Science of Computer Programming 14 (1990) 255-279
North-Holland

DATA STRUCTURES AND PROGRAM TRANSFORMATION

Grant MALCOLM

Department of Computing Science, Groningen University, P.O. Box 800, 9700 AV Groningen, Netherlands

Received January 1990

Abstract. The construction of structure-preserving maps, "homomorphisms", is described for an arbitrary data type, and a "promotion" theorem is derived for proving equalities of homomorphisms. Examples are given for finite lists, tree structures and types defined by mutual induction; the construction is then dualised to data types with infinite objects, such as infinite lists. The promotion theorem allows the development of concise, calculational proofs: several examples are given of its application to program transformation.

Introduction

Our starting point in this paper is the "Bird-Meertens formalism", a mathematical framework for program transformation currently being developed by Richard Bird at Oxford and Lambert Meertens at Amsterdam. The formalism comprises a concise functional notation and a small number of remarkably powerful theorems for proving equalities of functions. The notation is based on a few elementary operations and is so designed that it lends itself well to the construction of elegant, calculational transformations: from the basic operations, one constructs a specification in algorithmic form—there is no separate specification language—and transforms this specification into a more efficient program. A hallmark of the formalism is that much of this transformation process can be conducted as a linear, equational proof that the original specification is extensionally equal to the resulting more efficient version.

Much of the early work on the Bird-Meertens formalism concentrates on functions on one data type, finite lists [3, 4, 17]; more recent work by Bird, Meertens and Jeuring [5, 12, 18] has applied the formalism to other data structures such as binary trees, arrays and "rose trees". It turns out that the basic operations and theorems used in the work on finite lists have analogous operations and theorems which are applicable to those other data structures, which raises the question as to how one should go about incorporating new data structures into the formalism. This is the question that we address in the present paper. We propose an algebraic notation for data structures based on that of Hagino [9], and closely related to the notion of initial data types (as, for example, to be found in Meseguer and Goguen [20]).

The notation has the advantage that it allows both finite and infinite data structures, and emphasises the duality between the two; moreover, it gives rise in a natural way to the notion of homomorphism, structure-preserving functions which we employ as the basic means of constructing recursive functions on a given data structure. The algebraic nature of the notation permits homomorphisms to be treated in an equational manner, and we exploit this in deriving, for an arbitrary data type, a "promotion theorem" for proving equalities of homomorphisms. The term "promotion" is due to Bird [4]; the promotion theorem for finite lists first appears in Meertens [17] and was independently discovered by Backhouse in [1].

We present the notation for finite data types in Section 2, and prove the promotion theorem for an arbitrary data type; in Section 3 we show by example how the promotion theorem can be extended to types defined by mutual induction, and in Section 4 we dualise to infinite data structures, obtaining a dual form of the promotion theorem. We give examples of promotion for several data types, showing how promotability allows the construction of—in our opinion—very elegant proofs. We begin by introducing two concepts that we make use of later on: type functors and natural transformations.

1. Type-functors and natural transformations

Definition 1.1 (*Type-functors*). A *type-functor* is a function from types to types that has a corresponding action on functions which respects identity and composition. That is, F is a type-functor if for all (sequences of) types A the application of F to A, denoted by $A\mathsf{F}$, is a type, and for all (sequences of) types A and B and (sequences of) functions $f \in A \to B$, there is a function $f\mathsf{F} \in A\mathsf{F} \to B\mathsf{F}$. Moreover, we require that $I\mathsf{F} = I \in A\mathsf{F} \to A\mathsf{F}$ and $(f \circ g)\mathsf{F} = f\mathsf{F} \circ g\mathsf{F}$. (Note that we use I to denote the polymorphic identity function.)

If A and B are types, we denote their cartesian product by $A \| B$: thus, $\|$ is a type-functor of two arguments (we use infix notation for its application) since for functions $f_1 \in A_1 \to B_1$ and $f_2 \in A_2 \to B_2$, we have $f_1 \| f_2 \in A_1 \| A_2 \to B_1 \| B_2$, the function that takes the pair $\langle a_1, a_2 \rangle$ to the pair $\langle f_1.a_1, f_2.a_2 \rangle$; moreover, $\|$ preserves identity and composition: $I \| I = I$, and

$$(f_1 \| f_2) \circ (g_1 \| g_2) = (f_1 \circ g_1) \| (f_2 \circ g_2). \tag{1}$$

Since we shall be using $\|$ quite extensively, we shall adopt the convention that it has higher priority than composition, so that we may write the left-hand side of the above equation without brackets, thus: $f_1 \| f_2 \circ g_1 \| g_2$.

While we are on the subject of cartesian product, we may as well introduce some more notation that will be used later on. If $f \in A \to B$ and $g \in A \to C$, then we write $\langle f, g \rangle$ for the function of type $A \to B \| C$ which takes $a \in A$ to $\langle f.a, g.a \rangle$. The function

enjoys the following compositional properties:

$$\langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle,$$

$$d \parallel e \circ \langle f, g\rangle = \langle d \circ f, e \circ g\rangle,$$

$$\mathsf{fst} \circ \langle f, g\rangle = f,$$

$$\mathsf{snd} \circ \langle f, g\rangle = g.$$

Moreover, $f \parallel g = \langle f \circ \mathsf{fst}, g \circ \mathsf{snd}\rangle$.

To return to type-functors, the simplest example is the identity functor I, which is such that $X\mathsf{I} = X$, whether X be a type or a function. Another example is $\lessdot$, a type-functor of two arguments, which always returns the left argument (we shall always use infix notation for binary operators) thus: $X \lessdot Y = X$, for X and Y types or functions.

