Vector* and *Vector′* result from an application of Theorem 1: *Id* corresponds to the multiset A and *Fork* and *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 *Tree*. This is usually the creative part. Then we decompose this function into a base and two recursive cases mirroring the decomposition

of *Tree* into *Id*, *Fork*, and *Node*. Finally, we write 'wrapping code' for *Vector* and *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

$$\textbf{type}\ \mathit{Flatten}\ t \quad = \quad \forall a.\ t\ a \to [a].$$

Note that *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 *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 *Fork* l r represents the vector $(b_1, c_1, \ldots, b_n, c_n)$ and *Node* a l r represents $(a, b_1, c_1, \ldots, b_n, c_n)$. Transliterating this description into Haskell yields the desired traversal function:

$$\begin{array}{lcl}
\mathit{flatten} & :: & \mathit{Flatten}\ \mathit{Tree} \\
\mathit{flatten}\ (\mathit{Id}\ a) & = & [a] \\
\mathit{flatten}\ (\mathit{Fork}\ l\ r) & = & \mathit{interleave}\ (\mathit{flatten}\ l)\ (\mathit{flatten}\ r) \\
\mathit{flatten}\ (\mathit{Node}\ a\ l\ r) & = & a : \mathit{interleave}\ (\mathit{flatten}\ l)\ (\mathit{flatten}\ r) \\
\mathit{interleave} & :: & \forall a.\ [a] \to [a] \to [a] \\
\mathit{interleave}\ [\,]\ as_2 & = & as_2 \\
\mathit{interleave}\ (a : as_1)\ as_2 & = & a : \mathit{interleave}\ as_2\ as_1.
\end{array}$$

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

$$\begin{array}{lcl}
\mathit{flattenId} & :: & \mathit{Flatten}\ \mathit{Id} \\
\mathit{flattenId}\ (\mathit{Id}\ a) & = & [a] \\
\mathit{flattenFork} & :: & \forall t.\ \mathit{Flatten}\ t \to \mathit{Flatten}\ (\mathit{Fork}\ t) \\
\mathit{flattenFork}\ \mathit{flt}\ (\mathit{Fork}\ l\ r) & = & \mathit{interleave}\ (\mathit{flt}\ l)\ (\mathit{flt}\ r) \\
\mathit{flattenNode} & :: & \forall t.\ \mathit{Flatten}\ t \to \mathit{Flatten}\ (\mathit{Node}\ t) \\
\mathit{flattenNode}\ \mathit{flt}\ (\mathit{Node}\ a\ l\ r) & = & a : \mathit{interleave}\ (\mathit{flt}\ l)\ (\mathit{flt}\ r)
\end{array}$$

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 *flattenFork* and *flattenNode* have so-called *rank-2 type signatures* (Leivant, 1983), that is, they operate on polymorphic functions.[4] Now, in defining the traver-

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

$$\begin{array}{lcl}
\mathit{flattenFork} & :: & \forall t.\ \forall a.\ (t\ a \to [a]) \to (\mathit{Fork}\ t\ a \to [a]) \\
\mathit{flattenNode} & :: & \forall t.\ \forall a.\ (t\ a \to [a]) \to (\mathit{Node}\ t\ a \to [a]).
\end{array}$$

sal function for *Vector* we follow again the structure of the datatype definition.

```
flattenVector                 ::  Flatten Vector
flattenVector Empty           =   []
flattenVector (NonEmpty v)    =   flattenVector' flattenId v
flattenVector'                ::  ∀t. Flatten t → 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:

```
type Sub t                =  ∀a. Int → t a → 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

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

function into the three cases.

$$
\begin{array}{lcl}
subId & :: & Sub\ Id \\
subId\ n\ (Id\ a) & & \\
\quad \mid n == 0 & = & Just\ a \\
\quad \mid otherwise & = & Nothing \\
subFork & :: & \forall t.\ Sub\ t \to Sub\ (Fork\ t) \\
subFork\ subt\ n\ (Fork\ l\ r) & & \\
\quad \mid n\ `mod`\ 2 == 0 & = & subt\ (n\ `div`\ 2)\ l \\
\quad \mid otherwise & = & subt\ (n\ `div`\ 2)\ r \\
subNode & :: & \forall t.\ Sub\ t \to Sub\ (Node\ t) \\
subNode\ subt\ n\ (Node\ a\ l\ r) & & \\
\quad \mid n == 0 & = & Just\ a \\
\quad \mid m\ `mod`\ 2 == 0 & = & subt\ (m\ `div`\ 2)\ l \\
\quad \mid otherwise & = & subt\ (m\ `div`\ 2)\ r \\
\quad \textbf{where}\ m & = & n-1
\end{array}
$$

