Polytypic Programming With Ease

Ralf Hinze

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

December 10, 2001

Abstract

This article proposes a new framework for a polytypic extension of functional programming languages. A polytypic functional program is one that is parameterised by datatype. Since polytypic functions are defined by induction on types rather than by induction on values, they typically operate on a higher level of abstraction than their monotypic counterparts. However, polytypic programming is not necessarily more complicated than conventional programming. In fact, a polytypic function is uniquely defined by its action on projection functors and on primitive functors such as sums and products. This information is sufficient to specialize a polytypic function to arbitrary datatypes, including mutually recursive datatypes and nested datatypes. The key idea is to use infinite trees as index sets for polytypic functions and to interpret datatypes as algebraic trees. This approach is simpler, more general, and more efficient than previous ones that are based on the initial algebra semantics of datatypes. Polytypic functions enjoy polytypic properties. We show among other things that well-known properties of various functions are obtained as direct consequences of two polytypic fusion laws.

1 Introduction

This article proposes a new framework for a polytypic extension of functional programming languages such as Haskell or Standard ML. The framework is simpler, more general, and more efficient than previous ones such as PolyP [14] that are based on the initial algebra semantics of datatypes.

A polytypic function is one that is defined by induction on the structure of types. The archetypical example of a polytypic function is $size :: F\ a \to Int$, which counts the number of values of type a in a given value of type $F\ a$. The function *size* can sensibly be defined for each parameterised datatype and it is often—but not always—a tiresomely routine matter to do so. A polytypic programming language enables the user to program *size* once and for all times. The

specialization of *size* to concrete instances of F is then handled automatically by the system. Polytypic programs are ubiquitous: typical examples include equality and comparison functions, mapping and zipping functions, pretty printers (such as Haskell's *show* function), parsers (such as Haskell's *read* function), data compression [17], and digital searching [11]. The ability to define such programs generically for all datatypes greatly simplifies the construction and maintenance of software systems.

Since polytypic functions are defined by induction on types rather than by induction on values, they tend to be more abstract than their monotypic counterparts. However, once a certain familiarity has been gained, it turns out that polytypic programming is actually simpler than conventional programming. To support this claim let us define two simple functions, *size* and *sum*, for some illustrative datatypes. Examples are given in the functional programming language Haskell 98 [23]. As a first example, consider the datatype of rose trees.

$$\begin{array}{lcl} \textbf{data}\ \mathit{Rose}\ a & = & \mathit{Branch}\ a\ (\mathit{List}\ (\mathit{Rose}\ a)) \\ \textbf{data}\ \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \end{array}$$

The size of a rose tree can be determined as follows, see, for instance [2].

$$\begin{array}{lcl} \mathit{sizer} & :: & \mathit{Rose}\ a \to \mathit{Int} \\ \mathit{sizer}\ (\mathit{Branch}\ a\ ts) & = & 1 + \mathit{suml}\ (\mathit{list}\ \mathit{sizer}\ ts) \\ \mathit{list} & :: & (a \to a') \to (\mathit{List}\ a \to \mathit{List}\ a') \\ \mathit{list}\ \varphi\ \mathit{Nil} & = & \mathit{Nil} \\ \mathit{list}\ \varphi\ (\mathit{Cons}\ a\ as) & = & \mathit{Cons}\ (\varphi\ a)\ (\mathit{list}\ \varphi\ as) \\ \mathit{suml} & :: & \mathit{List}\ \mathit{Int} \to \mathit{Int} \\ \mathit{suml}\ \mathit{Nil} & = & 0 \\ \mathit{suml}\ (\mathit{Cons}\ a\ as) & = & a + \mathit{suml}\ as \end{array}$$

The definition of *sizer* is already quite sophisticated: it makes use of the combining form *list* and the auxiliary function *suml*. The use of *list*, however, also incurs a slight run-time penalty: *list* produces an intermediate list, which is immediately consumed by *suml*. This inefficiency can be eliminated by fusing $\mathit{suml} \circ \mathit{list}\ \mathit{sizer}$ into a single function (called *sizef* below).

$$\begin{array}{lcl} \mathit{sizer} & :: & \mathit{Rose}\ a \to \mathit{Int} \\ \mathit{sizer}\ (\mathit{Branch}\ a\ ts) & = & 1 + \mathit{sizef}\ ts \\ \mathit{sizef} & :: & \mathit{List}\ (\mathit{Rose}\ a) \to \mathit{Int} \\ \mathit{sizef}\ \mathit{Nil} & = & 0 \\ \mathit{sizef}\ (\mathit{Cons}\ t\ ts) & = & \mathit{sizer}\ t + \mathit{sizef}\ ts \end{array}$$

Interestingly, this very definition naturally arises if we change the definition of rose trees to

$$\begin{array}{lcl} \textbf{data}\ \mathit{Rose}'\ a & = & \mathit{Branch}\ a\ (\mathit{Forest}\ a) \\ \textbf{data}\ \mathit{Forest}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ (\mathit{Rose}'\ a)\ (\mathit{Forest}\ a). \end{array}$$

The definition of $sumr :: Rose\ Int \to Int$ proceeds in an analogous fashion—it suffices, in fact, to replace '1' by 'a' in the definitions above.

A slightly more complex datatype is the type of perfect binary search trees.

$$\begin{array}{lcl} \textbf{data}\ Perfect\ a\ k & = & Zero\ a \mid Succ\ (Perfect\ (Node\ a\ k)\ k) \\ \textbf{data}\ Node\ a\ k & = & Node\ a\ k\ a \end{array}$$

The definition of *Perfect* is somewhat unusual in that the recursive call on the right-hand side, $Perfect\ (Node\ a\ k)\ k$, is not identical to the left-hand side of the equation: *Perfect* is an example of a so-called *nested datatype*, a term coined by R. Bird and L. Meertens [5]. Since *Perfect* only encompasses perfect binary search trees, the size of a tree of type $Perfect\ a\ a$ can be computed in logarithmic time.

$$\begin{array}{lcl} sizep & :: & Perfect\ a\ k \to Int \\ sizep\ (Zero\ a) & = & 1 \\ sizep\ (Succ\ t) & = & 2 * sizep\ t + 1 \end{array}$$

The function *sizep* counts the number of values of type a in a tree of type $Perfect\ a\ a$. However, *sizep* cannot be assigned the type $Perfect\ a\ a \to Int$ since the recursive call has type $Perfect\ (Node\ a\ k)\ k \to Int$.

Summing up a perfect tree of integers is more challenging.

$$\begin{array}{lcl} sump & :: & Perfect\ Int\ Int \to Int \\ sump\ (Zero\ a) & = & a \\ sump\ (Succ\ t) & = & sump\ (perfect\ sumn\ id\ t) \\ sumn & :: & Node\ Int\ Int \to Int \\ sumn\ (Node\ l\ k\ r) & = & l + k + r \\ perfect & :: & (a \to a') \to (k \to k') \\ & & \to (Perfect\ a\ k \to Perfect\ a'\ k') \\ perfect\ \varphi_1\ \varphi_2\ (Zero\ a) & = & Zero\ (\varphi_1\ a) \\ perfect\ \varphi_1\ \varphi_2\ (Succ\ t) & = & Succ\ (perfect\ (node\ \varphi_1\ \varphi_2)\ \varphi_2\ t) \\ node & :: & (a \to a') \to (k \to k') \\ & & \to (Node\ a\ k \to Node\ a'\ k') \\ node\ \varphi_1\ \varphi_2\ (Node\ l\ k\ r) & = & Node\ (\varphi_1\ l)\ (\varphi_2\ k)\ (\varphi_1\ r) \end{array}$$

Note that *perfect* and *node* denote the mapping functions of the binary functors *Perfect* and *Node*. Improving the efficiency of *sump* by fusing $sump \circ perfect\ sumn\ id$ is left as an exercise to the reader.

Now, let us define *size* and *sum* once and for all times. To this end we must first take a closer look at the structure of types. Reconsider the datatype definitions given above. Haskell's **data** construct combines several features in a single coherent form: sums, products, and recursion. The structure of the types

becomes more apparent if the definitions are rewritten as *functor equations*:

$$\begin{array}{lcl} \mathit{Rose} & = & \mathit{Id} \times \mathit{List} \cdot \mathit{Rose} \\ \mathit{List} & = & K1 + \mathit{Id} \times \mathit{List} \\ \mathit{Perfect} & = & \mathit{Fst} + \mathit{Perfect} \cdot (\mathit{Node}, \mathit{Snd}) \\ \mathit{Node} & = & \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst}, \end{array}$$

where $K\,T$ is the constant functor, Id is the identity functor, Fst and Snd are projection functors, and $F \cdot (F_1, \ldots, F_n)$ denotes the composition of an n-ary functor F with n functors, all of the same arity. Sum and product are defined pointwise: $(F_1 + F_2)\ T = F_1\ T + F_2\ T$ and $(F_1 \times F_2)\ T = F_1\ T \times F_2\ T$. In essence, functor equations are written in a compositional or 'point-free' style while **data** definitions are written in an applicative or 'pointwise' style. We treat 1, '+', and '×' as if they were given by the following datatype declarations.

$$\begin{array}{lcl} \textbf{data}\ 1 & = & () \\ \textbf{data}\ a_1 + a_2 & = & \mathit{Inl}\ a_1 \mid \mathit{Inr}\ a_2 \\ \textbf{data}\ a_1 \times a_2 & = & (a_1, a_2) \end{array}$$

To define *size* it suffices to specify its action on the identity functor, on constant functors, on sums, and on products. Polytypic functions are written using angle brackets to distinguish them from ordinary functions.

$$\begin{array}{lcl} \mathit{size}\langle F\rangle & :: & \forall a.F\ a \to \mathit{Int} \\ \mathit{size}\langle \mathit{Id}\rangle\ x & = & 1 \\ \mathit{size}\langle K\,T\rangle\ x & = & 0 \\ \mathit{size}\langle F_1 + F_2\rangle\ (\mathit{Inl}\ x_1) & = & \mathit{size}\langle F_1\rangle\ x_1 \\ \mathit{size}\langle F_1 + F_2\rangle\ (\mathit{Inr}\ x_2) & = & \mathit{size}\langle F_2\rangle\ x_2 \\ \mathit{size}\langle F_1 \times F_2\rangle\ (x_1, x_2) & = & \mathit{size}\langle F_1\rangle\ x_1 + \mathit{size}\langle F_2\rangle\ x_2 \end{array}$$

Each equation is more or less inevitable: a value of type $\mathit{Id}\ \ a = a$ contains one element of type a; a value of type $K\,T\ \ a = T$ contains no elements. To determine the size of an element of type $F_1\ \ a + F_2\ \ a$ we must either calculate the size of a structure of type $F_1\ \ a$ or that of a structure of type $F_2\ \ a$. The size of a structure of type $F_1\ \ a \times F_2\ \ a$ is given by the sum of the size of the two components. We can define *size* even more succinctly using a point-free style:

$$\begin{array}{lcl} \mathit{size}\langle F\rangle & :: & \forall a.F\ a \to \mathit{Int} \\ \mathit{size}\langle \mathit{Id}\rangle & = & \mathit{const}\ 1 \\ \mathit{size}\langle K\,T\rangle & = & \mathit{const}\ 0 \\ \mathit{size}\langle F_1 + F_2\rangle & = & \mathit{size}\langle F_1\rangle \triangledown \mathit{size}\langle F_2\rangle \\ \mathit{size}\langle F_1 \times F_2\rangle & = & \mathit{plus} \circ (\mathit{size}\langle F_1\rangle \times \mathit{size}\langle F_2\rangle), \end{array}$$

where $\mathit{plus}\ (a, b) = a + b$. The definitions of the other combining forms can be found in the appendix. Both styles have their pros and cons. The point-free style is usually more amenable to (mechanical) reasoning whereas the pointwise style is often more readable.

Now, from the polytypic definition of *size* the following specializations can be automatically derived.

$$
\begin{array}{lcl}
\mathit{sizeRose} & = & \mathit{sizeRose}_1\ (\mathit{const}\ 1) \\
\mathit{sizeRose}_1\ \varphi\ (\mathit{Branch}\ a\ ts) & = & \varphi\ a + \mathit{sizeList}_1\ (\mathit{sizeRose}_1\ \varphi)\ ts \\
\mathit{sizeList}_1\ \varphi\ \mathit{Nil} & = & 0 \\
\mathit{sizeList}_1\ \varphi\ (\mathit{Cons}\ a\ as) & = & \varphi\ a + \mathit{sizeList}_1\ \varphi\ as \\
\mathit{sizePerfect} & = & \mathit{sizePerfect}_2\ (\mathit{const}\ 1, \mathit{const}\ 1) \\
\mathit{sizePerfect}_2\ (\varphi_1, \varphi_2)\ (\mathit{Zero}\ a) & = & \varphi_1\ a \\
\mathit{sizePerfect}_2\ (\varphi_1, \varphi_2)\ (\mathit{Succ}\ t) & = & \mathit{sizePerfect}_2\ (\mathit{sizeNode}_2\ (\varphi_1, \varphi_2), \varphi_2)\ t \\
\mathit{sizeNode}_2\ (\varphi_1, \varphi_2)\ (\mathit{Node}\ l\ k\ r) & = & \varphi_1\ l + \varphi_2\ k + \varphi_1\ r
\end{array}
$$

We postpone a detailed explanation of the specialization process until Section 4. For the moment it suffices to note that the different instances rigidly follow the structure of the respective datatypes. How do these functions relate to the handcrafted definitions? Now, *sizeRose* corresponds to the second, more efficient definition of *sizer*: we have $\mathit{sizer} = \mathit{sizeRose}_1\ (\mathit{const}\ 1)$ and $\mathit{sizef} = \mathit{sizeList}_1\ \mathit{sizer}$. On the negative side, *sizePerfect* is a linear-time implementation as opposed to the logarithmic *sizep*. In general, a 'structure-strict', polytypic function has at least a linear running time. So we cannot reasonably expect to achieve the efficiency of a handcrafted implementation that exploits data-structural invariants.