We can make new type-functors from old. Composition of type-functors is denoted by juxtaposition: if F and G are type-functors, then their composition FG is also a type-functor, taking type A to $A\mathsf{FG} = (A\mathsf{F})\mathsf{G}$. The identity functor, I, is such that $\mathsf{IF} = \mathsf{F} = \mathsf{FI}$. If F and G are type-functors, and $\oplus$ is a type-functor of two arguments, then $\mathsf{F} \hat{\oplus} \mathsf{G}$ is a type-functor, defined by $A(\mathsf{F} \hat{\oplus} \mathsf{G}) = A\mathsf{F} \oplus A\mathsf{G}$. Moreover, $\mathsf{F}(\mathsf{G} \hat{\oplus} \mathsf{H}) = (\mathsf{FG}) \hat{\oplus} (\mathsf{FH})$.

Type-functors are related by natural transformations:

Definition 1.2 (*Natural transformations*). Let F and G be type-functors. A *natural transformation* from F to G is, for our purposes, a polymorphic function η such that for all types A, $\eta \in A\mathsf{F} \to A\mathsf{G}$ and for all types A and B and functions $f \in A \to B$,

$$\eta \circ f\mathsf{F} = f\mathsf{G} \circ \eta.$$

If η is a natural transformation from F to G, we write $\eta : \mathsf{F} \dot{\to} \mathsf{G}$.

A simple example is $\mathsf{fst} : \mathsf{F} \parallel \mathsf{G} \dot{\to} \mathsf{F}$; that is, for all f, $\mathsf{fst} \circ f\mathsf{F} \parallel f\mathsf{G} = f\mathsf{F} \circ \mathsf{fst}$.

If $\oplus \in A \parallel B \to C$ is a binary operator, and $x \in A$, then we write $x\oplus$ for the function of type $B \to C$ which takes y to $x \oplus y$. Similarly, the function $\oplus y$ takes x to $x \oplus y$. This notation is referred to as "sectioning" in [4]. We also use sectioning with type-functors: for example, if A is a type, then $A\lessdot$ is the constant functor returning A when its argument is a type and returning the identity on A when its argument is a function. That is,

$$B(A\lessdot) = A \quad \text{for types } B,$$

$$f(A\lessdot) = I_A \quad \text{for functions } f \in B \to C.$$

An interesting point is that all functions $f \in A \to B$ are natural transformations $f : A\lessdot \dot{\to} B\lessdot$.

2. Promotability for finite data types

We now come to the heart of the matter: defining promotability for an arbitrary data structure. In this section we define data structures with finite objects to be "initial" algebras, and from the property of initiality we derive the promotion theorem. We illustrate the use of promotion by proving some simple properties of two data structures: finite lists and rose trees. In Sections 3 and 4, we generalise the construction to allow types defined by mutual induction and also, via a process of dualisation, types with infinite objects, such as the type of infinite lists. In order to be able to discuss data structures in general, we adopt a notation due to Hagino [9]: Hagino's work has a categorical setting; Mendler, in his thesis [19], uses similar constructions for defining data types in the context of type theory and the second-order typed lambda calculus.

2.1. An algebraic notation for type structures

As in a typed programming language such as SML, or as in a type theory such as Martin-Löf's, a type is defined by giving its constructors and stating the types of these constructors. In this paper we shall consider the types of the constructors to be determined by a type-functor; thus if we define a type $\mathcal{T}$ by:

$$\mathcal{T} \triangleq \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n),$$

then we mean that the type $\mathcal{T}$ has n constructors τ_i, and the type of each constructor is determined by the type-functors F_i according to:

$$\tau_i \in \mathcal{T}\mathsf{F}_i \to \mathcal{T}.$$

For example the type of lists over a type A could be defined as:

$$A* \triangleq \mu(\mathsf{nil} : \mathbb{1} \lessdot, \succ\!\!- : \| A),$$

which states that the type $A*$ has two constructors: the empty list $\mathsf{nil} \in \mathbb{1} \to A*$ and concatenation $\succ\!\!- \in A* \| A \to A*$. We use $\mathbb{1}$ to denote the unit type having one element; as exemplified in the type of nil above, we associate constants $a \in A$ with constant-valued functions of type $\mathbb{1} \to A$, for any type A.

For the remainder of this section, we shall consider the arbitrary type $\mathcal{T}$ defined as above, and use the type $A*$ to provide examples.

The type $\mathcal{T}$ together with its constructors constitutes an algebra, a set with structure, and we shall exploit this structure in formulating a paradigm of inductive definitions of functions ("homomorphisms") on the type $\mathcal{T}$, and in deriving the "promotion theorem" for the type. First we formulate the induced algebra.

Definition 2.1 (*$\mathcal{T}$-algebras*). A *$\mathcal{T}$-algebra* is a tuple $(P, f_1, \ldots, f_n)$, where P is a type and for each i,

$$f_i \in P\mathsf{F}_i \to P.$$

If $(P, f_1, \ldots, f_n)$ and $(Q, g_1, \ldots, g_n)$ are $\mathcal{T}$-algebras, then a *$\mathcal{T}$-homomorphism*

$$h : (P, f_1, \ldots, f_n) \to (Q, g_1, \ldots, g_n)$$

is a function $h \in P \to Q$ such that

$$\forall (i :: h \circ f_i = g_i \circ h\mathsf{F}_i).$$

Clearly, $\mathcal{T}$-homomorphisms are closed under composition.

Example 2.2 (*A*-algebras*). *A**-algebras are triples $(P, d, \otimes)$, where $d \in \mathbb{1} \to P$ and $\otimes \in P \parallel A \to P$. An *A**-homomorphism $h : (P, d, \oplus) \to (Q, e, \oplus)$ is a function $h \in P \to Q$ such that:

$$(h \circ d = e) \ \wedge \ (h \circ \otimes = \oplus \circ h \parallel I).$$

These equations are expressed perhaps more familiarly as:

$$(h.d = e) \ \wedge \ \forall (x \in P, y \in A :: h.(x \otimes y) = h.x \oplus y).$$

In the above equations and henceforth we adopt the convention that application of unary functions (denoted by a dot) is more binding than infix application of binary operators.