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

$$
\begin{array}{lcl}
subVector & :: & Sub\ Vector \\
subVector\ n\ Empty & = & Nothing \\
subVector\ n\ (NonEmpty\ v) & = & subVector'\ subId\ n\ v \\
subVector' & :: & Sub\ t \to Sub\ (Vector'\ t) \\
subVector'\ subt\ n\ (Base\ v) & = & subt\ n\ v \\
subVector'\ subt\ n\ (Zero\ v) & = & subVector'\ (subFork\ subt)\ n\ v \\
subVector'\ subt\ n\ (One\ v) & = & subVector'\ (subNode\ subt)\ n\ v
\end{array}
$$

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 $Base\ (t\ (t\ a))$ instead of $Base\ (t\ a)$. Consequently, we only have to change one equation.

$$
\begin{array}{lcl}
\textbf{type}\ SubMatrix\ m & = & \forall a.\ (Int, Int) \to m\ a \to Maybe\ a \\
subSquare & :: & SubMatrix\ Square \\
subSquare\ (i,j)\ Empty & = & Nothing \\
subSquare\ (i,j)\ (NonEmpty\ m) & = & subSquare'\ subId\ (i,j)\ m \\
subSquare' & :: & \forall t.\ Sub\ t \to SubMatrix\ (Square'\ t) \\
subSquare'\ subt\ (i,j)\ (Base\ m) & = & subt\ i\ m \gg\!= subt\ j \\
subSquare'\ subt\ (i,j)\ (Zero\ m) & = & subSquare'\ (subFork\ subt)\ (i,j)\ m \\
subSquare'\ subt\ (i,j)\ (One\ m) & = & subSquare'\ (subNode\ subt)\ (i,j)\ m
\end{array}
$$

Note that the call $subt\ n\ v$ has been replaced by $subt\ i\ m \gg\!= subt\ j$. A remark is appropriate: the so-called monadic application ($\gg\!=$) takes care of 'out of range'

exceptions. If *subt i m* yields *Nothing*, then *Nothing* is propagated; if it returns *Just v*, then *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*.

```
type Make t    =  ∀a. a → t a
make           ::  [Bit] → Make Tree      -- LSB first
make [1] a     =  Id a
make (0 : bs) a  =  Fork (make bs a) (make bs a)
make (1 : bs) a  =  Node a (make bs a) (make bs a)
```

The call *make* (*bits n*) *a* creates a vector containing *n* copies of *a*. Now, since *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 *Zero* and *One* constructors. Consequently, the size parameter is no longer necessary when the actual tree is constructed motivating the following definitions:

```
makeId           ::  Make Id
makeId a         =  Id a
makeFork         ::  ∀t. Make t → Make (Fork t)
makeFork maket a =  Fork (maket a) (maket a)
makeNode         ::  ∀t. Make t → Make (Node t)
makeNode maket a =  Node a (maket a) (maket a).
```