The polytypic definition of *sum* is equally simple.

$$
\begin{array}{lcl}
\mathit{sum}\langle F\rangle & :: & F\ \mathit{Int} \to \mathit{Int} \\
\mathit{sum}\langle \mathit{Id}\rangle\ x & = & x \\
\mathit{sum}\langle K\ T\rangle\ x & = & 0 \\
\mathit{sum}\langle F_1 + F_2\rangle\ (\mathit{Inl}\ x_1) & = & \mathit{sum}\langle F_1\rangle\ x_1 \\
\mathit{sum}\langle F_1 + F_2\rangle\ (\mathit{Inr}\ x_2) & = & \mathit{sum}\langle F_2\rangle\ x_2 \\
\mathit{sum}\langle F_1 \times F_2\rangle\ (x_1, x_2) & = & \mathit{sum}\langle F_1\rangle\ x_1 + \mathit{sum}\langle F_2\rangle\ x_2
\end{array}
$$

Specializing $\mathit{sum}\langle F\rangle$ to the various datatypes yields definitions similar to those obtained for $\mathit{size}\langle F\rangle$. In this case the generated functions are as efficient as the best handcoded implementations.

The moral of the story so far: giving ad-hoc definitions of functions like *size* and *sum* is sometimes simple and sometimes involving. While the polytypic definition is slightly more abstract, it is also to a high degree inevitable. It is this feature that makes polytypic programming light and sweet.

The rest of this article is organized as follows. Section 2 reviews background material from the theory of infinite trees. Section 3 introduces the basic ingredients of polytypic programming: functor expressions and algebraic trees. Section 4 explains how to specialize a polytypic function to concrete instances of datatypes. Polytypic functions enjoy polytypic properties. Fixpoint induction on the functor level and fusion laws analogous to the fusion law for catamorphisms are described in Section 5. Section 6 presents several examples of polytypic functions and associated polytypic properties. Finally, Section 7 reviews related work and points out a direction for future work.

2 Preliminaries

This section introduces basic notions and facts from the theory of infinite trees as needed in the subsequent sections. For a more detailed survey the reader is referred to B. Courcelle's article [7], which also contains further references.

A *ranked set* is a family $(F^k \mid k \in \mathbb{N})$ of pairwise disjoint sets F^k of symbols of rank (or arity) k. Given a ranked set F of function symbols we denote by $\mathbb{A}(F)$ the set of all finite trees over F and by $\mathbb{A}^\infty(F)$ the set of all trees (finite and infinite) over F, see [7] for details. It is well-known that $\mathbb{A}(F)$ constitutes an *initial algebra*, which implies the following two facts.

FACT 1 (DEFINITION BY STRUCTURAL INDUCTION) Given a set A and a family of functions $(f_A :: A \times \cdots \times A \to A \mid f \in F^k)$, there exists a unique function $\varphi :: \mathbb{A}(F) \to A$ such that

$$\varphi(f(t_1, \ldots, t_k)) \quad = \quad f_A(\varphi(t_1), \ldots, \varphi(t_k))$$

for all $k \in \mathbb{N}$ and $f \in F^k$. □

FACT 2 (PROOF BY STRUCTURAL INDUCTION) Let $\mathcal{P} \subseteq \mathbb{A}(F)$ be a property of finite trees. To establish $\mathcal{P}(t)$ for all $t \in \mathbb{A}(F)$ it suffices to show

$$\mathcal{P}(t_1) \wedge \cdots \wedge \mathcal{P}(t_k) \quad \Longrightarrow \quad \mathcal{P}(f(t_1, \ldots, t_k))$$

for all $k \in \mathbb{N}$ and $f \in F^k$. □

The set of infinite trees $\mathbb{A}^\infty(F)$ can be turned into a complete partial order if we extend F by a new function symbol of arity 0, say, Ω such that Ω is the least element with respect to a suitably chosen partial order. The set $\mathbb{A}^\infty(F \cup \{\Omega\})$ even constitutes an *initial continuous algebra*, which implies the following two facts.

FACT 3 Every monotone function $\varphi :: \mathbb{A}(F \cup \{\Omega\}) \to D$, where D is a complete partial order, can be uniquely extended into a continuous function of type $\mathbb{A}^\infty(F \cup \{\Omega\}) \to D$. □

Note that a function $\varphi :: \mathbb{A}(F) \to D$ can be turned into a monotone function $\mathbb{A}(F \cup \{\Omega\}) \to D$ by setting $\varphi(\Omega) = \bot$, where $\bot$ is the least element of D.

A property $\mathcal{P} \subseteq \mathbb{A}^\infty(F \cup \{\Omega\})$ is called *pointed* if $\mathcal{P}(\Omega)$; it is called *chain-complete* if $\mathcal{P}(S) \Longrightarrow \mathcal{P}(\bigsqcup S)$ for every chain $S \subseteq \mathbb{A}^\infty(F \cup \{\Omega\})$.

FACT 4 Let $\mathcal{P} \subseteq \mathbb{A}^\infty(F \cup \{\Omega\})$ be a pointed and chain-complete property of infinite trees. To establish $\mathcal{P}(t)$ for all $t \in \mathbb{A}^\infty(F \cup \{\Omega\})$ it suffices to show $\mathcal{P}(t)$ for all $t \in \mathbb{A}(F)$. □

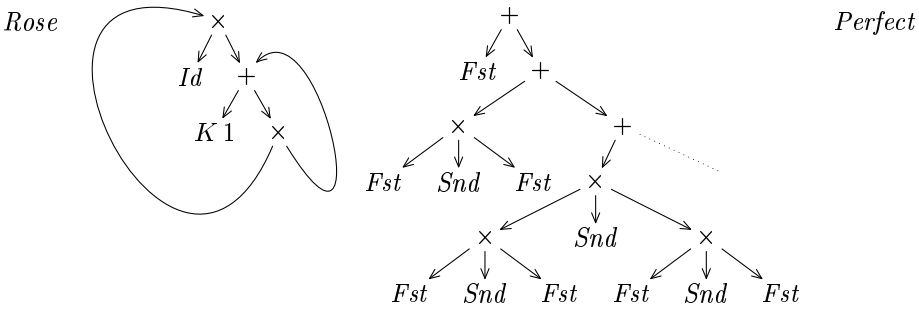


Figure 1: Types interpreted as infinite trees

3 Datatypes as Algebraic Trees

In the introduction we have seen that type definitions in Haskell take the form of recursion equations. If we interpret these equations syntactically, each equation defines a unique *infinite functor tree*. Figure 1 displays the trees defined by the type equations given in Section 1. Note that *Rose* is interpreted by a *rational tree* while *Perfect* denotes an *algebraic tree*. A rational tree is a possibly infinite tree that has only a finite number of subtrees. Algebraic trees are obtained as solutions of so-called algebraic equations, which are akin to functor equations defined below.

Given a ranked set of functor variables $\mathbb{X}$ and a ranked set of primitive functors $\mathbb{P}$, the set of *functor expressions of arity* n is inductively defined as follows.

$$\mathbb{F}^n \quad ::= \quad \mathbb{X}^n \mid \Pi^n_i \mid \mathbb{P}^n \mid \mathbb{F}^k \cdot (\mathbb{F}^n_1, \ldots, \mathbb{F}^n_k) \mid \mathbb{F}^n \textbf{ where } \mathbb{D}$$

Here Π^n_i denotes the n-ary projection functor selecting the i-th component. For unary and binary projection functors we use the following more familiar names: $Id = \Pi^1_1$, $Fst = \Pi^2_1$, and $Snd = \Pi^2_2$. The expression $F \cdot (F_1, \ldots, F_k)$ denotes the composition of a k-ary functor F with functors F_i, all of arity n. We omit the parentheses when $k = 1$ and we write F instead of $F \cdot ()$ when $k = 0$. The keyword '**where**' introduces local functor equations, where the set $\mathbb{D}$ of *functor equations* is given by the following grammar.

$$\mathbb{D} \quad ::= \quad \{\mathbb{X}^{k_1} = \mathbb{F}^{k_1}; \ldots; \mathbb{X}^{k_p} = \mathbb{F}^{k_p}\}$$

Note that Haskell does not provide local type declarations. We have included them, however, to make the language of functors more uniform.

The choice of $\mathbb{P}$ is more or less arbitrary. For concreteness, we set $\mathbb{P} = \mathbb{P}^0 \cup \mathbb{P}^2$ with $\mathbb{P}^0 = \{K1, KInt\}$ and $\mathbb{P}^2 = \{+, \times\}$. As usual, binary functors are written infix, that is, we write $F_1 \oplus F_2$ instead of $\oplus \cdot (F_1, F_2)$ for $\oplus \in \mathbb{P}^2$.

Each functor expression defines a unique infinite tree, called functor tree, whose inner nodes are labelled with primitive functors of arity $\geqslant 1$ and whose

leaves are decorated with nullary functors and projection functors. In a sense, we can view infinite trees as a kind of 'infinite normal form' of functor expressions. Possibly infinite *functor trees* are formed according to the following grammar.

$$\mathbb{T}^n \quad ::= \quad \Pi_i^n \mid \mathbb{P}^k \cdot (\mathbb{T}_1^n, \ldots, \mathbb{T}_k^n)$$

Thus, $P \cdot (T_1, \ldots, T_k)$ with $P \in \mathbb{P}^k$ corresponds to a tree, whose root is labelled with P and which has subtrees $T_1, \ldots, T_k$, and Π_i^n corresponds to a leaf.

The functor tree denoted by a functor expression is given by

$$\begin{array}{lcl}
[\![X]\!]\eta & = & \eta(X) \\
[\![\Pi_i^n]\!]\eta & = & \Pi_i^n \\
[\![P]\!]\eta & = & P \cdot (\Pi_1^k, \ldots, \Pi_k^k) \\
[\![F \cdot (F_1, \ldots, F_k)]\!]\eta & = & [\![F]\!]\eta \circ ([\![F_1]\!]\eta, \ldots, [\![F_k]\!]\eta) \\
[\![F \textbf{ where } X_1 = F_1]\!]\eta & = & [\![F]\!]\eta(X_1 := \mathbf{lfp}(\lambda T_1.[\![F_1]\!]\eta(X_1 := T_1))).
\end{array}$$

Here η is an environment mapping functor variables to infinite functor trees and $\eta(X := T)$ is syntax for extending the environment η by the binding $X := T$. If the functor expression is closed, we omit the environment, that is, we write $[\![F]\!]$ instead of $[\![F]\!]\eta$. Functor composition is interpreted by a substitution operation on functor trees:

$$\begin{array}{lcl}
\Omega \circ \mathbf{U} & = & \Omega \\
\Pi_i^n \circ \mathbf{U} & = & U_i \\
(P \cdot (T_1, \ldots, T_k)) \circ \mathbf{U} & = & P \cdot (T_1 \circ \mathbf{U}, \ldots, T_k \circ \mathbf{U}),
\end{array}$$

where $\mathbf{U} = (U_1, \ldots, U_n)$. The first equation specifies that '$\circ$' is strict in its first argument, the second defines the substitution of a leaf by a tree, and the third formalizes the propagation of a substitution to the subtrees of a node. Note that Fact 3 implies that '$\circ$' can be uniquely extended to a continuous function on infinite functor trees. We will make use of the following properties of the substitution operation:

$$\begin{array}{lcll}
T \circ (\Pi_1^n, \ldots, \Pi_n^n) & = & T & (1) \\
(T \circ (T_1, \ldots, T_k)) \circ \mathbf{U} & = & T \circ (T_1 \circ \mathbf{U}, \ldots, T_k \circ \mathbf{U}). & (2)
\end{array}$$

The semantics of a functor equation is given by its least fixpoint (for reasons of readability we consider only single functor equations). Note that an equation of the form $X = P \cdot (F_1, \ldots, F_k)$ with $P \in \mathbb{P}^k$ even has a *unique* solution in the realm of infinite trees [7].