Obviously, the type $\mathcal{T}$ together with its constructors is itself a $\mathcal{T}$-algebra; in fact, we complete the definition of the type $\mathcal{T}$ by ascribing to it the property of initiality:

Definition 2.3 (*Initial Hagino types*). The type $\mathcal{T} = \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n)$ is characterised ("up to isomorphism") by defining $(\mathcal{T}, \tau_1, \ldots, \tau_n)$ to be the *initial* $\mathcal{T}$-algebra; that is, for every $\mathcal{T}$-algebra $(Q, g_1, \ldots, g_n)$, there is exactly one $\mathcal{T}$-homomorphism

$$(g_1, \ldots, g_n)_{\mathcal{T}} : (\mathcal{T}, \tau_1, \ldots, \tau_n) \to (Q, g_1, \ldots, g_n).$$

The fact that this is a homomorphism gives the following equations, which prescribe the evaluation of the function:

$$\forall (i :: (g_1, \ldots, g_n)_{\mathcal{T}} \circ \tau_i = g_i \circ (g_1, \ldots, g_n)_{\mathcal{T}} \mathsf{F}_i). \tag{2}$$

When we regard this homomorphism as a function, its uniqueness is expressed by: for all h,

$$(h = (g_1, \ldots, g_n)_{\mathcal{T}}) \equiv \forall (i :: h \circ \tau_i = g_i \circ h\mathsf{F}_i). \tag{3}$$

We shall use the same notation $(\cdot)_{\mathcal{T}}$ for homomorphisms for all data types, and use the subscripted $\mathcal{T}$ to indicate which data type we intend. When no ambiguity seems likely, we drop the subscript.

The uniqueness of homomorphisms property gives us immediately the following useful identity:

Property 2.4 (Identity). $(\tau_1, \ldots, \tau_n) = I \in \mathcal{T} \to \mathcal{T}$.

Proof.

$$
\begin{array}{ll}
& (\tau_1, \ldots, \tau_n) = I \\
\equiv & \{(3)\} \\
& \forall(i :: I \circ \tau_i = \tau_i \circ I\mathsf{F}_i) \\
\equiv & \{\text{functors preserve identity}\} \\
& \text{true.} \quad \square
\end{array}
$$

Example 2.5 (*Snoc lists*). $(A*, \text{nil}, \succ\!\!-)$ is defined to be the initial $A*$-algebra. This means that for all types P with operations $d \in P$ and $\otimes \in P \parallel A \to P$, there is a unique function $(d, \otimes). \in A* \to P$ such that:

$$(d, \otimes).\text{nil} = d, \tag{4}$$

$$(d, \otimes) \circ \succ\!\!- = \otimes \circ (d, \otimes) \parallel I, \tag{5}$$

i.e., $(d, \otimes)$ is the unique $A*$-homomorphism $(A*, \text{nil}, \succ\!\!-) \to (P, d, \otimes)$. Thus homomorphisms provide a paradigm of recursive definition of functions, and the equations (4) and (5) specify how these functions are to be evaluated. From Property 2.4 we obtain that the identity homomorphism is $(\text{nil}, \succ\!\!-)$.

We can now state the promotion theorem for the type $\mathcal{T}$:

Theorem 2.6 (Promotion). *For a type* $\mathcal{T} = \mu(\tau_1 : \mathsf{F}_1, \ldots, \tau_n : \mathsf{F}_n)$,

$$\forall(i :: h \circ f_i = g_i \circ h\mathsf{F}_i) \Rightarrow (h \circ (f_1, \ldots, f_n) = (g_1, \ldots, g_n)).$$

Proof.

$$
\begin{array}{ll}
& h \circ (f_1, \ldots, f_n) = (g_1, \ldots, g_n) \\
\equiv & \{(3)\} \\
& \forall(i :: h \circ (f_1, \ldots, f_n) \circ \tau_i = g_i \circ (h \circ (f_1, \ldots, f_n))\mathsf{F}_i) \\
\equiv & \{(2), \text{functors respect composition}\} \\
& \forall(i :: h \circ f_i \circ (f_1, \ldots, f_n)\mathsf{F}_i = g_i \circ h\mathsf{F}_i \circ (f_1, \ldots, f_n)\mathsf{F}_i) \\
\Leftarrow & \{\text{congruence}\} \\
& \forall(i :: h \circ f_i = g_i \circ h\mathsf{F}_i). \quad \square
\end{array}
$$

Example 2.7 (*Snoc lists*). If $h \circ e = d$ and $h \circ \oplus = \otimes \circ h \parallel I$, then $h \circ (e, \oplus) = (d, \otimes)$. Or, equivalently,

$$(h \circ \oplus = \otimes \circ h \parallel I) \Rightarrow (h \circ (e, \oplus) = (h.e, \otimes)). \tag{6}$$

Whenever a promotion theorem can be simplified like this, we state and use only the simplified form.

If we pronounce the antecedent of formula (6) as "h is $\oplus/\otimes$-promotable", then we see from equation (5) that all homomorphisms $(\!|d, \otimes|\!)$ are $\rightarrowtail/\otimes$-promotable: this observation, which holds for all types, will prove useful later on. We shall give an illustration of this by using the results we have so far obtained to prove three simple properties of list-concatenation: that the empty list is the left- and right-unit of concatenation and that concatenation is associative. First we define concatenation:

Definition 2.8. For $x,y \in A*$, define $x +\!\!+ y \triangleq (\!|x, \rightarrowtail|\!).y$.