The case analysis *makeVector′* performs is similar to that of *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 *maket* is constructed bottom-up.

```
makeVector                ::  [Bit] → Make Vector      -- MSB first
makeVector [] a           =  Empty
makeVector (1 : bs) a     =  NonEmpty (makeVector′ bs makeId a)
makeVector′               ::  [Bit] → ∀t. Make t → Make (Vector′ t)
makeVector′ [] maket a    =  Base (maket a)
makeVector′ (0 : bs) maket a  =  Zero (makeVector′ bs (makeFork maket) a)
makeVector′ (1 : bs) maket a  =  One (makeVector′ bs (makeNode 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 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 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:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{datatype}\ `a\ \mathit{perfect}\ \textbf{with}\ \mathit{nat}} \\
= & \mathit{Leaf}\ (0)\ \textbf{of}\ `a & \\
\mid & \{n : \mathit{nat}\}\mathit{Fork}\ (n+1)\ \textbf{of}\ `a\ \mathit{perfect}\ (n) * `a\ \mathit{perfect}\ (n). &
\end{array}
$$

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.

A direction for future work suggests itself. It remains 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? 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.

References

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

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1983) *Data Structures and Algorithms.* Addison-Wesley Publishing Company.

Bird, R. and de Moor, O. (1997) *Algebra of Programming.* Prentice Hall Europe.

Bird, R. and Meertens, L. (1998) Nested datatypes. Jeuring, J. (ed), *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden.* Lecture Notes in Computer Science 1422, pp. 52–67. Springer-Verlag.

Braun, W. and Rem, M. (1983) *A logarithmic implementation of flexible arrays.* Memorandum MR83/4, Eindhoven University of Technology.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1991) *Introduction to Algorithms.* The MIT Press.

Dielissen, V. J. and Kaldewaij, A. (1995) A simple, efficient, and flexible implementation of flexible arrays. *Third International Conference on Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany.* Lecture Notes in Computer Science 947, pp. 232–241. Springer-Verlag.

Guibas, L. J. and Sedgewick, R. (1978) A diochromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* pp. 8–21. IEEE Computer Society.

Hinze, R. (1998) *Numerical Representations as Higher-Order Nested Datatypes.* Tech. rept. IAI-TR-98-12. Institut für Informatik III, Universität Bonn.

Hinze, R. (1999) Constructing red-black trees. Okasaki, C. (ed), *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France* pp. 89–99. The proceedings appeared as a technical report of Columbia University, CUCS-023-99, also available from `http://www.cs.columbia.edu/~cdo/waaapl.html`.

Hinze, R. (2000a) Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming* **10**(3):305–317.

Hinze, R. (2000b) A new approach to generic functional programming. Reps, T. W. (ed), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21* pp. 119–132.

Hinze, R. (2000c) Polytypic programming with ease. *Journal of Functional and Logic Programming.* To appear.

Hoogendijk, P. and de Moor, O. (2000) Container types categorically. *Journal of Functional Programming* **10**(2):91–225.

Jones, M. P. (1995) Functional programming with overloading and higher-order polymorphism. Jeuring, J. and Meijer, E. (eds), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden.* Lecture Notes in Computer Science 925, pp. 97–136. Springer-Verlag.

Leivant, D. (1983) Polymorphic type inference. *Proc. 10th Symposium on Principles of Programming Languages.*

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA.* Lecture Notes in Computer Science 523, pp. 124–144. Springer-Verlag.

Mycroft, A. (1984) Polymorphic type schemes and recursive definitions. Paul, M. and Robinet, B. (eds), *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France.* Lecture Notes in Computer Science 167, pp. 217–228.

Okasaki, C. (1997) Functional Pearl: Three algorithms on Braun trees. *Journal of Functional Programming* **7**(6).

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Okasaki, C. (1999) From fast exponentiation to square matrices: An adventure in types. Lee, P. (ed), *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, Paris, France* pp. 28–35.