EXAMPLE 1 The functor equations $X = X$ and $X = X \cdot X$ have the least solution Ω (note that composition is strict in its first argument). □

EXAMPLE 2 The unique solution of the functor equation $\mathit{Perfect} = \mathit{Fst} + \mathit{Perfect} \cdot (\mathit{Node}, \mathit{Snd})$ is given by $P_0 + (P_1 + (P_2 + \cdots))$, where $P_0 = \mathit{Fst}$ and $P_{n+1} = P_n \times \mathit{Snd} \times P_n$. Thus, *Perfect* is the disjoint union of perfectly balanced trees of arbitrary height. □

Building upon the semantics we can classify functors into polynomial, regular, and nested functors. A functor is called *polynomial* if it denotes a finite functor tree.

$$\begin{array}{lcl} \Delta & = & Id \times Id \\ Node23 & = & Fst \times Snd \times Fst + Fst \times Snd \times Fst \times Snd \times Fst \end{array}$$

The functor Δ is called the diagonal or square functor; *Node23* is used below in the definition of 2-3 trees.

A *regular* functor is one that yields a rational functor tree.

$$Bintree = Fst + Bintree \times Snd \times Bintree$$

The functor *Bintree* models external, binary search trees. Further examples of regular functors are *Rose*, *List*, *Rose′*, and *Forest*.

The functor *Perfect* is an example of a *non-regular* or *nested* functor, which is interpreted by an algebraic functor tree.

$$\begin{array}{lcl} Sequ & = & K1 + Sequ \cdot \Delta + Id \times Sequ \cdot \Delta \\ Tree23 & = & Fst + Tree23 \cdot (Node23, Snd) \end{array}$$

C. Okasaki [22] uses the functor *Sequ* as the basis for a sequence implementation with an efficient indexing operation. The functor *Tree23* models 2-3 trees. The novelty of this definition is that the data-structural invariants of 2-3 trees—each interior node has two or three children and each path from the root to a leaf has the same length—are made manifest.

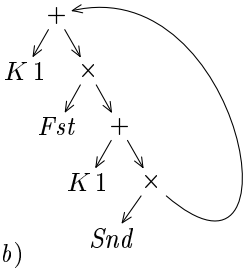
The same functor can usually be defined in a variety of ways. Take, for instance, the two definitions of rose trees given in Section 1.

$$\begin{array}{lcllcl} Rose & = & Id \times List \cdot Rose & Rose' & = & Id \times Forest \\ List & = & K1 + Id \times List & Forest & = & K1 + Rose' \times Forest \end{array}$$

Both *Rose* and *Rose′* denote the same rational tree depicted in Figure 1.

Finally, let us note that the classification of functors into regular and nested functors deviates slightly from the usual notions. As an example, consider the following datatype definition.

$$\textbf{data}\ ZigZag\ a\ b = Nil \mid Cons\ a\ (ZigZag\ b\ a)$$



Since the recursive call on the right-hand side of the definition, *ZigZag b a*, is not a copy of the left-hand side, *ZigZag* qualifies as a nested datatype. It denotes, however, the rational tree depicted on the right. An equivalent, non-nested definition for *ZigZag* is given by

$$\begin{array}{lcl} \textbf{data}\ Zig\ a\ b & = & Nil \mid Cons\ a\ (Zag\ a\ b) \\ \textbf{data}\ Zag\ a\ b & = & Nil \mid Cons\ b\ (Zig\ a\ b). \end{array}$$

While the standard classification of functors is purely syntactical—the form matters—, the classification via tree classes has a mild semantic flavour—the structure matters.

4 Polytypic Functions

We have already seen two examples of polytypic functions. In general, a polytypic function that is parameterised by m-ary functors is defined by induction on the structure of finite functor trees.

$$\begin{array}{lcl} poly\langle T\rangle & :: & Poly\langle T\rangle \\ poly\langle \Pi_i^m\rangle & = & poly_{\Pi_i^m} \\ poly\langle P\cdot(T_1,\ldots,T_k)\rangle & = & poly_P\ (poly\langle T_1\rangle,\ldots,poly\langle T_k\rangle) \end{array}$$

The type of $poly\langle T\rangle$ is specified by the type scheme $Poly\langle T\rangle$, which may contain polymorphic types. Since *poly* is parameterised by m-ary functors, $poly_{\Pi_i^m}$ must be specified for $1 \leqslant i \leqslant m$. Furthermore, an equation specifying $poly_P$ must be given for each primitive functor $P \in \mathbb{P}$. Now, setting

$$poly\langle\Omega\rangle \quad = \quad \bot$$

Fact 1 and Fact 3 imply that *poly* can be uniquely extended to a continuous function on infinite functor trees. We tacitly assume that $poly_{\Pi_i^m}$ and $poly_P$ are elements in some cpo (typically, they will be programs in Haskell or Standard ML). Fact 3 furthermore implies that the information above is sufficient to define a unique value $poly\langle[\![F]\!]\rangle$ for each closed functor expression F.

EXAMPLE 3 The code of *size* given in the introduction defines the type scheme $Size\langle F\rangle = \forall a.F\ a \to Int$ and the functions $size_{Id}$, $size_{K1}$, $size_{K\,Int}$, $size_+$, and $size_\times$.

$$\begin{array}{lcl} size_{Id} & = & const\ 1 \\ size_{K1} & = & const\ 0 \\ size_{K\,Int} & = & const\ 0 \\ size_+\ (\varphi_1,\varphi_2) & = & \varphi_1\ \triangledown\ \varphi_2 \\ size_\times\ (\varphi_1,\varphi_2) & = & plus\circ(\varphi_1\times\varphi_2) \end{array}$$

Note that $size\langle K\,T\rangle$ specifies both $size_{K1}$ and $size_{K\,Int}$. □

The question we pursue in this section is how to derive specializations of $poly\langle[\![F]\!]\rangle$ for given instances of F. The process of specialization is necessary since $poly\langle[\![F]\!]\rangle$ cannot directly be implemented in languages such as Haskell or Standard ML. The reason is simply that the type of *poly* depends on the first argument, which is itself a type (or possibly, the encoding of a type). Even if we circumvented this problem by using encodings into a universal datatype [27] or by using dynamic types and a **typecase** [1], the result would be rather inefficient because *poly* would interpret its type argument at each stage of the recursion. By specializing $poly\langle[\![F]\!]\rangle$ for a given F we remove this interpretative layer. Thus, we can view the following as a very special instance of partial evaluation.

Since datatypes correspond to algebraic trees, which may have an infinite number of different subtrees, the specialization cannot be based on the structure of datatypes. However, using functor expressions algebraic trees can be finitely represented so the idea suggests itself to carry out the specialization on the representation of functor trees. The main challenge lies in the treatment of functor composition. Assume, for instance, that a unary functor is defined in terms of a binary functor: $F = B \cdot (F_1, F_2)$. Now, if we want to specialize $size\langle[\![F]\!]\rangle$, how can we generate code for the right-hand side? Ideally, $size\langle[\![F]\!]\rangle$ should be compositionally defined in terms of the specializations for B, F_1, and F_2. Now, $[\![B]\!]$ is a binary functor mapping (T_1, T_2) to $[\![B]\!] \circ (T_1, T_2)$. This suggests defining an auxiliary function $size_2\langle\!\langle B\rangle\!\rangle$ that maps $(size\langle T_1\rangle, size\langle T_2\rangle)$ to $size\langle[\![B]\!] \circ (T_1, T_2)\rangle$ for all functor trees T_1 and T_2. We have seen that functor composition corresponds to a substitution operation on trees. Using the function $size_2\langle\!\langle B\rangle\!\rangle$ we imitate substitution on the function level. In general, we define, for each functor G of arity n, a polytypic function

$$poly_n\langle\!\langle G\rangle\!\rangle :: \forall T_1\ \ldots\ T_n.(Poly\langle T_1\rangle, \ldots, Poly\langle T_n\rangle) \to Poly\langle[\![G]\!] \circ (T_1, \ldots, T_n)\rangle$$

satisfying

$$poly_n\langle\!\langle G\rangle\!\rangle\ (poly\langle T_1\rangle, \ldots, poly\langle T_n\rangle) \quad = \quad poly\langle[\![G]\!] \circ (T_1, \ldots, T_n)\rangle,$$

for all functor trees $T_1, \ldots, T_n$. For convenience we introduce the following abbreviations: $\mathbf{T} = (T_1, \ldots, T_n)$ and $poly\langle\mathbf{T}\rangle = (poly\langle T_1\rangle, \ldots, poly\langle T_n\rangle)$. The specification captures the central idea of the specialization: the structure of types is mimicked on the value level. Compare $[\![G]\!]$ and $poly_n\langle\!\langle G\rangle\!\rangle$: $[\![G]\!]$ is an n-ary functor sending $\mathbf{T}$ to $[\![G]\!] \circ \mathbf{T}$; likewise $poly_n\langle\!\langle G\rangle\!\rangle$ is an n-ary function sending $poly\langle\mathbf{T}\rangle$ to $poly\langle[\![G]\!] \circ \mathbf{T}\rangle$.

Now, before we get down to the derivation of $poly_n$, we must first generalize the specification slightly. The equation above assumes that the functor expression G is closed. Of course, G may in general contain free functor variables. Thus, $poly_n$ must be extended by an environment mapping functor variables to continuous functions. Let ϱ be such an environment and let η be an environment mapping functor variables to infinite trees. The refined specification then reads: if $\varrho(X)\ (poly\langle\mathbf{T}\rangle) = poly\langle\eta(X) \circ \mathbf{T}\rangle$ for all free variables X of G, then

$$poly_n\langle\!\langle G\rangle\!\rangle\,\varrho\ (poly\langle\mathbf{T}\rangle) \quad = \quad poly\langle[\![G]\!]\eta \circ \mathbf{T}\rangle. \qquad \text{(specification)}$$

The condition relating ϱ and η is called the *environment condition*. The derivation of $poly_n\langle\!\langle G\rangle\!\rangle$ is a nice example in program calculation and proceeds almost mechanically.

Case $G = X$: for functor variables we obtain

$$\begin{array}{ll} & poly_n\langle\!\langle X\rangle\!\rangle\varrho\ (poly\langle\mathbf{T}\rangle) \\ = & \quad \{\ \text{specification}\ \} \\ & poly\langle[\![X]\!]\eta \circ \mathbf{T}\rangle \\ = & \quad \{\ \text{definition of } [\![\text{-}]\!]\ \} \end{array}$$

$$
\begin{array}{ll}
 & poly\langle \eta(X) \circ \mathbf{T} \rangle \\
= & \quad \{ \text{ environment condition } \} \\
 & \varrho(X)\ (poly\langle \mathbf{T} \rangle).
\end{array}
$$

Case $G = \Pi_i^n$: the derivation for projection functors is equally straightforward.

$$
\begin{array}{ll}
 & poly_n\langle\!\langle \Pi_i^n \rangle\!\rangle \varrho\ (poly\langle \mathbf{T} \rangle) \\
= & \quad \{ \text{ specification } \} \\
 & poly\langle [\![\Pi_i^n]\!] \eta \circ \mathbf{T} \rangle \\
= & \quad \{ \text{ definition of } [\![\text{-}]\!] \text{ and definition of `}\circ\text{' } \} \\
 & poly\langle T_i \rangle
\end{array}
$$

Case $G = P$: for primitive functors we calculate

$$
\begin{array}{ll}
 & poly_n\langle\!\langle P \rangle\!\rangle \varrho\ (poly\langle \mathbf{T} \rangle) \\
= & \quad \{ \text{ specification } \} \\
 & poly\langle [\![P]\!] \eta \circ \mathbf{T} \rangle \\
= & \quad \{ \text{ definition of } [\![\text{-}]\!] \text{ and definition of `}\circ\text{' } \} \\
 & poly\langle P \cdot \mathbf{T} \rangle \\
= & \quad \{ \text{ definition of } poly \ \} \\
 & poly_P\ (poly\langle \mathbf{T} \rangle).
\end{array}
$$

Case $G = H \cdot (H_1, \ldots, H_k)$: the derivation for functor composition makes essential use of the fact that the substitution operation '$\circ$' is associative.

$$
\begin{array}{ll}
 & poly_n\langle\!\langle H \cdot (H_1, \ldots, H_k) \rangle\!\rangle \varrho\ (poly\langle \mathbf{T} \rangle) \\
= & \quad \{ \text{ specification } \} \\
 & poly\langle [\![H \cdot (H_1, \ldots, H_k)]\!] \eta \circ \mathbf{T} \rangle \\
= & \quad \{ \text{ definition of } [\![\text{-}]\!] \text{ and (2) } \} \\
 & poly\langle [\![H]\!] \eta \circ ([\![H_1]\!] \eta \circ \mathbf{T}, \ldots, [\![H_k]\!] \eta \circ \mathbf{T}) \rangle \\
= & \quad \{ \text{ specification } \} \\
 & poly_k\langle\!\langle H \rangle\!\rangle \varrho\ (poly\langle [\![H_1]\!] \eta \circ \mathbf{T} \rangle, \ldots, poly\langle [\![H_k]\!] \eta \circ \mathbf{T} \rangle) \\
= & \quad \{ \text{ specification } \} \\
 & poly_k\langle\!\langle H \rangle\!\rangle \varrho\ (poly_n\langle\!\langle H_1 \rangle\!\rangle \varrho\ (poly\langle \mathbf{T} \rangle), \ldots, poly_n\langle\!\langle H_k \rangle\!\rangle \varrho\ (poly\langle \mathbf{T} \rangle)).
\end{array}
$$