To prove that the empty list is the left-unit of $+\!\!+$:

$$
\begin{aligned}
& \mathsf{nil} +\!\!+ x \\
= \quad & \{\text{Definition 2.8}\} \\
& (\!|\mathsf{nil}, \rightarrowtail|\!).x \\
= \quad & \{\text{Property 2.4, identity}\} \\
& x.
\end{aligned}
$$

To prove it is the right-unit:

$$
\begin{aligned}
& x +\!\!+ \mathsf{nil} \\
= \quad & \{\text{Definition 2.8}\} \\
& (\!|x, \rightarrowtail|\!).\mathsf{nil} \\
= \quad & \{(4)\} \\
& x.
\end{aligned}
$$

And finally associativity, where we make use of the observation that all homomorphisms are promotable; in particular, that $(\!|x, \rightarrowtail|\!)$ is $\rightarrowtail/\rightarrowtail$-promotable:

$$
\begin{aligned}
& x +\!\!+ (y +\!\!+ z) \\
= \quad & \{\triangleq +\!\!+\} \\
& ((\!|x, \rightarrowtail|\!) \circ (\!|y, \rightarrowtail|\!)).z \\
= \quad & \{(5);\ \text{promotion}\} \\
& (\!|(\!|x, \rightarrowtail|\!).y, \rightarrowtail|\!).z \\
= \quad & \{\triangleq +\!\!+\} \\
& (x +\!\!+ y) +\!\!+ z.
\end{aligned}
$$

Of course, the proof that concatenation is associative is trivial: but the point is that the above proof proceeds very simply, and is entirely calculational, which makes it considerably shorter than an inductive proof would be. If nothing else, it is at least a case where the triviality of the proof matches the triviality of the proposition it proves. After all, no one would use induction to prove that $1+n-1=n$ for all natural numbers n: the calculation $1+n-1=n+1-1=n+0=n$ is simpler and more transparent. In the examples in the remainder of the present paper, we shall explore the use of promotion in providing a calculational alternative to inductive proof. To continue with snoc lists, we prove that $*$ is a type-functor; i.e., we show that for every $f \in A \rightarrow B$ there is a function $f* \in A* \rightarrow B*$ (the "map" of f, which applies f componentwise to each element of a given list) and we show that this operator respects identity and composition.

Definition 2.9 (*Map*). For $f \in A \to B$, define $f* \triangleq (\mathsf{nil}, \rightarrowtail \circ I \| f) \in A* \to B*$.

We begin with a lemma:

Property 2.10. $(e, \oplus) \circ f* = (e, \oplus \circ I \| f)$.

Proof.

$$
\begin{array}{ll}
 & (e, \oplus) \circ f* = (e, \oplus \circ I \| f) \\
\equiv & \{\text{Definition 2.9}\} \\
 & (e, \oplus) \circ (\mathsf{nil}, \rightarrowtail \circ I \| f) = (e, \oplus \circ I \| f) \\
\Leftarrow & \{\text{promotion}\} \\
 & (e, \oplus) \circ \rightarrowtail \circ I \| f = \oplus \circ I \| f \circ (e, \oplus) \| I \\
\equiv & \{(5);\ (1)\} \\
 & \oplus \circ (e, \oplus) \| I \circ I \| f = \oplus \circ (e, \oplus) \| f \\
\equiv & \{(1)\} \\
 & \mathsf{true}. \quad \square
\end{array}
$$

Property 2.11. $I* = I$.

Proof.

$$
\begin{array}{ll}
 & I* \\
= & \{\text{Definition 2.9}\} \\
 & (\mathsf{nil}, \rightarrowtail \circ I \| I) \\
= & \{\| \text{ respects identity; Property 2.4}\} \\
 & I. \quad \square
\end{array}
$$

Property 2.12. $f* \circ g* = (f \circ g)*$.

Proof.

$$
\begin{array}{ll}
 & f* \circ g* \\
= & \{\text{Definition 2.9}\} \\
 & (\mathsf{nil}, \rightarrowtail \circ I \| f) \circ g* \\
= & \{\text{Property 2.10}\} \\
 & (\mathsf{nil}, \rightarrowtail \circ I \| f \circ I \| g) \\
= & \{(1);\ \text{Definition 2.9}\} \\
 & (f \circ g)*. \quad \square
\end{array}
$$

We conclude that $*$ is a type-functor.

The point of this last exercise is that amongst the ways we have of defining new type-functors is our mechanism for defining new (parameterised) types. All of the types which we discuss below are parameterised, and we shall see that each has a "map" operation which induces a type-functor (in another paper [15] we define

map operators for an arbitrary parameterised type and show that these always respect identity and composition). It is then the case that the constructors of a parameterised type are natural transformations: for example, from Definition 2.9 and (5) we have that

$$\rightarrowtail : * \Vert 1 \dot{\rightarrow} *.$$

We use the fact that $*$ is a type-functor in the following subsection, where we give another example of the promotion theorem, this time for the type of "rose trees", a type invented by Lambert Meertens in [18].

2.2. Promotion and natural transformations

Rose trees are a type of tree with labels at the leaves and an arbitrary branching factor: a tree is either a leaf or a node consisting of a list of subtrees. We introduce the type to give a further example of the promotion theorem, and also to give a foretaste of the relationship between promotability and natural transformations: further examples will be given in Sections 4.3 and 4.4 below, but for the present, we shall simply show that rose tree homomorphisms preserve natural transformations.

Definition 2.13 (*Rose trees*). For each type A, define the type of rose trees over A by:

$$A\rho \triangleq \mu(\eta : A \lessdot, \pi : *).$$

The type of rose trees, then, has two constructors: the leaf constructor $\eta \in A \rightarrow A\rho$, and the node constructor $\pi \in A\rho* \rightarrow A\rho$, which governs a list of subtrees. The details of the construction of $A\rho$-algebras should be clear to the reader, and we shall not spell them out. The initiality of the algebra $(A\rho, \eta, \pi)$ prescribes the construction of rose tree homomorphisms: for every $f \in A \rightarrow P$ and $g \in P* \rightarrow P$, there is a unique homomorphism $(\!|f, g|\!)_\rho \in A\rho \rightarrow P$, which satisfies

$$(\!|f, g|\!)_\rho \circ \eta = f, \tag{7}$$

$$(\!|f, g|\!)_\rho \circ \pi = g \circ (\!|f, g|\!)_\rho*. \tag{8}$$

These equations determine the evaluation of the homomorphisms. For the remainder of this section, all homomorphisms are rose tree homomorphisms, and we shall drop the subscript ρ. We introduce the following notation for rose tree promotability.