Peyton Jones, S. and Hughes, J. (eds). (1999) *Haskell 98 — A Non-strict, Purely Functional Language.* Available from `http://www.haskell.org/definition/`.

Snyder, L. (1977) On uniquely represented data structures (extended abstract). *18th Annual Symposium on Foundations of Computer Science, Providence* pp. 142–146. IEEE Computer Society Press.

Williams, J. (1964) Algorithm 232: Heapsort. *Communications of the ACM* **7**(6):347–348.

Xi, H. (1999) Dependently typed data structures. Okasaki, C. (ed), *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France* pp. 17–32. The proceedings appeared as a technical report of Columbia University, CUCS-023-99, also available from `http://www.cs.columbia.edu/~cdo/waaapl.html`.

Zenger, C. (1997) Indexed types. *Theoretical Computer Science* **187**(1–2):147–165.

Zenger, C. (1998) *Indizierte Typen.* PhD thesis, Universität Karlsruhe.

A Proofs

A.1 Proof of Theorem 1

Recall the definitions of X and f

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

where N is some finite index set, A is an arbitrary multiset and the h_i are homomorphisms, that is, they satisfy $h_i\ \varnothing = \varnothing$ 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 $\mathcal{F}$

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

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

$$\begin{array}{lcllcl} \mathcal{X}_0 & = & \varnothing & \mathcal{F}_0 & = & \lambda Y \to \varnothing \\ \mathcal{X}_{n+1} & = & \mathcal{X}\ \mathcal{X}_n & \mathcal{F}_{n+1} & = & \mathcal{F}\ \mathcal{F}_n. \end{array}$$

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

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

by induction over the natural numbers.

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

$$\begin{array}{ll} & \mathcal{X}_0 \\ = & \quad \{\ \text{definition } \mathcal{X}_0\ \} \\ & \varnothing \\ = & \quad \{\ \text{definition } \mathcal{F}_0\ \} \\ & \mathcal{F}_0\ A. \end{array}$$

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

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

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** $n = 0$:

$$\begin{array}{ll} & \biguplus_{i \in N} h_i \; (\mathcal{F}_0 \; Y) \\ = & \quad \{ \text{ definition } \mathcal{F}_0 \text{ } \} \\ & \biguplus_{i \in N} h_i \; \varnothing \\ = & \quad \{ \text{ the } h_i \text{ are homomorphisms } \} \\ & \biguplus_{i \in N} \varnothing \\ = & \quad \{ \text{ definition } \mathcal{F}_0 \text{ } \} \\ & \biguplus_{i \in N} \mathcal{F}_0 \; (h_i \; Y). \end{array}$$

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

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

  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:

$$\begin{array}{ll} & \biguplus_{i \in N} h_i \; (\mathcal{F}_{m+1} \; Y) \\ = & \quad \{ \text{ definition } \mathcal{F}_{m+1} \text{ and } \mathcal{F} \text{ } \} \\ & \biguplus_{i \in N} h_i \; (Y \uplus \biguplus_{k \in N} \mathcal{F}_m \; (h_k \; Y)) \end{array}$$

$$
\begin{aligned}
&= \quad \{ \text{ the } h_i \text{ are homomorphisms } \} \\
&\quad \biguplus_{i \in N} (h_i\ Y \uplus \biguplus_{k \in N} h_i\ (\mathcal{F}_m\ (h_k\ Y))) \\
&= \quad \{ \text{ associativity } \} \\
&\quad \biguplus_{i \in N} h_i\ Y \uplus \biguplus_{j \in N} \biguplus_{k \in N} h_j\ (\mathcal{F}_m\ (h_k\ Y)) \\
&= \quad \{ \text{ interchanging the order of union } \} \\
&\quad \biguplus_{i \in N} h_i\ Y \uplus \biguplus_{k \in N} \biguplus_{j \in N} h_j\ (\mathcal{F}_m\ (h_k\ Y)) \\
&= \quad \{ \text{ ex hypothesi } \} \\
&\quad \biguplus_{i \in N} h_i\ Y \uplus \biguplus_{k \in N} \biguplus_{j \in N} \mathcal{F}_m\ (h_j\ (h_k\ Y)) \\
&= \quad \{ \text{ associativity } \} \\
&\quad \biguplus_{i \in N} (h_i\ Y \uplus \biguplus_{j \in N} \mathcal{F}_m\ (h_j\ (h_i\ Y))) \\
&= \quad \{ \text{ definition } \mathcal{F}_{m+1} \text{ and } \mathcal{F} \} \\
&\quad \biguplus_{i \in N} \mathcal{F}_{m+1}\ (h_i\ Y).
\end{aligned}
$$

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 *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 *List* is only a very popular instance. Unfortunately, in Haskell the programmer must define *sum* for each datatype from scratch, typically, by induction on the structure of values. Using a polytypic definition *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

$$
\begin{array}{lcl}