Case $G = (H \textbf{ where } X_1 = H_1)$: the calculation for local functor equations proceeds as follows

$$
\begin{array}{ll}
 & poly_n\langle\!\langle H \textbf{ where } X_1 = H_1 \rangle\!\rangle \varrho\ (poly\langle \mathbf{T} \rangle) \\
= & \quad \{ \text{ specification } \} \\
 & poly\langle [\![H \textbf{ where } X_1 = H_1]\!] \eta \circ \mathbf{T} \rangle \\
= & \quad \{ \text{ definition of } [\![\text{-}]\!] \ \}
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{poly}\langle [\![H]\!]\eta(X_1 := \mathbf{lfp}(\lambda U_1.[\![H_1]\!]\eta(X_1 := U_1))) \circ \mathbf{T}\rangle \\
= & \quad \{ \text{ specification and proof obligation } \} \\
& \mathit{poly}_k \langle\!\langle H \rangle\!\rangle \varrho(X_1 := \mathbf{lfp}(\lambda\psi_1.\mathit{poly}_{k_1}\langle\!\langle H_1\rangle\!\rangle \varrho(X_1 := \psi_1))) \ (\mathit{poly}\langle \mathbf{T}\rangle).
\end{array}
$$

The last step is not entirely obvious. To be able to apply the specification we have to prove that the extended environments satisfy the environment condition. If we define the relation $\mathcal{P}$ by

$$
\mathcal{P}(\psi, U) \iff \psi\ (\mathit{poly}\langle \mathbf{T}\rangle) = \mathit{poly}\langle U \circ \mathbf{T}\rangle,
$$

then we have to show that

$$
\mathcal{P}(\mathbf{lfp}(\lambda\psi_1.\mathit{poly}_{k_1}\langle\!\langle H_1\rangle\!\rangle \varrho(X_1 := \psi_1)), \mathbf{lfp}(\lambda U_1.[\![H_1]\!]\eta(X_1 := U_1))).
$$

In other words, we must show that least fixpoints are related. To this end we can use a 2-argument variant of fixpoint induction, which can be stated as the following inference rule:

$$
\frac{\mathcal{Q}(\bot, \bot) \qquad \forall v\ w. \mathcal{Q}(v, w) \Longrightarrow \mathcal{Q}(\varphi\ v, \psi\ w)}{\mathcal{Q}(\mathbf{lfp}\ \varphi, \mathbf{lfp}\ \psi)}.
$$

Recall that fixpoint induction is only sound for chain-complete predicates. The predicate $\mathcal{P}$ satisfies this requirement since it takes the form of an equation, which involves only continuous functions. Now, it is easy to see that $\mathcal{P}$ is pointed, that is, $\mathcal{P}(\bot, \Omega)$. The induction step, where we have to show that $\mathcal{P}(\psi, U) \Longrightarrow \mathcal{P}(\mathit{poly}_{k_1}\langle\!\langle H_1\rangle\!\rangle \varrho(X_1 := \psi), [\![H_1]\!]\eta(X_1 := U))$, is a direct consequence of the specification; the assumption $\mathcal{P}(\psi, U)$ guarantees that the extended environments $\varrho(X_1 := \psi)$ and $\eta(X_1 := U)$ satisfy the environment condition.

Now, generalizing $\mathit{poly}\langle \mathbf{T}\rangle$ to the tuple $\boldsymbol{\varphi} = (\varphi_1, \ldots, \varphi_n)$ we have calculated the following definition of poly_n.

$$
\begin{array}{lcl}
\mathit{poly}_n\langle\!\langle X \rangle\!\rangle \varrho\ \boldsymbol{\varphi} & = & \varrho(X)\ \boldsymbol{\varphi} \\
\mathit{poly}_n\langle\!\langle \Pi_i^n \rangle\!\rangle \varrho\ \boldsymbol{\varphi} & = & \varphi_i \\
\mathit{poly}_n\langle\!\langle P \rangle\!\rangle \varrho\ \boldsymbol{\varphi} & = & \mathit{poly}_P\ \boldsymbol{\varphi} \\
\mathit{poly}_n\langle\!\langle H \cdot (H_1, \ldots, H_k) \rangle\!\rangle \varrho\ \boldsymbol{\varphi} & & \\
\qquad & = & \mathit{poly}_k\langle\!\langle H \rangle\!\rangle \varrho\ (\mathit{poly}_n\langle\!\langle H_1 \rangle\!\rangle \varrho\ \boldsymbol{\varphi}, \ldots, \mathit{poly}_n\langle\!\langle H_k \rangle\!\rangle \varrho\ \boldsymbol{\varphi}) \\
\mathit{poly}_n\langle\!\langle H \ \mathbf{where}\ X_1 = H_1 \rangle\!\rangle \varrho\ \boldsymbol{\varphi} & & \\
\qquad & = & \mathit{poly}_k\langle\!\langle H \rangle\!\rangle \varrho(X_1 := \mathbf{lfp}(\lambda\psi_1.\mathit{poly}_{k_1}\langle\!\langle H_1 \rangle\!\rangle \varrho(X_1 := \psi_1)))\ \boldsymbol{\varphi}
\end{array}
$$

Using a point-free style $poly_n$ can be defined as

$$\begin{array}{lcl}
poly_n\langle\!\langle X\rangle\!\rangle\varrho & = & \varrho(X) \\
poly_n\langle\!\langle \Pi_i^n\rangle\!\rangle\varrho & = & \pi_i^n \\
poly_n\langle\!\langle P\rangle\!\rangle\varrho & = & poly_P \\
poly_n\langle\!\langle H\cdot(H_1,\ldots,H_k)\rangle\!\rangle\varrho & & \\
\qquad & = & poly_k\langle\!\langle H\rangle\!\rangle\varrho \star (poly_n\langle\!\langle H_1\rangle\!\rangle\varrho,\ldots,poly_n\langle\!\langle H_k\rangle\!\rangle\varrho) \\
poly_n\langle\!\langle H \textbf{ where } X_1 = H_1\rangle\!\rangle\varrho & & \\
\qquad & = & poly_k\langle\!\langle H\rangle\!\rangle\varrho(X_1 := \mathbf{lfp}(\lambda\psi_1.poly_{k_1}\langle\!\langle H_1\rangle\!\rangle\varrho(X_1 := \psi_1))),
\end{array}$$

where $\pi_i^n\,(\varphi_1,\ldots,\varphi_n) = \varphi_i$ is the i-th projection function and '$\star$' denotes n-ary composition defined by $(\varphi\star(\varphi_1,\ldots,\varphi_n))\,a = \varphi\,(\varphi_1\ a,\ldots,\varphi_n\ a)$. It is worth emphasizing that the definition of $poly_n\langle\!\langle G\rangle\!\rangle$ is *inductive on the structure of functor expressions*. On a more abstract level we can view $poly_n$ as an interpretation of functor expressions: Π_i^n is interpreted by π_i^n, P by $poly_P$, '$\cdot$' by '$\star$', and recursion by least fixpoints.

It remains to define $poly\langle[\![F]\!]\rangle$ in terms of $poly_m\langle\!\langle F\rangle\!\rangle$—here we assume that F is closed:

$$\begin{array}{lcl}
 & & poly\langle[\![F]\!]\rangle \\
 & = & \quad \{\ (1)\ \} \\
 & & poly\langle[\![F]\!]\circ(\Pi_1^m,\ldots,\Pi_m^m)\rangle \\
 & = & \quad \{\ \text{specification}\ \} \\
 & & poly_m\langle\!\langle F\rangle\!\rangle\ (poly\langle\Pi_1^m\rangle,\ldots,poly\langle\Pi_m^m\rangle) \\
 & = & \quad \{\ \text{definition of } poly\ \} \\
 & & poly_m\langle\!\langle F\rangle\!\rangle\ (poly_{\Pi_1^m},\ldots,poly_{\Pi_m^m}).
\end{array}$$

The following proposition summarizes the derivation.

PROPOSITION 1 (CORRECTNESS OF THE SPECIALIZATION) Let $poly$ be a polytypic function parameterized by m-ary functors and let $poly_n$ be defined as above, then

$$poly\langle[\![F]\!]\rangle \quad = \quad poly_m\langle\!\langle F\rangle\!\rangle\ (poly_{\Pi_1^m},\ldots,poly_{\Pi_m^m})$$

for all closed functor expressions F of arity m. □

The above derivation takes place in a semantic setting, that is, complete partial orders and continuous functions. Of course, if we aim at implementing $poly_n$, say, as a part of a preprocessor or a compiler, we must operate on a syntactic level, that is, we have to transform functor expressions into Haskell, Standard ML, or some intermediate language. Now, the syntactic variant of $poly_n\langle\!\langle\text{-}\rangle\!\rangle$, which we denote $poly_n\langle\text{-}\rangle$, simply works by mapping the different constructs in the functorial language to corresponding constructs in the expression

language. For instance, recursion equations on the type level are mapped to recursion equations on the value level:

$$poly_n \left\langle \begin{array}{lcl} X_1 & = & F_1 \\ \cdots & & \\ X_p & = & X_p \end{array} \right\rangle \varrho \quad = \quad \begin{array}{lcl} poly X_1 & = & poly_{k_1}\langle F_1\rangle \bar{\varrho} \\ \cdots & & \\ poly X_p & = & poly_{k_p}\langle F_p\rangle \bar{\varrho}, \end{array}$$

where $\bar{\varrho} = \varrho(X_1 := poly X_1, \ldots, X_p := poly X_p)$ and k_i is the arity of X_i. Note that the $poly X_i$ are expression variables. It is straightforward to define the remaining cases of $poly_n\langle\text{-}\rangle$ such that $\mathcal{E}[\![poly_n\langle F\rangle]\!] = poly_n\langle\!\langle F\rangle\!\rangle$, where $\mathcal{E}$ is the standard denotational semantics of expressions. The correctness of the transformation then follows immediately from Proposition 1. We leave it to the reader to fill out the details.

EXAMPLE 4 Let us specialize *size* to the datatypes *Perfect* and *Node* given by

$$\begin{array}{lcl} Perfect & = & + \cdot (\Pi_1^2, Perfect \cdot (Node, \Pi_2^2)) \\ Node & = & \times \cdot (\Pi_1^2, \times \cdot (\Pi_2^2, \Pi_1^2)). \end{array}$$

This system of functor equations can be mechanically transformed into the following system of function equations.

$$\begin{array}{lcl} sizePerfect_2 & = & size_+ \star (\pi_1^2, sizePerfect_2 \star (sizeNode_2, \pi_2^2)) \\ sizeNode_2 & = & size_\times \star (\pi_1^2, size_\times \star (\pi_2^2, \pi_1^2)) \end{array}$$

Expanding the definitions of π_i^n, '$\star$', and $size_P$ we get

$$\begin{array}{lcl} sizePerfect_2\ (\varphi_1, \varphi_2) & = & \varphi_1 \triangledown sizePerfect_2\ (sizeNode_2\ (\varphi_1, \varphi_2), \varphi_2) \\ sizeNode_2\ (\varphi_1, \varphi_2) & = & plus \circ (\varphi_1 \times plus \circ (\varphi_2 \times \varphi_1)). \end{array}$$

Finally, if we use the original constructor names we obtain the Haskell code given in the introduction. □

It is worth mentioning that the specialization yields the same result for different representations of the same functor tree, that is,

$$[\![F_1]\!] = [\![F_2]\!] \implies poly_n\langle\!\langle F_1\rangle\!\rangle = poly_n\langle\!\langle F_2\rangle\!\rangle.$$

This is a simple consequence of Proposition 1. Recall that $[\![Rose]\!] = [\![Rose']\!]$. Consequently, we have $size_1\langle\!\langle Rose\rangle\!\rangle = size_1\langle\!\langle Rose'\rangle\!\rangle$. If we translate $size_1\langle\!\langle Rose\rangle\!\rangle$ and $size_1\langle\!\langle Rose'\rangle\!\rangle$ into Haskell, we obtain, of course, two different functions simply because *Rose* and *Rose'* are different types. In Haskell datatype declarations are generative, that is, each **data** declaration introduces a new distinct type. (If *Rose* and *Rose'* are defined within the same scope they even have to use different constructor names.)

5 Polytypic Properties

This section investigates another facet of polytypism: polytypic reasoning. We show how to prove properties of polytypic functions using the principle of fixpoint induction and we present two polytypic fusion laws, analogous to the fusion law for catamorphisms.

Fact 2 and Fact 4 suggest the following induction principle. Let $\mathcal{P}$ be a pointed and chain-complete property of functor trees of arity m. In order to prove that $\mathcal{P}$ holds for all functor trees, it suffices to show that

$$\begin{aligned}&\mathcal{P}(\Pi_i^m)\\&\mathcal{P}(T_1) \wedge \cdots \wedge \mathcal{P}(T_k) \Longrightarrow \mathcal{P}(P \cdot (T_1, \ldots, T_k)).\end{aligned}$$

That is, $\mathcal{P}(\Pi_i^m)$ must hold for $1 \leqslant i \leqslant m$ and the implication must hold for each primitive functor $P \in \mathbb{P}$.