Notation 2.14 (*Promotability*). $(h : g \overset{\rho}{\mapsto} i) \triangleq (h \circ g = i \circ h*)$ (pronounce as "h is g-to-i-promotable").

It is always the case that homomorphisms enjoy a promotability property; for example, from (8) we have that

$$(\!|f, g|\!) : \pi \overset{\rho}{\mapsto} g, \tag{9}$$

for all rose tree homomorphisms (f, g). Instantiating the promotion theorem for the type of rose trees gives:

Theorem 2.15 (ρ-promotion). $(h : g \overset{\rho}{\mapsto} i) \Rightarrow (h \circ (f, g) = (h \circ f, i))$.

Each data type has its own definition of promotability, and each type of promotability has its own relation to natural transformations (see, for example, Section 4.3 below, in connection with infinite lists); for the type of rose trees we have the following:

Property 2.16. $(\theta : \mathsf{G} \dot{\to} \mathsf{G}*) \Rightarrow \forall(h :: h\mathsf{G} : \theta \overset{\rho}{\mapsto} \theta)$.

The importance of this property is that, as shown by Wadler [23] and de Bruin [7], many polymorphic functions are natural transformations, as a consequence of Reynold's abstraction theorem [21]. Many other properties, such as the promotion theorem and Property 2.19 (see, e.g., [14]) can also be inferred by the abstraction theorem from the higher-order polymorphic type of the operators involved, but here we prefer to build up a body of program transformations by using equational reasoning where possible. However, the relationship pointed out by Wadler and by de Bruin between polymorphism, natural transformations and equalities such as the promotion theorem suggests that it is worthwhile to study in some depth the properties of natural transformations. In Section 4.4, for example, we shall see that natural transformations play a role in developing transformations of initial-segment programs on infinite lists.

In order to show that rose tree homomorphisms preserve natural transformations, we first show that ρ is a type-functor. Defining the action of ρ on functions gives us the map function.

Definition 2.17 (*Map*). For $h \in A \to B$, define $h\rho \triangleq (\eta \circ h, \pi) \in A\rho \to B\rho$.

That ρ preserves identity follows from the above definition and the fact that (η, π) is the identity homomorphism (cf. Property 2.4); that it preserves composition follows straightforwardly from the following property, in the same way as for snoc lists (see Properties 2.10 and 2.12 above).

Property 2.18. $(f, g) \circ h\rho = (f \circ h, g)$.

Proof.

$$
\begin{aligned}
& (f, g) \circ h\rho \\
= \quad & \{\text{Definition 2.17}\} \\
& (f, g) \circ (\eta \circ h, \pi) \\
= \quad & \{(9);\ \text{promotion}\} \\
& ((f, g) \circ \eta \circ h, g) \\
= \quad & \{(7)\} \\
& (f \circ h, g). \qquad \square
\end{aligned}
$$

And now we show that homomorphisms preserve natural transformations. The similarity between the following property and the type rules for contructing homomorphisms is an instance of the principles underlying de Bruin's "naturalness of polymorphism".

Property 2.19. *If* $\varepsilon : F \dot{\to} G$ *and* $\theta : G* \dot{\to} G$, *then* $(\varepsilon, \theta) : F\rho \dot{\to} G$.

Proof.

$$
\begin{aligned}
& (\varepsilon, \theta) \circ fF\rho \\
= \quad & \{\text{Property 2.18}\} \\
& (\varepsilon \circ fF, \theta) \\
= \quad & \{\varepsilon : F \dot{\to} G\} \\
& (fG \circ \varepsilon, \theta) \\
= \quad & \{\theta : G* \dot{\to} G;\ \text{Property 2.16; promotion}\} \\
& fG \circ (\varepsilon, \theta). \qquad \square
\end{aligned}
$$

3. Mutually inductive types: Trees and forests

In this section we consider homomorphisms on two types defined by mutual induction: trees and forests. Forests are list-like structures of trees; trees consist of an internal label and a forest of subtrees, so the trees have an arbitrary branching factor. For a base type A, we denote the type of trees over A by AT, and the type of forests over A by AF. Trees are constructed by the node operator $\diamond \in A \parallel AF \to AT$; forests have as constructors: the empty forest $\square \in AF$ and the concatenation operator $+\!\!< \in AT \parallel AF \to AF$.

In order to define homomorphisms on types defined by mutual induction, we have to extend the construction of the previous section; rather than treat the general case, we consider only the particular example of trees and forests. We begin by giving the functors and algebra which define the types.

We define the pair of types, trees and forests, simultaneously:

$$\langle AT, AF \rangle \triangleq \mu(\diamond : G; \square : F_1, +\!\!< : F_2),$$

where the functors G and F_i are defined as follows: each functor takes a pair of arguments; the functor G pertains to trees, the functors F_i to forests:

$$
\begin{aligned}
& \langle X, Y \rangle G = A \parallel Y, \qquad && \langle f, g \rangle G = I \parallel g, \\
& \langle X, Y \rangle F_1 = \mathbb{1}, && \langle f, g \rangle F_1 = I, \\
& \langle X, Y \rangle F_2 = X \parallel Y, && \langle f, g \rangle F_2 = f \parallel g.
\end{aligned}
$$

The types of the constructors can be simplified to:

$$\diamond \in A \| AF \to AT,$$

$$\square \in \mathbf{1} \to AF,$$

$$\leftarrowtail \in AT \| AF \to AF.$$

The algebra induced by this structure is given by a simple extension to the construction of Section 2.1:

Definition 3.1 (T-F-algebras). A T-F-*algebra* is a quintuple $(P, Q, \oplus, e, \otimes)$ where P and Q are types, $\oplus \in \langle P, Q \rangle \mathsf{G} \to P$ (i.e., $A \| Q \to P$), $e \in \langle P, Q \rangle \mathsf{F}_1 \to Q$ (i.e., $\mathbf{1} \to Q$), and $\otimes \in \langle P, Q \rangle \mathsf{F}_2 \to Q$ (i.e., $P \| Q \to Q$).