sum\langle F \rangle & :: & F\ Nat \to Nat \\
sum\langle K\ Unit \rangle\ x & = & 0 \\
sum\langle Id \rangle\ x & = & x \\
sum\langle F_1 \mid F_2 \rangle\ x & = & \mathbf{case}\ x\ \mathbf{of}\ \{ Left\ x_1 \to sum\langle F_1 \rangle\ x_1; Right\ x_2 \to sum\langle F_2 \rangle\ x_2 \} \\
sum\langle F_1 \times F_2 \rangle\ x & = & sum\langle F_1 \rangle\ (fst\ x) + sum\langle F_2 \rangle\ (snd\ x).
\end{array}
$$

A polytypic function is uniquely defined by giving cases for the constant functor $K\ Unit$, 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, that is, $sum\langle K\ Void \rangle = \bot$ where $K\ Void$ is the 'bottom functor' and $\bot$ is the bottom element of $Void \to Nat$.

The second polytypic function we require is $fan\langle F \rangle$, which generates the multiset

of all structures of type $F\ a$ from a given seed of type a.

$$\begin{array}{lcl}
fan\langle F\rangle & :: & \forall a.a \to \wr F\ a\wr \\
fan\langle K\ Unit\rangle\ x & = & \wr()\wr \\
fan\langle Id\rangle\ x & = & \wr x\wr \\
fan\langle F_1 \mid F_2\rangle\ x & = & \wr Left\ x_1 \mid x_1 \leftarrow fan\langle F_1\rangle\ x\wr \uplus \wr Right\ x_2 \mid x_2 \leftarrow fan\langle F_2\rangle\ x\wr \\
fan\langle F_1 \times F_2\rangle\ x & = & \wr(x_1, x_2) \mid x_1 \leftarrow fan\langle F_1\rangle\ x; x_2 \leftarrow fan\langle F_2\rangle\ x\wr
\end{array}$$

Note that $fan\langle K\ Void\rangle = \lambda x \to \varnothing$ by strictness. Finally, we define

$$\begin{array}{lcl}
Size\langle F\rangle & :: & \wr Nat\wr \\
Size\langle F\rangle & = & \wr sum\langle F\rangle\ a \mid a \leftarrow fan\langle F\rangle\ 1\wr,
\end{array}$$

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 $M = Size\langle F\rangle$. The following equations can be shown using straightforward calculations:

$$\begin{array}{rcll}
Size\langle K\ Void\rangle & = & \varnothing & \text{(A 1)} \\
Size\langle K\ Unit\rangle & = & \wr 0\wr & \text{(A 2)} \\
Size\langle Id\rangle & = & \wr 1\wr & \text{(A 3)} \\
Size\langle F_1 \mid F_2\rangle & = & Size\langle F_1\rangle \uplus Size\langle F_2\rangle & \text{(A 4)} \\
Size\langle F_1 \times F_2\rangle & = & Size\langle F_1\rangle + Size\langle F_2\rangle. & \text{(A 5)}
\end{array}$$

It remains to establish

$$Size\langle F \cdot G\rangle \quad = \quad Size\langle F\rangle * Size\langle G\rangle \qquad \text{if } Size\langle G\rangle \text{ is simple.} \qquad \text{(A 6)}$$

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

- **Case** $F = K\ Void$:

$$\begin{array}{ll}
 & Size\langle K\ Void \cdot G\rangle \\
= & \quad \{\ K\ Void \cdot F = K\ Void\ \} \\
 & Size\langle K\ Void\rangle \\
= & \quad \{\ \text{Eq. (A 1)}\ \} \\
 & \varnothing \\
= & \quad \{\ \varnothing * A = \varnothing\ \} \\
 & \varnothing * Size\langle G\rangle \\
= & \quad \{\ \text{Eq. (A 1)}\ \} \\
 & Size\langle K\ Void\rangle * Size\langle G\rangle.
\end{array}$$

- **Case** $F = K\ Unit$: analogous using $K\ Unit \cdot F = K\ Unit$ and $\wr 0\wr * a = \wr 0\wr$.
- **Case** $F = Id$:

$$\begin{array}{ll}
 & Size\langle Id \cdot G\rangle \\
= & \quad \{\ Id \cdot F = F\ \}
\end{array}$$

$$
\begin{aligned}
& \mathit{Size}\langle G\rangle \\
= & \quad \{\ \langle\!|1|\!\rangle * A = A\ \} \\
& \langle\!|1|\!\rangle * \mathit{Size}\langle G\rangle \\
= & \quad \{\ \text{Eq. (A 2)}\ \} \\
& \mathit{Size}\langle \mathit{Id}\rangle * \mathit{Size}\langle G\rangle.
\end{aligned}
$$

- **Case** $F = F_1 \mid F_2$:

$$
\begin{aligned}
& \mathit{Size}\langle (F_1 \mid F_2) \cdot G\rangle \\
= & \quad \{\ (F \mid G) \cdot H = F \cdot H \mid G \cdot H\ \} \\
& \mathit{Size}\langle F_1 \cdot G \mid F_2 \cdot G\rangle \\
= & \quad \{\ \text{Eq. (A 4)}\ \} \\
& \mathit{Size}\langle F_1 \cdot G\rangle \uplus \mathit{Size}\langle F_2 \cdot G\rangle \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
& \mathit{Size}\langle F_1\rangle * \mathit{Size}\langle G\rangle \uplus \mathit{Size}\langle F_2\rangle * \mathit{Size}\langle G\rangle \\
= & \quad \{\ (A \uplus B) * C = A * C \uplus B * C\ \} \\
& (\mathit{Size}\langle F_1\rangle \uplus \mathit{Size}\langle F_2\rangle) * \mathit{Size}\langle G\rangle \\
= & \quad \{\ \text{Eq. (A 4)}\ \} \\
& \mathit{Size}\langle F_1 \mid F_2\rangle * \mathit{Size}\langle G\rangle.
\end{aligned}
$$

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

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