EXAMPLE 5 Assume that A is a parameterized type comprising only containers of the same size, that is, $size\langle A\rangle = const\ a$. Let us illustrate the induction principle by showing

$$\mathcal{P}(F) \quad \Longleftrightarrow \quad size\langle F \circ A\rangle = times\ a \circ size\langle F\rangle,$$

where $times\ a\ b = a * b$. First of all, since $times\ a$ is strict, we have that $\mathcal{P}$ is pointed. Since $\mathcal{P}$ takes the form of an equation, it is furthermore chain-complete. We show $\mathcal{P}(F)$ for $F = Id$ and $F = F_1 \times F_2$. The remaining cases, $F = K\,T$ and $F = F_1 + F_2$, are left as exercises to the reader.
Case $F = Id$:

$$\begin{aligned}&size\langle Id \circ A\rangle\\=\quad&\{\text{ definition of `}\circ\text{' }\}\\&size\langle A\rangle\\=\quad&\{\text{ assumption }\}\\&const\ a\\=\quad&\{\text{ arithmetic: } a = a * 1\ \}\\&times\ a \circ const\ 1\\=\quad&\{\text{ definition of `}size\text{' }\}\\&times\ a \circ size\langle Id\rangle.\end{aligned}$$

Case $F = F_1 \times F_2$:

$$\begin{aligned}&size\langle (F_1 \times F_2) \circ A\rangle\\=\quad&\{\ (2)\ \}\\&size\langle F_1 \circ A \times F_2 \circ A\rangle\\=\quad&\{\text{ definition of `}size\text{' }\}\end{aligned}$$

$$
\begin{array}{ll}
 & \mathit{plus} \circ (\mathit{size}\langle F_1 \circ A\rangle \times \mathit{size}\langle F_2 \circ A\rangle) \\
= & \quad \{ \text{ ex hypothesi } \} \\
 & \mathit{plus} \circ ((\mathit{times}\ a \circ \mathit{size}\langle F_1\rangle) \times (\mathit{times}\ a \circ \mathit{size}\langle F_2\rangle)) \\
= & \quad \{ \text{ `}\times\text{' respects composition } \} \\
 & \mathit{plus} \circ (\mathit{times}\ a \times \mathit{times}\ a) \circ (\mathit{size}\langle F_1\rangle \times \mathit{size}\langle F_2\rangle) \\
= & \quad \{ \text{ arithmetic: } a * (b + c) = a * b + a * c \} \\
 & \mathit{times}\ a \circ \mathit{plus} \circ (\mathit{size}\langle F_1\rangle \times \mathit{size}\langle F_2\rangle) \\
= & \quad \{ \text{ definition of `}\mathit{size}\text{' } \} \\
 & \mathit{times}\ a \circ \mathit{size}\langle F_1 \times F_2\rangle. \quad \square
\end{array}
$$

Turning to the polytypic fusion laws let us remark beforehand that these laws are quite general. They hold for *every* polytypically defined function. This brings about a certain level of abstractness. It is likely that the reader will only fully acknowledge the value of these laws in Section 6 when we take a look at several examples. The first fusion law states conditions under which we can fuse a monotypic and a polytypic function, that is, $\psi \circ \mathit{poly}_n\langle\!\langle G \rangle\!\rangle\ \boldsymbol{\varphi} = \mathit{poly}'_n\langle\!\langle G \rangle\!\rangle\ (\psi \circ \boldsymbol{\varphi})$ where $\psi \circ \boldsymbol{\varphi}$ is given by $\psi \circ (\varphi_1, \ldots, \varphi_n) = (\psi \circ \varphi_1, \ldots, \psi \circ \varphi_n)$.

PROPOSITION 2 (MONO-POLY FUSION) Let poly_n and poly'_n be two polytypic functions and let ψ be a strict function. If $\psi \circ \mathit{poly}_P\ \boldsymbol{\varphi} = \mathit{poly}'_P\ (\psi \circ \boldsymbol{\varphi})$ holds for all primitive functors $P \in \mathbb{P}$, then

$$\psi \circ \mathit{poly}_n\langle\!\langle G \rangle\!\rangle\ \boldsymbol{\varphi} \quad = \quad \mathit{poly}'_n\langle\!\langle G \rangle\!\rangle\ (\psi \circ \boldsymbol{\varphi})$$

for all closed functor expressions G of arity n.

PROOF The structure of the proof is quite similar to the derivation of poly_n. First of all, to be able to use induction over the structure of functor expressions, we have to show a generalized statement: let ϱ and ϱ' be two environments such that $\psi \circ \varrho(X)\ \boldsymbol{\varphi} = \varrho'(X)\ (\psi \circ \boldsymbol{\varphi})$ for all free variables X of G, then $\psi \circ \mathit{poly}_n\langle\!\langle G \rangle\!\rangle \varrho\ \boldsymbol{\varphi} = \mathit{poly}'_n\langle\!\langle G \rangle\!\rangle \varrho'\ (\psi \circ \boldsymbol{\varphi})$.
Case $G = X$:

$$
\begin{array}{ll}
 & \psi \circ \mathit{poly}_n\langle\!\langle X \rangle\!\rangle \varrho\ \boldsymbol{\varphi} \\
= & \quad \{ \text{ definition of } \mathit{poly}_n \} \\
 & \psi \circ \varrho(X)\ \boldsymbol{\varphi} \\
= & \quad \{ \text{ environment condition } \} \\
 & \varrho'(X)\ (\psi \circ \boldsymbol{\varphi}) \\
= & \quad \{ \text{ definition of } \mathit{poly}'_n \} \\
 & \mathit{poly}'_n\langle\!\langle X \rangle\!\rangle \varrho'\ (\psi \circ \boldsymbol{\varphi}).
\end{array}
$$

Case $G = \Pi^n_i$:

$$\psi \circ \mathit{poly}_n\langle\!\langle \Pi^n_i \rangle\!\rangle \varrho\ \boldsymbol{\varphi}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ definition of } poly_n \ \} \\
& \psi \circ \varphi_i \\
= & \quad \{ \text{ definition of } poly'_n \ \} \\
& poly'_n \langle\!\langle \Pi_i^n \rangle\!\rangle \varrho' \ (\psi \circ \boldsymbol{\varphi}).
\end{array}
$$

Case $G = P$:

$$
\begin{array}{ll}
& \psi \circ poly_n \langle\!\langle P \rangle\!\rangle \varrho \ \boldsymbol{\varphi} \\
= & \quad \{ \text{ definition of } poly_n \ \} \\
& \psi \circ poly_P \ \boldsymbol{\varphi} \\
= & \quad \{ \text{ assumption } \} \\
& poly'_P \ (\psi \circ \boldsymbol{\varphi}) \\
= & \quad \{ \text{ definition of } poly'_n \ \} \\
& poly'_n \langle\!\langle P \rangle\!\rangle \varrho' \ (\psi \circ \boldsymbol{\varphi}).
\end{array}
$$

Case $G = H \cdot (H_1, \ldots, H_k)$:

$$
\begin{array}{ll}
& \psi \circ poly_n \langle\!\langle H \cdot (H_1, \ldots, H_k) \rangle\!\rangle \varrho \ \boldsymbol{\varphi} \\
= & \quad \{ \text{ definition of } poly_n \ \} \\
& \psi \circ poly_k \langle\!\langle H \rangle\!\rangle \varrho \ (poly_n \langle\!\langle H_1 \rangle\!\rangle \varrho \ \boldsymbol{\varphi}, \ldots, poly_n \langle\!\langle H_k \rangle\!\rangle \varrho \ \boldsymbol{\varphi}) \\
= & \quad \{ \text{ ex hypothesi } \} \\
& poly'_k \langle\!\langle H \rangle\!\rangle \varrho' \ (\psi \circ poly_n \langle\!\langle H_1 \rangle\!\rangle \varrho \ \boldsymbol{\varphi}, \ldots, \psi \circ poly_n \langle\!\langle H_k \rangle\!\rangle \varrho \ \boldsymbol{\varphi}) \\
= & \quad \{ \text{ ex hypothesi } \} \\
& poly'_k \langle\!\langle H \rangle\!\rangle \varrho' \ (poly'_n \langle\!\langle H_1 \rangle\!\rangle \varrho' \ (\psi \circ \boldsymbol{\varphi}), \ldots, poly'_n \langle\!\langle H_k \rangle\!\rangle \varrho' \ (\psi \circ \boldsymbol{\varphi})) \\
= & \quad \{ \text{ definition of } poly'_n \ \} \\
& poly'_n \langle\!\langle H \cdot (H_1, \ldots, H_k) \rangle\!\rangle \varrho' \ (\psi \circ \boldsymbol{\varphi})
\end{array}
$$

Case $G = (H \ \textbf{where} \ X_1 = H_1)$:

$$
\begin{array}{ll}
& \psi \circ poly_n \langle\!\langle H \ \textbf{where} \ X_1 = H_1 \rangle\!\rangle \varrho \ \boldsymbol{\varphi} \\
= & \quad \{ \text{ definition of } poly_n \ \} \\
& \psi \circ poly_n \langle\!\langle H \rangle\!\rangle \varrho (X_1 := \mathbf{lfp}(\lambda \psi_1 . poly_{k_1} \langle\!\langle H_1 \rangle\!\rangle \varrho (X_1 := \psi_1))) \ \boldsymbol{\varphi} \\
= & \quad \{ \text{ ex hypothesi and proof obligation } \} \\
& poly'_n \langle\!\langle H \rangle\!\rangle \varrho' (X_1 := \mathbf{lfp}(\lambda \psi'_1 . poly'_{k_1} \langle\!\langle H_1 \rangle\!\rangle \varrho' (X_1 := \psi'_1))) \ (\psi \circ \boldsymbol{\varphi}) \\
= & \quad \{ \text{ definition of } poly'_n \ \} \\
& poly'_n \langle\!\langle H \ \textbf{where} \ X_1 = H_1 \rangle\!\rangle \varrho' \ (\psi \circ \boldsymbol{\varphi})
\end{array}
$$

It remains to show that the extended environments satisfy the environment condition. If we define $\mathcal{P}$ by

$$
\mathcal{P}(p, p') \iff \psi \circ p \ \boldsymbol{\varphi} = p' \ (\psi \circ \boldsymbol{\varphi}),
$$

then we have to show that

$$\mathcal{P}(\mathbf{lfp}(\lambda\psi_1.poly_{k_1}\langle\!\langle H_1\rangle\!\rangle\varrho(X_1 := \psi_1)), \mathbf{lfp}(\lambda\psi_1'.poly'_{k_1}\langle\!\langle H_1\rangle\!\rangle\varrho'(X_1 := \psi_1'))).$$

The straightforward fixpoint induction is omitted (but note that $\mathcal{P}$ is pointed since ψ is strict). □

EXAMPLE 6 To illustrate the first fusion law let us show

$$\mathit{length} \circ \mathit{flatten}\langle F\rangle \quad = \quad \mathit{size}\langle F\rangle.$$

The polytypic function $\mathit{flatten}\langle F\rangle$, which listifies a given structure, is defined as follows

$$\begin{array}{lcl}
\mathit{flatten}\langle F\rangle & :: & \forall a.F\ a \to \mathit{List}\ a \\
\mathit{flatten}\langle \mathit{Id}\rangle & = & \mathit{wrap} \\
\mathit{flatten}\langle K\ T\rangle & = & \mathit{const}\ [\,] \\
\mathit{flatten}\langle F_1 + F_2\rangle & = & \mathit{flatten}\langle F_1\rangle \triangledown \mathit{flatten}\langle F_2\rangle \\
\mathit{flatten}\langle F_1 \times F_2\rangle & = & \mathit{cat} \circ (\mathit{flatten}\langle F_1\rangle \times \mathit{flatten}\langle F_2\rangle),
\end{array}$$

where $\mathit{wrap}\ a = [a]$ and $\mathit{cat}\ (x, y) = x +\!\!+ y$. By Proposition 2 we know

$$\mathit{length} \circ \mathit{flatten}_1\langle\!\langle F\rangle\!\rangle\ \mathit{wrap} \quad = \quad \mathit{size}_1\langle\!\langle F\rangle\!\rangle\ (\mathit{length} \circ \mathit{wrap}), \tag{3}$$

provided that

$$\begin{array}{lcl}
\mathit{length} \circ \mathit{const}\ [\,] & = & \mathit{const}\ 0 \\
\mathit{length} \circ (\varphi_1 \triangledown \varphi_2) & = & (\mathit{length} \circ \varphi_1) \triangledown (\mathit{length} \circ \varphi_2) \\
\mathit{length} \circ \mathit{cat} \circ (\varphi_1 \times \varphi_2) & = & \mathit{plus} \circ ((\mathit{length} \circ \varphi_1) \times (\mathit{length} \circ \varphi_2)).
\end{array}$$

All three conditions hold—the relevant laws are listed in the appendix. Noting that $\mathit{length} \circ \mathit{wrap} = \mathit{const}\ 1$ the proposition follows immediately from (3). □

The second polytypic fusion law states conditions under which a computation of two polytypic functions can be fused into a single polytypic function. Setting $(\varphi_1, \ldots, \varphi_n) \circ (\psi_1, \ldots, \psi_n) = (\varphi_1 \circ \psi_1, \ldots, \varphi_n \circ \psi_n)$, we have

PROPOSITION 3 (POLY-POLY FUSION) Let $poly_n$, $poly'_n$, and $poly''_n$ be three polytypic functions. If $poly_P\ \boldsymbol{\varphi} \circ poly'_P\ \boldsymbol{\psi} = poly''_P\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})$ holds for all primitive functors $P \in \mathbb{P}$, then

$$poly_n\langle\!\langle G\rangle\!\rangle\ \boldsymbol{\varphi} \circ poly'_n\langle\!\langle G\rangle\!\rangle\ \boldsymbol{\psi} \quad = \quad poly''_n\langle\!\langle G\rangle\!\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})$$

for all closed functor expressions G of arity n.