A T-F-*homomorphism* from algebra $(P, Q, \oplus, e, \otimes)$ to algebra $(P', Q', \oplus', e', \otimes')$ is a *pair* of functions $\langle f, g \rangle$ where $f \in P \to P'$ and $g \in Q \to Q'$ such that:

$$f \circ \oplus = \oplus' \circ \langle f, g \rangle \mathsf{G} \quad \{= \oplus' \circ I \| g\},$$

$$g \circ e = e' \circ \langle f, g \rangle \mathsf{F}_1 \quad \{g.e = e'\},$$

$$g \circ \otimes = \otimes' \circ \langle f, g \rangle \mathsf{F}_2 \quad \{= \otimes' \circ f \| g\}.$$

As for the data types in the previous sections, we define $(AT, AF, \diamond, \square, \leftarrowtail)$ to be the initial T-F-algebra. This means that for every T-F-algebra $(P, Q, \oplus, e, \otimes)$ there is a unique homomorphism (which we write as $(\oplus; e, \otimes)$ rather than as a pair of functions) which satisfies:

$$(\oplus; e, \otimes) \circ \diamond = \oplus \circ I \| (\oplus; e, \otimes),$$

$$(\oplus; e, \otimes).\square = e,$$

$$(\oplus; e, \otimes) \circ \leftarrowtail = \otimes \circ (\oplus; e, \otimes) \| (\oplus; e, \otimes).$$

Note that since we write these homomorphisms as one function rather than as a pair of functions, we must consider $(\oplus; e, \otimes)$ as having two types: $AT \to P$ and $AF \to Q$.

The promotion theorem for trees and forests is again a consequence of the initiality of trees and forests. It can be expressed as: if $\langle f, g \rangle$ is a T-F-homomorphism from $(P, Q, \oplus, e, \otimes)$ to $(P', Q', \oplus', e', \otimes')$, then:

$$f \circ (\oplus; e, \otimes) = (\oplus'; e', \otimes') \in AT \to P'$$

and

$$g \circ (\oplus; e, \otimes) = (\oplus'; e', \otimes') \in AF \to Q'.$$

Or, equivalently,

Theorem 3.2 (Promotion). *For all* $f \in P \to P'$, $g \in Q \to Q'$, $\oplus \in A \| Q \to P$, $\otimes' \in P' \| Q' \to P'$, $e \in Q$, $\otimes \in P \| Q \to Q$ *and* $\otimes' \in P' \| Q' \to Q'$: *if*

$$f \circ \oplus = \oplus' \circ I \| g \quad \textit{and} \quad g \circ \otimes = \otimes' \circ f \| g,$$

then

$$f \circ (\oplus; e, \otimes) = (\oplus'; g.e, \otimes') \in AT \to P'$$

and

$$g \circ (\oplus; e, \otimes) = (\oplus'; g.e, \otimes') \in AF \to Q'.$$

Finally, as was the case with the data types of the previous sections, the identity homomorphism is obtained from the constructors of the types: $(\diamond; \square, \dashv\prec)$ (cf. Property 2.4).

4. Dualising to infinite data structures

In this section we explore the consequences of applying the principle of duality to initial data types. Duality is often explained as the process of "turning all the arrows around": every function $f \in A \to B$ becomes $f \in B \to A$, and accordingly every composition $f \circ g$ becomes $g \circ f$. What is remarkable about duality is that the dual of every true proposition is also true. A full account of duality can be found in category theory textbooks such as Mac Lane [13]; what we are concerned with here is the dualisation of the results obtained above for initial data types.

Consider the initial data types of Section 2.1: they were effectively defined by a number of constructors $\tau_i \in \mathcal{T}\mathsf{F}_i \to \mathcal{T}$. If we turn these arrows around, then we get what we might call *de*structors $\tau_i \in \mathcal{T} \to \mathcal{T}\mathsf{F}_i$. Initiality of the algebra $(\mathcal{T}, \tau_1, \ldots, \tau_n)$ meant that for every type P and functions $f_i \in P\mathsf{F}_1 \to P$ there was a unique homomorphism $(f_1, \ldots, f_n) \in \mathcal{T} \to P$ such that:

$$(f_1, \ldots, f_n) \circ \tau_i = f_i \circ (f_1, \ldots, f_n)\mathsf{F}_i.$$

Turning these arrows around, we obtain the notion of a "terminal" algebra: for every type P and functions $f_i \in P \to P\mathsf{F}_i$ there is a unique homomorphism $(f_1, \ldots, f_n) \in P \to \mathcal{T}$ such that

$$\tau_i \circ (f_1, \ldots, f_n) = (f_1, \ldots, f_n)\mathsf{F}_i \circ f_i.$$

To summarise, in the case of the initial data types, we had constructors whose range was the type being defined, and recursive homomorphisms whose domain was the type being defined; in the case of terminal data types, we have destructors whose domain is the type being defined, and recursive homomorphisms whose range is the type being defined. We shall presently see that terminal data types allow the definition of data types with infinite objects, such as infinite lists. We begin, however, with an example of dualisation: disjoint sum and its dual, cartesian product.