This law is proved in the same way as Proposition 2.

Applications of the second fusion law will be given in the following section.

6 Examples

This section presents further examples of polytypic functions and associated polytypic properties. Note that most of these functions arise as generalizations of corresponding list processing functions.

6.1 Mapping Functions

In categorical terms a functor is the combination of a parameterized type and a mapping function, which describes the action of the functor on functions. The mapping function of a unary functor F applies a given function to each element of type a in a given structure of type $F\ a$. The polytypic variant of the mapping function is given by[1]

$$\begin{array}{lcl} fmap\langle F\rangle & :: & \forall a\ a'.(a \to a') \to (F\ a \to F\ a') \\ fmap\langle F\rangle\ \varphi & = & map\langle F\rangle \\ \quad \textbf{where}\ map\langle F\rangle & :: & F\ a \to F\ a' \\ \qquad\quad map\langle Id\rangle & = & \varphi \\ \qquad\quad map\langle K\ T\rangle & = & id \\ \qquad\quad map\langle F_1 + F_2\rangle & = & map\langle F_1\rangle + map\langle F_2\rangle \\ \qquad\quad map\langle F_1 \times F_2\rangle & = & map\langle F_1\rangle \times map\langle F_2\rangle. \end{array}$$

It is revealing to consider the typings of the subsidiary functions map_n.

$$\begin{array}{lcl} map_n\langle\!\langle G\rangle\!\rangle & :: & \forall T_1\ \ldots\ T_n.(T_1\ a \to T_1\ a', \ldots, T_n\ a \to T_n\ a') \\ & & \quad \to (G\ (T_1\ a)\ \ldots\ (T_n\ a) \to G\ (T_1\ a')\ \ldots\ (T_n\ a')) \end{array}$$

Replacing $T_i\ a$ by a_i and $T_i\ a'$ by a_i' we obtain the typings of n-ary mapping functions. And, in fact, $map_n\langle\!\langle G\rangle\!\rangle$ corresponds to the mapping function of the n-ary functor G. Thus, to define *fmap* generically for all unary functors, mapping functions of functors of arbitrary arity are required. The good news is that the programmer need not take care of their definition. Instead, they are generated automatically.

The mapping function of a unary functor is required to satisfy the following two functorial properties.

$$\begin{array}{lcl} fmap\langle F\rangle\ id & = & id \\ fmap\langle F\rangle\ (\varphi \circ \psi) & = & fmap\langle F\rangle\ \varphi \circ fmap\langle F\rangle\ \psi. \end{array}$$

The first property can be shown using fixpoint induction. Only the proof of $fmap\langle\Omega\rangle\ id = id$ requires a bit of fudging. We have that $fmap\langle\Omega\rangle = \bot$ so we must in effect show that $\bot = id$. This equation holds, however, since Ω accommodates only the bottom element (recall that we are working in a cpo setting). The second property is an instance of the second polytypic fusion

[1] We assume that type variables appearing in type signatures are scoped, that is, the type variables a and b in the signature of $map\langle F\rangle$ are *not* universally quantified but refer to the occurrences in $fmap\langle F\rangle$'s signature.

law. Setting $poly_n = poly'_n = poly''_n = map_n$, we must establish $map_P\ \boldsymbol{\varphi} \circ map_P\ \boldsymbol{\psi} = map_P\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})$ for $map_{K\,T} = id$, $map_+(\varphi_1, \varphi_2) = \varphi_1 + \varphi_2$, and $map_\times(\varphi_1, \varphi_2) = \varphi_1 \times \varphi_2$. All of these conditions hold.

So far we have excluded types that contain function spaces. Perhaps surprisingly, our approach to polytypic programming works equally well if $\mathbb{P}^2$ additionally includes the function space constructor. We have only omitted '$\to$' because most of the polytypic functions cannot sensibly be defined for the function space. For instance, *fmap* cannot be defined for functional types since '$\to$' is contravariant in its first argument:

$$\begin{array}{lcl} (\to) & :: & (a' \to a) \to (b \to b') \to ((a \to b) \to (a' \to b')) \\ (\varphi_1 \to \varphi_2)\ h & = & \varphi_2 \circ h \circ \varphi_1. \end{array}$$

Now, drawing from the theory of embeddings and projections [9] we can remedy the situation as follows. The central idea is to supply a pair of functions $(\imath, \pi)$, where π is the left-inverse of $\imath$, that is, $\pi \circ \imath = id$. If the functions additionally satisfy $\imath \circ \pi \sqsubseteq id$, then $(\imath, \pi)$ is called an *embedding-projection pair*. Given this notion we can define a variant of *fmap*, which additionally works for the function space constructor:

$$\begin{array}{lcl} mapE\langle F\rangle & :: & \forall a\ a'.(a \to a', a' \to a) \to (F\ a \to F\ a') \\ mapE\langle Id\rangle\ \varphi & = & apply\ \varphi \\ mapE\langle K\,T\rangle\ \varphi & = & id \\ mapE\langle F_1 + F_2\rangle\ \varphi & = & mapE\langle F_1\rangle\ \varphi + mapE\langle F_2\rangle\ \varphi \\ mapE\langle F_1 \times F_2\rangle\ \varphi & = & mapE\langle F_1\rangle\ \varphi \times mapE\langle F_2\rangle\ \varphi \\ mapE\langle F_1 \to F_2\rangle\ \varphi & = & mapE\langle F_1\rangle\ \varphi^\circ \to mapE\langle F_2\rangle\ \varphi, \end{array}$$

where $apply\ (\imath, \pi) = \imath$ and $(\imath, \pi)^\circ = (\pi, \imath)$. Now, assume that φ is an embedding-projection pair, then we can prove

$$mapE\langle G\rangle\ \varphi^\circ \circ mapE\langle G\rangle\ \varphi \quad = \quad id$$

by fixpoint induction. We confine ourselves to the interesting cases.
Case $G = Id$:

$$\begin{array}{ll} & mapE\langle Id\rangle\ \varphi^\circ \circ mapE\langle Id\rangle\ \varphi \\ = & \quad \{\ \text{definition of } mapE\ \} \\ & apply\ \varphi^\circ \circ apply\ \varphi \\ = & \quad \{\ \text{definition of } apply \text{ and definition of } \varphi^\circ\ \} \\ & \pi \circ \imath = id \\ = & \quad \{\ \varphi \text{ is an embedding-projection pair}\ \} \\ & id. \end{array}$$

Case $G = F_1 \to F_2$:

$$mapE\langle F_1 \to F_2\rangle\ \varphi^\circ \circ mapE\langle F_1 \to F_2\rangle\ \varphi$$

$$
\begin{array}{ll}
= & \quad \{ \text{ definition of } \mathit{mapE} \ \} \\
& (\mathit{mapE}\langle F_1\rangle\ (\varphi^\circ)^\circ \to \mathit{mapE}\langle F_2\rangle\ \varphi^\circ) \circ (\mathit{mapE}\langle F_1\rangle\ \varphi^\circ \to \mathit{mapE}\langle F_2\rangle\ \varphi) \\
= & \quad \{ \text{ definition of } `\to' \ \} \\
& \lambda h.\mathit{mapE}\langle F_2\rangle\ \varphi^\circ \circ (\mathit{mapE}\langle F_2\rangle\ \varphi \circ h \circ \mathit{mapE}\langle F_1\rangle\ \varphi^\circ) \circ \mathit{mapE}\langle F_1\rangle\ (\varphi^\circ)^\circ \\
= & \quad \{ \text{ ex hypothesi and } (\varphi^\circ)^\circ = \varphi \ \} \\
& \mathit{id}.
\end{array}
$$

Using similar calculations we can furthermore show that

$$\mathit{mapE}\langle G\rangle\ \varphi \circ \mathit{mapE}\langle G\rangle\ \varphi^\circ \quad \sqsubseteq \quad \mathit{id}.$$

Both laws imply that $(\mathit{mapE}\langle G\rangle\ \varphi, \mathit{mapE}\langle G\rangle\ \varphi^\circ)$ is again an embedding-projection pair. Finally, one can show that $\lambda\varphi \to (\mathit{mapE}\langle G\rangle\ \varphi, \mathit{mapE}\langle G\rangle\ \varphi^\circ)$ is the functorial action of G in the category CPO^E of complete partial orders and embedding-projection pairs.

Another variant of *fmap* is the so-called *monadic map* [8]. Before we discuss its definition let us briefly review the basics of monads. For a more in-depth treatment we refer the interested reader to P. Wadler's papers [24, 25, 26]. One can think of a monad as an abstract type for computations. In Haskell monads are captured by the following class declaration.

```
class Monad m where
  return  ::  a → m a
  (>>=)   ::  m a → (a → m b) → m b
```

The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of $m\ a$ represents a computation that yields a value of type a. A computation may involve, for instance, state, exceptions, or nondeterminism. The trivial computation that immediately returns the value a is denoted *return a*. The operator $(\gg\!=)$ combines two computations: $m \gg\!= k$ applies k to the result of the computation m. Several datatypes have a computational content. For instance, the type *Maybe* given by

```
data Maybe a  =  Nothing | Just a
```

can be used to model exceptions: *Just x* represents a 'normal' or successful computation yielding the value x while *Nothing* represents an exceptional or failing computation. The following instance declaration shows how to define *return* and $(\gg\!=)$ for *Maybe*.

```
instance Monad Maybe where
  return           =  Just
  Nothing >>= k    =  Nothing
  Just a  >>= k    =  k a
```

Thus, $m \gg\!= k$ can be seen as a strict postfix application: if m is an exception, the exception is propagated; if m succeeds, then k is applied to the result.

The definition of *fmap* makes use of the combining forms '+' and '×'. The monadic mapping function can be succinctly defined if we raise these combinators to *procedures*, where a procedure is a function that maps values to computations. In other words, a procedure is a function of type $a \to M\ b$ where M is a monad.

$$
\begin{array}{lcl}
mmap & :: & (Monad\ m) \Rightarrow (a \to a') \to (m\ a \to m\ a') \\
mmap\ \varphi\ m & = & m \gg\!= (return \circ \varphi) \\
(\boxplus) & :: & (Monad\ m) \Rightarrow (a \to m\ a') \to (b \to m\ b') \\
 & & \qquad \to (a + b \to m\ (a' + b')) \\
(h_1 \boxplus h_2)\ (Inl\ x_1) & = & mmap\ Inl\ (h_1\ x_1) \\
(h_1 \boxplus h_2)\ (Inr\ x_2) & = & mmap\ Inr\ (h_2\ x_2) \\
(\boxtimes) & :: & (Monad\ m) \Rightarrow (a \to m\ a') \to (b \to m\ b') \\
 & & \qquad \to (a \times b \to m\ (a' \times b')) \\
(h_1 \boxtimes h_2)\ (x_1, x_2) & = & h_1\ x_1 \gg\!= \lambda y_1 \to h_2\ x_2 \gg\!= \lambda y_2 \to return\ (y_1, y_2)
\end{array}
$$

Given these combinators the definition of the monadic map is straightforward.

$$
\begin{array}{lcl}
mapMl\langle F \rangle & :: & \forall a\ a'.(Monad\ m) \Rightarrow (a \to m\ a') \to (F\ a \to m\ (F\ a')) \\
mapMl\langle Id \rangle\ \varphi & = & \varphi \\
mapMl\langle K\ T \rangle\ \varphi & = & return \\
mapMl\langle F_1 + F_2 \rangle\ \varphi & = & mapMl\langle F_1 \rangle\ \varphi \boxplus mapMl\langle F_2 \rangle\ \varphi \\
mapMl\langle F_1 \times F_2 \rangle\ \varphi & = & mapMl\langle F_1 \rangle\ \varphi \boxtimes mapMl\langle F_2 \rangle\ \varphi
\end{array}
$$

Note that the letter '*l*' in *mapMl* indicates that the structure $F\ a$ is traversed from left to right. The dual function, *mapMr*, is defined completely analogously except that in the definition of '⊠' the computations $h_1\ x_1$ and $h_2\ x_2$ are reversed. For applications of monadic maps we refer the interested reader to the tutorial of E. Meijer and J. Jeuring [21].

6.2 Reductions

The functions *size*, *sum*, and *flatten* are instances of a more general concept, due to L. Meertens [19], termed *reduction* or *crush*. A reduction is a function of type $F\ a \to a$ that collapses a structure of values of type a into a single value of type a. To define a reduction two ingredients are required: a value $e :: a$ and a binary operation $op :: a \to a \to a$. Usually but not necessarily e is the neutral element of *op*.

$$
\begin{array}{lcl}
reduce\langle F \rangle & :: & \forall a.a \to (a \to a \to a) \to (F\ a \to a) \\
reduce\langle F \rangle\ e\ op & = & red\langle F \rangle \\
\quad \textbf{where}\ red\langle F \rangle & :: & F\ a \to a \\
\qquad red\langle Id \rangle\ x & = & x \\
\qquad red\langle K\ T \rangle\ x & = & e \\
\qquad red\langle F_1 + F_2 \rangle\ (Inl\ x_1) & = & red\langle F_1 \rangle\ x_1 \\
\qquad red\langle F_1 + F_2 \rangle\ (Inr\ x_2) & = & red\langle F_2 \rangle\ x_2 \\
\qquad red\langle F_1 \times F_2 \rangle\ (x_1, x_2) & = & red\langle F_1 \rangle\ x_1\ `op`\ red\langle F_2 \rangle\ x_2
\end{array}
$$

$$
\begin{array}{lcl}
sum\langle F\rangle & :: & \forall n.(Num\ n) \Rightarrow F\ n \to n \\
sum\langle F\rangle & = & reduce\langle F\rangle\ 0\ (+) \\
and\langle F\rangle & :: & F\ Bool \to Bool \\
and\langle F\rangle & = & reduce\langle F\rangle\ True\ (\wedge) \\
minimum\langle F\rangle & :: & \forall a.(Bounded\ a, Ord\ a) \Rightarrow F\ a \to a \\
minimum\langle F\rangle & = & reduce\langle F\rangle\ maxBound\ min \\
size\langle F\rangle & :: & \forall a\ n.(Num\ n) \Rightarrow F\ a \to n \\
size\langle F\rangle & = & sum\langle F\rangle \circ fmap\langle F\rangle\ (const\ 1) \\
all\langle F\rangle & :: & \forall a.(a \to Bool) \to (F\ a \to Bool) \\
all\langle F\rangle\ p & = & and\langle F\rangle \circ fmap\langle F\rangle\ p \\
flatten\langle F\rangle & :: & \forall a.F\ a \to List\ a \\
flatten\langle F\rangle & = & reduce\langle F\rangle\ [\,]\ (+\!\!+) \circ fmap\langle F\rangle\ wrap \\
\textbf{data}\ Tree\ a & = & Empty \mid Leaf\ a \mid Fork\ (Tree\ a)\ (Tree\ a) \\
shape\langle F\rangle & :: & \forall a.F\ a \to Tree\ a \\
shape\langle F\rangle & = & reduce\langle F\rangle\ Empty\ Fork \circ fmap\langle F\rangle\ Leaf
\end{array}
$$

Figure 2: Examples of reductions.

Using a point free style we can define the helper function *red* more succinctly.

$$
\begin{array}{lcl}
red\langle Id\rangle & = & id \\
red\langle K\,T\rangle & = & const\ e \\
red\langle F_1 + F_2\rangle & = & red\langle F_1\rangle \triangledown red\langle F_2\rangle \\
red\langle F_1 \times F_2\rangle & = & uncurry\ op \circ (red\langle F_1\rangle \times red\langle F_2\rangle)
\end{array}
$$

A number of useful functions can be implemented in terms of *reduce* and *fmap*, see Figure 2. L. Meertens [19], P. Jansson and J. Jeuring [16] give further applications.

Again, it is interesting to inspect the subsidiary functions $reduce_n$ given by $reduce_n\langle\!\langle G\rangle\!\rangle\ e\ op = red_n\langle\!\langle G\rangle\!\rangle$ more closely. The following property

$$
reduce_n\langle\!\langle G\rangle\!\rangle\ e\ op\ (\varphi \circ \psi) \quad = \quad reduce_n\langle\!\langle G\rangle\!\rangle\ e\ op\ \varphi \circ map_n\langle\!\langle G\rangle\!\rangle\ \psi,
$$

which is an immediate consequence of Proposition 3, shows that $reduce_n$ combines a reduction with a map. This idiom is, in fact, a very old one. It appears, for instance, in R. Bird's lectures [6] where it is shown that each homomorphism on lists (that is, each catamorphism) can be expressed in this form. In a sense, $reduce_n$ can be seen as a generalization of that scheme.

Specializing Proposition 2 gives an important fusion law for reductions. Let ψ be a strict function, then

$$
\begin{array}{ll}
 & \psi\ e = e' \wedge \psi\ (op\ x_1\ x_2) = op'\ (\psi\ x_1)\ (\psi\ x_2) \\
\Longrightarrow & \psi \circ reduce_n\langle\!\langle G\rangle\!\rangle\ e\ op\ \varphi = reduce_n\langle\!\langle G\rangle\!\rangle\ e'\ op'\ (\psi \circ \varphi).
\end{array}
$$

An immediate consequence of the two fusion laws for reductions is, for instance, $\mathit{length} \circ \mathit{flatten}\langle F\rangle = \mathit{size}\langle F\rangle$, see Example 6.

The implementation of *flatten* given in Figure 2 has a quadratic running time since the computation of $x +\!\!+ y$ takes time proportional to the length of x. Using the well-known technique of *accumulation* [2] we can improve the running time to $O(n)$. The basic idea is to define a function $\mathit{redr}\langle F\rangle$ such that

$$\mathit{op}\ (\mathit{red}\langle F\rangle\ x)\ a \quad = \quad \mathit{redr}\langle F\rangle\ x\ a.$$

In a point-free style this condition can be written more succinctly as

$$\mathit{op} \circ \mathit{red}\langle F\rangle \quad = \quad \mathit{redr}\langle F\rangle,$$

which suggests to apply Proposition 2 for deriving a definition of $\mathit{redr}\langle F\rangle$. The calculation, which is left as an exercise to the reader, shows that the following definition improves $\mathit{reduce}\langle F\rangle$ provided *op* is associative and e is the neutral element of *op*.

$$\begin{array}{lcl}
\mathit{reduce}'\langle F\rangle & :: & \forall a.a \to (a \to a \to a) \to (F\ a \to a) \\
\mathit{reduce}'\langle F\rangle\ e\ \mathit{op}\ x & = & \mathit{redr}\langle F\rangle\ x\ e \\
\quad \textbf{where}\ \mathit{redr}\langle F\rangle & :: & F\ a \to (a \to a) \\
\qquad \mathit{redr}\langle \mathit{Id}\rangle & = & \mathit{op} \\
\qquad \mathit{redr}\langle K\ T\rangle & = & \mathit{const}\ \mathit{id} \\
\qquad \mathit{redr}\langle F_1 + F_2\rangle & = & \mathit{redr}\langle F_1\rangle \triangledown \mathit{redr}\langle F_2\rangle \\
\qquad \mathit{redr}\langle F_1 \times F_2\rangle & = & \mathit{uncurry}\ (\circ) \circ (\mathit{redr}\langle F_1\rangle \times \mathit{redr}\langle F_2\rangle)
\end{array}$$

The implementation guarantees that applications of *op* are only nested to the right as in $\mathit{op}\ a_1\ (\mathit{op}\ a_2\ (\ldots\ (\mathit{op}\ a_n\ e)\ \ldots))$—hence the name of the auxiliary function. This property also reveals that the type of $\mathit{reduce}'\langle F\rangle$ is unnecessarily restricted: the two arguments of *op* need not have the same type. Therefore, we may generalize the type signature as follows.

$$\begin{array}{lcl}
\mathit{reducer}\langle F\rangle & :: & \forall a\ b.(a \to (b \to b)) \to (F\ a \to (b \to b)) \\
\mathit{reducer}\langle F\rangle\ \mathit{op} & = & \mathit{redr}\langle F\rangle \\
\quad \textbf{where}\ \mathit{redr}\langle F\rangle & :: & F\ a \to (b \to b) \\
\qquad \mathit{redr}\langle \mathit{Id}\rangle & = & \mathit{op} \\
\qquad \mathit{redr}\langle K\ T\rangle & = & \mathit{const}\ \mathit{id} \\
\qquad \mathit{redr}\langle F_1 + F_2\rangle & = & \mathit{redr}\langle F_1\rangle \triangledown \mathit{redr}\langle F_2\rangle \\
\qquad \mathit{redr}\langle F_1 \times F_2\rangle & = & \mathit{uncurry}\ (\circ) \circ (\mathit{redr}\langle F_1\rangle \times \mathit{redr}\langle F_2\rangle)
\end{array}$$

Note that we have rearranged the arguments to emphasize the structure. Building upon $\mathit{reducer}\langle F\rangle$ we can now give a linear-time program for $\mathit{flatten}\langle F\rangle$.

$$\begin{array}{lcl}
\mathit{flatten}\langle F\rangle & :: & \forall a.F\ a \to \mathit{List}\ a \\
\mathit{flatten}\langle F\rangle\ x & = & \mathit{reducer}\langle F\rangle\ (:)\ x\ [\,]
\end{array}$$

Finally, note that *reducer* is equivalent to an *fmap* followed by an 'ordinary' reduction. We have

$$\mathit{reducer}\langle F\rangle\ \mathit{op} \quad = \quad \mathit{reduce}\langle F\rangle\ (\circ)\ \mathit{id} \circ \mathit{fmap}\langle F\rangle\ \mathit{op},$$

which can be shown using Proposition 3.

6.3 Equality and Comparison Functions

The equality function is a nice example of a function that is indexed by a nullary functor. In Haskell it can be automatically derived for datatypes of first-order kind and the following can be seen as a formalization of that process. We assume that a suitable equality function for *Int* is predefined.

$$
\begin{array}{lcl}
eq\langle T\rangle & :: & T \to T \to Bool \\
eq\langle K1\rangle\ x\ y & = & True \\
eq\langle KInt\rangle\ x\ y & = & eqInt\ x\ y \\
eq\langle T_1 + T_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = & eq\langle T_1\rangle\ x_1\ y_1 \\
eq\langle T_1 + T_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = & False \\
eq\langle T_1 + T_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = & False \\
eq\langle T_1 + T_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = & eq\langle T_2\rangle\ x_2\ y_2 \\
eq\langle T_1 \times T_2\rangle\ (x_1, x_2)\ (y_1, y_2) & = & eq\langle T_1\rangle\ x_1\ y_1 \wedge eq\langle T_2\rangle\ x_2\ y_2
\end{array}
$$

Note that *eq* unlike *fmap* and *reduce* cannot be defined in a uniform way for all constant functors. Functions that are indexed by nullary functors typically require a case analysis on constant functors.

Varying the definition of *eq* slightly we can also realize Haskell's *compare* function, which determines the precise ordering of two elements.

$$
\begin{array}{lcl}
\textbf{data}\ Ordering & = & LT \mid EQ \mid GT \\
cmp\langle T\rangle & :: & T \to T \to Ordering \\
cmp\langle K1\rangle\ x\ y & = & EQ \\
cmp\langle KInt\rangle\ x\ y & = & cmpInt\ x\ y \\
cmp\langle T_1 + T_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = & cmp\langle T_1\rangle\ x_1\ y_1 \\
cmp\langle T_1 + T_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = & LT \\
cmp\langle T_1 + T_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = & GT \\
cmp\langle T_1 + T_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = & cmp\langle T_2\rangle\ x_2\ y_2 \\
cmp\langle T_1 \times T_2\rangle\ (x_1, x_2)\ (y_1, y_2) & = & \textbf{case}\ cmp\langle T_1\rangle\ x_1\ y_1\ \textbf{of} \\
 & & \quad \{ EQ \to cmp\langle T_2\rangle\ x_2\ y_2; ord \to ord \}
\end{array}
$$

6.4 Zipping Functions

Closely related to mapping functions are zipping functions. A zipping function takes a pair of structures and returns a structure of pairs. If the argument structures have not the same shape, the zipping function returns *Nothing* (which we abbreviate by *fail*); otherwise it yields *Just z* where *z* is the desired structure. In other words, the zipping function uses the exception monad *Maybe* to signal incompatibility of argument structures. The definition of *zip* is similar to that

of *eq* and *cmp*.

$$
\begin{array}{lcl}
zip\langle F\rangle & :: & \forall a\ b.F\ a \to F\ b \to Maybe\ (F\ (a,b)) \\
zip\langle K1\rangle\ x\ y & = & return\ () \\
zip\langle KInt\rangle\ x\ y & = & \textbf{if}\ eqInt\ x\ y\ \textbf{then}\ return\ x\ \textbf{else}\ fail \\
zip\langle Id\rangle\ x\ y & = & return\ (x,y) \\
zip\langle F_1 + F_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = & mmap\ Inl\ (zip\langle F_1\rangle\ x_1\ y_1) \\
zip\langle F_1 + F_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = & fail \\
zip\langle F_1 + F_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = & fail \\
zip\langle F_1 + F_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = & mmap\ Inr\ (zip\langle F_2\rangle\ x_2\ y_2) \\
zip\langle F_1 \times F_2\rangle\ (x_1,x_2)\ (y_1,y_2) & = & zip\langle F_1\rangle\ x_1\ y_1 \gg\!= \lambda z_1 \to \\
 & & zip\langle F_2\rangle\ x_2\ y_2 \gg\!= \lambda z_2 \to \\
 & & return\ (z_1,z_2)
\end{array}
$$

Note that the type of *zip* could be generalized to arbitrary monads that contain a zero element such as *fail*.