4.1. Duality

Definition 4.1 (*Disjoint sum*). If A and B are types, their disjoint sum $A+B$ is defined by:

$$A+B \triangleq \mu(\tau_1 : A\prec, \tau_2 : B\prec).$$

The type has therefore two constructors (the standard injections into the left and right summands):

$$\tau_1 \in A \to A+B,$$

$$\tau_2 \in B \to A+B.$$

Following the construction of Section 2.1, $A+B$-algebras are triples (P, f, g) where $f \in A \to P$ and $g \in B \to P$. Moreover, $(A+B, \tau_1, \tau_2)$ is the initial $A+B$-algebra, so that for every $f \in A \to P$ and $g \in B \to P$, there is a unique homomorphism $(f, g)_+ \in A+B \to P$ which satisfies:

$$(f, g)_+ \circ \tau_1 = f, \tag{10}$$

$$(f, g)_+ \circ \tau_2 = g. \tag{11}$$

The uniqueness of these homomorphisms is expressed by: for all h,

$$(h = (f, g)_+) \equiv ((h \circ \tau_1 = f) \wedge (h \circ \tau_2 = g)). \tag{12}$$

Instantiation and simplification of Theorem 2.6 yields the following version of the promotion theorem.

Theorem 4.2 (Promotion). $i \circ (f, g)_+ = (i \circ f, i \circ g)_+$.

Proof.

$$\begin{aligned} & i \circ (f, g)_+ = (i \circ f, i \circ g)_+ \\ \equiv & \quad \{(12) \text{ with } h := i \circ (f, g)_+\} \\ & (i \circ (f, g)_+ \circ \tau_1 = i \circ f) \wedge (i \circ (f, g)_+ \circ \tau_2 = i \circ g) \\ \equiv & \quad \{(10); (11)\} \\ & \text{true.} \quad \square \end{aligned}$$

We shall now construct the dual of disjoint sum. That is, we shall construct the terminal data type from the same functors. We use the symbol ν to indicate the dual construction:

Definition 4.3 (*Cartesian product*). If A and B are types, their cartesian product $A \parallel B$ is defined by:

$$A \parallel B \triangleq \nu(\text{fst}: A\!\ll, \text{snd}: B\!\ll).$$

Instead of constructors, the type has two *de*structors (the projections):

$$\text{fst} \in A \parallel B \to A,$$

$$\text{snd} \in A \parallel B \to B.$$

Since all we are doing is turning the arrows around, it should be no surprise that $A \parallel B$-algebras are triples (P, f, g) where $f \in P \to A$ and $g \in P \to B$, i.e., $f \in P \to P(A\!\ll)$ and $g \in P \to P(B\!\ll)$. Above, we defined $(A+B, \tau_1, \tau_2)$ to be the *initial* $A+B$-algebra; this is dualised by defining $(A \parallel B, \text{fst}, \text{snd})$ to be the *terminal* $A \parallel B$-algebra. That is, there is exactly one homomorphism *from* $(A \parallel B, \text{fst}, \text{snd})$ *to* any other $A \parallel B$-algebra; in other words, for every $f \in P \to A$ and $g \in P \to B$ there is a unique homomorphism $(f, g)_\parallel \in P \to A \parallel B$ which satisfies

$$\text{fst} \circ (f, g)_\parallel = f, \tag{13}$$

$$\text{snd} \circ (f, g)_\parallel = g. \tag{14}$$

The uniqueness of these homomorphisms is expressed by: for all h,

$$(h = (\!|f, g|\!)_{\|}) \equiv ((\mathsf{fst} \circ h = f) \wedge (\mathsf{snd} \circ h = g)). \tag{15}$$

(Homomorphisms $(\!|f, g|\!)_{\|}$ on cartesian products are therefore the functions $\langle f, g \rangle$ of Section 1.)

The promotion theorem for cartesian product is the dual of the promotion theorem for disjoint sum.

Theorem 4.4 (Promotion). $(\!|f, g|\!)_{\|} \circ i = (\!|f \circ i, g \circ i|\!)_{\|}$.

Proof.

$$\begin{aligned}
& (\!|f, g|\!)_{\|} \circ i = (\!|f \circ i, g \circ i|\!)_{\|} \\
\equiv\ & \{(15) \text{ with } h := (\!|f, g|\!)_{\|} \circ i\} \\
& (\mathsf{fst} \circ (\!|f, g|\!)_{\|} \circ i = f \circ i) \wedge (\mathsf{snd} \circ (\!|f, g|\!)_{\|} \circ i = g \circ i) \\
\equiv\ & \{(13); (14)\} \\
& \mathsf{true}. \qquad \square
\end{aligned}$$

Actually, we did not need to supply a proof: all we needed do was state that it was the dual of disjoint sum promotion, and appeal to the principle of duality. In the following subsection, where we state the promotion theorem for an arbitrary terminal data type, we omit the proof for just this reason.

4.2. Promotability for infinite structures

Definition 4.5 (*Infinite data types*). Let $\mathsf{F}_1, \ldots, \mathsf{F}_n$ be type-functors, each of one argument. The terminal type $Y = \nu(v_1 : \mathsf{F}_1, \ldots, v_n : \mathsf{F}_n)$ has destructors $v_i \in Y \to Y\mathsf{F}_i$. The type is defined up to isomorphism by the property that for each type P and functions $f_i \in P \to P\mathsf{F}_i$, there is a unique homomorphism $(\!|f_1, \ldots, f_n|\!)^Y \in P \to Y$ such that:

$$v_i \circ (\!|f_1, \ldots, f_n|\!)^Y = (\!|f_1, \ldots, f_n|\!)^Y \mathsf{F}_i \circ f_i. \tag{16}$$

These equations effectively determine how such a homomorphism is evaluated. We decorate these homomorphisms for infinite types with a superscripted Y to indicate which data structure they pertain to; when no ambiguity as to type seems likely, we drop the superscripts. Expressing the uniqueness property equationally gives: for all h,

$$(h = (\!|f_1, \ldots, f_n|\!)) \equiv \forall(i :: v_i \circ h = h\mathsf{F}_i \circ f_i). \tag{17}$$

We give the type of infinite lists as an example.