7 Related and Future Work

This article can be regarded as a successor to my previous work on polytypic programming [10], where a similar approach using *rational trees* is presented. The major difference between the two frameworks lies in the treatment of functor composition. In the 'rational tree approach' functor composition is regarded as a function symbol, which implies that a polytypic definition must specify the action on $G \cdot (G_1, \ldots, G_n)$ for each $n \geqslant 1$. Clearly, this cannot be done exhaustively. Furthermore, since the cases for functor composition are redundant, there is no guarantee that the polytypic function behaves the same on equal functors such as *Rose* and *Rose'*. Both problems disappear if we handle functor composition on the meta level generalizing rational trees to algebraic trees.

The classic approach to polytypic programming as realized in the polytypic programming language extension PolyP [14] is based on the initial algebra semantics of datatypes. Here, functors are modeled by the following grammar.

$$
\begin{array}{lcl}
\mathbb{F} & ::= & \mu\mathbb{B} \\
\mathbb{B} & ::= & K\,T \mid Fst \mid Snd \mid \mathbb{B} + \mathbb{B} \mid \mathbb{B} \times \mathbb{B} \mid \mathbb{F} \cdot \mathbb{B}
\end{array}
$$

Recursive datatypes are modeled by fixpoints of associated base functors: the functor μB, which is known as a *type functor*, denotes the unary functor F given as the least solution of the equation $F\ a = B(a, F\ a)$. Polytypic functions are defined according to the above structure of types. In PolyP the polytypic

function $size\langle F\rangle$ is defined as follows—modulo change of notation.

$$
\begin{array}{lcl}
size\langle F\rangle & :: & \forall a.F\ a \to Int \\
size\langle \mu B\rangle & = & cata\langle \mu B\rangle\ (bsize\langle B\rangle) \\
bsize\langle B\rangle & :: & \forall a.B\ a\ Int \to Int \\
bsize\langle K\ T\rangle\ x & = & 0 \\
bsize\langle Fst\rangle\ x & = & 1 \\
bsize\langle Snd\rangle\ n & = & n \\
bsize\langle B_1 + B_2\rangle\ (Inl\ x_1) & = & bsize\langle B_1\rangle\ x_1 \\
bsize\langle B_1 + B_2\rangle\ (Inr\ x_2) & = & bsize\langle B_2\rangle\ x_2 \\
bsize\langle B_1 \times B_2\rangle\ (x_1, x_2) & = & bsize\langle B_1\rangle\ x_1 + bsize\langle B_2\rangle\ x_2 \\
bsize\langle F \cdot B\rangle\ x & = & sum\langle F\rangle\ (fmap\langle F\rangle\ (bsize\langle B\rangle)\ x)
\end{array}
$$

The program is quite elaborate as compared to the one given in Section 1: it involves two general combining forms, *cata* and *fmap*, and an auxiliary polytypic function, *sum*. The disadvantages of the initial algebra approach are fairly obvious. The above definition is redundant: we know that *size* is uniquely defined by its action on constant functors, *Id*, sums, and products. The definition is incomplete: *size* is only applicable to regular functors (recall that, for instance, *Perfect* is not a regular functor). Furthermore, the regular functor may not depend on functors of arity $\geqslant 2$ since functor composition is only defined for unary functors. Finally, the definition exhibits a slight inefficiency: the combing form *fmap* produces an intermediate data structure, which is immediately consumed by *sum*. In other words, $size\langle Rose\rangle$ corresponds to the first, less efficient definition of *sizer*.

Unlike the framework we have presented so far PolyP allows the programmer to define general recursion schemes like cata- and anamorphisms [20]. As an example, the recursion scheme *cata*, which is used in *size*, is given by

$$
\begin{array}{lcl}
cata\langle F\rangle & :: & \forall a\ b.(F'\ a\ b \to b) \to (F\ a \to b) \\
cata\langle \mu B\rangle\ \varphi & = & \varphi \circ bmap\langle B\rangle\ id\ (cata\langle \mu B\rangle\ \varphi) \circ out.
\end{array}
$$

The operation (-)′, which is called *FunctorOf* in PolyP, maps a type functor to its base functor: $F' = B$ for $F = \mu B$. The function $out :: F\ a \to F'\ a\ (F\ a)$ decomposes an element of type $F\ a$ by unfolding one level of recursion. While the explicit treatment of type recursion is unnecessary for many applications—all polytypic functions of PolyLib [16] can be implemented in our framework, except, of course, cata- and anamorphisms—it is indispensable for others. The polytypic unification algorithm described by P. Jansson and J. Jeuring [15], for instance, requires a function that determines the immediate subterms of a term. A similar function appears in the article of R. Bird, O. de Moor, and P. Hoogendijk [4], who present a generalization of the maximum segment sum problem. In both cases the recursive structure of a datatype must be known.

Now, since our framework deals with type recursion on the meta level, one could feel tempted to conclude that we cannot deal with such applications. It appears, however, that the availability of operations on types is an orthogonal

design issue. Hence, nothing prevents us from incorporating the operator $(\text{-})'$ depicted above. To this end we simply take over the type inference algorithm described by P. Jansson and J. Jeuring [14]. It is useful to generalize $(\text{-})'$ so that it maps an n-ary, regular functor to its associated $(n+1)$-ary base functor. Given two polytypic functions $in :: F'\ a_1 \ldots a_n\ (F\ a_1 \ldots a_n) \to F\ a_1 \ldots a_n$ and $out :: F\ a_1 \ldots a_n \to F'\ a_1 \ldots a_n\ (F\ a_1 \ldots a_n)$ we can define cata- and anamorphisms for functors of arbitrary arity. Here are the definitions for unary functors.

$$\begin{array}{lcl} cata\langle F\rangle & :: & \forall a\ b.(F'\ a\ b \to b) \to (F\ a \to b) \\ cata\langle F\rangle\ \varphi & = & \varphi \circ bmap\langle F'\rangle\ id\ (cata\langle F\rangle\ \varphi) \circ out \\ ana\langle F\rangle & :: & \forall a\ b.(b \to F'\ a\ b) \to (b \to F\ a) \\ ana\langle F\rangle\ \psi & = & in \circ bmap\langle F'\rangle\ id\ (ana\langle F\rangle\ \psi) \circ \psi \end{array}$$

It is worth noting that the definition of the subsidiary function *bmap* proceeds as before. In particular, there is no need to consider functor composition or type functors. Furthermore, the base functor may be a nested functor. The function that collects the immediate subterms of a term can be defined as follows.

$$\begin{array}{lcl} subterms\langle T\rangle & :: & T \to List\ T \\ subterms\langle T\rangle & = & flatten\langle T'\rangle \circ out \end{array}$$

A direction for future work suggests itself: it remains to broaden the approach to include higher-order polymorphism [18]. Currently, the author is working on an extension so that polytypic functions can be defined generically for all datatypes expressible in Haskell. First results are summarized in [12]. An application of polytypic programming to digital searching is described in a companion paper [11], where we show how to define *tries* and operations on tries generically for arbitrary datatypes of first-order kind. The central insight is that a trie can be considered as a *type-indexed datatype*, which adds an interesting new dimension to polytypic programming.

Acknowledgements

I would like to thank the four anonymous referees of FLOPS'99 and the three anonymous referees of this special issue for many valuable comments.

A Categorical Combinators

This appendix contains the definitions of the categorical combinators used in the main text. The notation is fairly standard so the cognoscenti should have no problems in reading the article. For background material the reader is referred to the textbook by R. Bird and O. de Moor [3].

Standard combinators *id* is the identity function, '$\circ$' denotes functional composition, and *const* creates a constant-valued function.

$$\begin{array}{lcl} id & :: & a \to a \\ id\ x & = & x \\ (\circ) & :: & (b \to c) \to (a \to b) \to (a \to c) \\ (f \circ g)\ x & = & f\ (g\ x) \\ const & :: & a \to b \to a \\ const\ x\ y & = & y \end{array}$$

Sums We treat sums as if they were given by the following datatype definition.

$$\begin{array}{lcl} \textbf{data}\ a + b & = & Inl\ a \mid Inr\ b \\ (\triangledown) & :: & (a \to c) \to (b \to c) \to (a + b \to c) \\ (f\ \triangledown\ g)\ (Inl\ x) & = & f\ x \\ (f\ \triangledown\ g)\ (Inr\ y) & = & g\ y \\ (+) & :: & (a \to a') \to (b \to b') \to (a + b \to a' + b') \\ f + g & = & (Inl \circ f)\ \triangledown\ (Inr \circ g) \end{array}$$

Elements of $a + b$ are synthesized using the injection functions *Inl* and *Inr*; they are analysed using the 'case' operator ($\triangledown$). The disjoint sum is a bifunctor; its mapping function is given by (+). The operators ($\triangledown$) and (+) satisfy a variety of properties, among others

$$\begin{array}{lcll} Inl\ \triangledown\ Inr & = & id & \\ (f\ \triangledown\ g) \circ Inl & = & f & \\ (f\ \triangledown\ g) \circ Inr & = & g & \\ h \circ (f\ \triangledown\ g) & = & (h \circ f)\ \triangledown\ (h \circ g) & \text{if } h \text{ is strict.} \end{array}$$

Note that, for fundamental reasons, '+' is not a categorical coproduct in non-strict languages such as Haskell, see, for instance [13].

Products Products are given by the following datatype.[2]

$$\begin{array}{lcl} \textbf{data}\ a \times b & = & (a, b) \\ outl & :: & a \times b \to a \\ outl\ (x, y) & = & x \\ outr & :: & a \times b \to b \\ outr\ (x, y) & = & y \\ (\triangledown) & :: & (c \to a) \to (c \to b) \to (c \to a \times b) \\ (f\ \triangle\ g)\ z & = & (f\ z, g\ z) \\ (\times) & :: & (a \to a') \to (b \to b') \to (a \times b \to a' \times b') \\ f \times g & = & (f \circ outl)\ \triangle\ (g \circ outr) \end{array}$$

[2] Note that in Haskell '$\times$' has an extra element $\bot$ such that $\bot \neq (\bot, \bot)$. We simply ignore this extra element.

Elements of $a \times b$ are analysed using the projection functions *outl* and *outr*; they are synthesized using the 'split' operator ($\triangle$). The product is a bifunctor, as well; its mapping function is given by ($\times$). The properties of ($\triangle$) and ($\times$) are dual to those of ($\triangledown$) and ($+$).

$$\begin{array}{lcl} outl \triangle outr & = & id \\ outl \circ (f \triangle g) & = & f \\ outr \circ (f \triangle g) & = & g \\ (f \triangle g) \circ h & = & (f \circ h) \triangle (g \circ h). \end{array}$$

Currying *curry* converts a non-curried function to curried form with *uncurry* as its inverse.

$$\begin{array}{lcl} curry & :: & ((a,b) \to c) \to (a \to b \to c) \\ curry\ f\ x\ y & = & f\ (x,y) \\ uncurry & :: & (a \to b \to c) \to ((a,b) \to c) \\ uncurry\ f\ (x,y) & = & f\ x\ y \end{array}$$

References

[1] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[2] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, London, 2nd edition, 1998.

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

[4] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, January 1996.

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

[6] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer-Verlag, 1988.

[7] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

[8] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report Memoranda Informatica 94-28, University of Twente, June 1994.

[9] G. Gierz, K.H. Hofmann, K Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, 1980.

[10] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(4):159–180, September 1999.

[11] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 2000. To appear.

[12] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

[13] Hagen Huwig and Axel Poigné. A note on inconsistencies caused by fixpoints in a Cartesian closed category. *Theoretical Computer Science*, 73(1):101–112, June 1990.

[14] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482. ACM-Press, January 1997.

[15] Patrik Jansson and Johan Jeuring. Functional Pearl: Polytypic unification. *Journal of Functional Programming*, 8(5), September 1998.

[16] Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden*. Department of Computing Science, Chalmers University of Technology and Göteborg University, June 1998.

[17] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *Proceedings European Symposium on Programming, ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287, Berlin, 1999. Springer-Verlag.

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

[19] Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1996.

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

[21] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *1st International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer-Verlag, Berlin, 1995.

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

[23] Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.

[24] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78. ACM-Press, June 1990.

[25] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico*, pages 1–14, January 1992.

[26] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, May 1995.

[27] Zhe Yang. Encoding types in ML-like languages. *SIGPLAN Notices*, 34(1):289–300, January 1999.