Example 4.6 (*Infinite lists*). For each type A, let $A\ll$ denote the constant type-functor always returning A or, given a function, the identity function on A, and let I denote the identity functor. We define the type of infinite lists over A, denoted by $A\infty$, by:

$$A\infty \triangleq \nu(\mathsf{hd} : A\ll, \mathsf{tl} : \mathsf{I}).$$

Thus the type has two destructors $\mathsf{hd} \in A^\infty \to A$ and $\mathsf{tl} \in A^\infty \to A^\infty$. Given an infinite list, one can imagine hd as returning the head and tl the tail of that list.

According to Definition 4.5, an infinite list is constructed by means of functions $f \in P \to A$ and $g \in P \to P$, which induce a homomorphism $(f, g)^\infty \in P \to A^\infty$ (we shall immediately drop the superscripted ∞). Corresponding to equations (16), this homomorphism is the unique function satisfying:

$$\mathsf{hd} \circ (f, g) = f, \tag{18}$$

$$\mathsf{tl} \circ (f, g) = (f, g) \circ g. \tag{19}$$

That is, given an object $p \in P$, we can construct the infinite list $(f, g).p$ whose head is given by $f.p$ and whose tail is given by $(f, g).(g.p)$; in general, the $(n+1)$th element of the list is $f.(g^n.p)$. The reader may visualise the infinite list $(f, g).p$ as the sequence

$$f.p,\ f.(g.p),\ f.(g^2.p),\ f.(g^3.p), \ldots$$

but we shall always treat homomorphisms as "lazy" functions, so that expressions of the form $(f, g).p$ are considered to be in canonical form, and not to be further evaluated: the only computation rules for homomorphisms on infinite lists are those given by equations (18) and (19) above.

An elementary example of a homomorphism on infinite lists is the identity homomorphism, (hd, tl). Given an infinite list l, the $(n+1)$th element of the list (hd, tl).l is computed by hd.(tl^n.l), which strongly suggests that (hd, tl).$l = l$. Formally, we prove this identity for the general case as a corollary to the uniqueness property (17) of homomorphisms. The following is the dual of Property 2.4; the reader may compare the two proofs, which are each other's duals.

Corollary 4.7 (Identity). $I = (v_1, \ldots, v_n) \in Y \to Y$.

Proof.

$$\begin{aligned} & I = (v_1, \ldots, v_n) \\ \equiv\ & \{(17)\} \\ & \forall(i :: v_i \circ I = I\mathsf{F}_i \circ v_i) \\ \equiv\ & \{\text{functors preserve identity}\} \\ & \text{true.} \quad \square \end{aligned}$$

We now present the promotion theorem for terminal data structures, which is dual to the promotion theorem for initial data structures. Promotion theorems for the particular data types which we consider below are instantiations of this theorem.

Theorem 4.8 (Promotion). *For a type* $Y = \nu(v_1 : \mathsf{F}_1, \ldots, v_n : \mathsf{F}_n)$,

$$\forall(i :: f_i \circ h = h\mathsf{F}_i \circ g_i) \Rightarrow ((f_1, \ldots, f_n)^Y \circ h = (g_1, \ldots, g_n)^Y).$$

Proof. Dual of proof of Theorem 2.6. □

Since we shall be using infinite list promotion extensively, we introduce the following notation for its antecedent:

Notation 4.9 (*Promotability*). If $f \circ h = h \circ g$, we write this as $h : f \overset{\infty}{\mapsto} g$ (pronounced "h is f-to-g-promotable").

The instantiation of the promotion theorem for infinite lists is:

Theorem 4.10 (∞-promotion). *For* $f \in Q \to A$, $f' \in P \to A$, $g \in Q \to Q$, $g' \in P \to P$ *and* $h \in P \to Q$,

$$((f \circ h = f') \wedge (g \circ h = h \circ g')) \Rightarrow ((f, g) \circ h = (f', g')).$$

Or, equivalently,

$$(h : g \overset{\infty}{\mapsto} g') \Rightarrow ((f, g) \circ h = (f \circ h, g')).$$

It is this latter form that we use in the sequel.

4.3. Promotability and natural transformations

For each type that we introduce there seem to be certain recurring points of interest: homomorphisms enjoy a promotability property, as do natural transformations; there is often a map operation which induces a type-functor, and homomorphisms preserve natural transformations. These points were illustrated in Section 2.2 above for rose trees, and are generally useful in building up a small theory for any data type that one introduces. We go through the steps for the type of infinite lists. First of all, we note that from (19) we have:

$$(f, g) : \mathsf{tl} \overset{\infty}{\mapsto} g \tag{20}$$

for every homomorphism (f, g). Moreover, from the definition of natural transformations (Definition 1.2) we obtain two promotability properties:

$$(\eta : \mathsf{F} \dot{\to} \mathsf{G}) \Rightarrow \forall (h :: \eta : h\mathsf{F} \overset{\infty}{\mapsto} h\mathsf{G}), \tag{21}$$

$$(\eta : \mathsf{F} \dot{\to} \mathsf{F}) \Rightarrow \forall (h :: h\mathsf{F} : \eta \overset{\infty}{\mapsto} \eta). \tag{22}$$

Given a function $h \in A \to B$, we can construct the map $h\infty \in A\infty \to B\infty$, which applies h componentwise to every element of a given infinite list:

Definition 4.11 (*Map*). For $h \in A \to B$, define $h\infty \triangleq (h \circ \mathsf{hd}, \mathsf{tl}) \in A\infty \to B\infty$.

A general property of maps is that they can always be "brought inside" a homomorphism, cf. Properties 2.10 and 2.18:

Property 4.12. $h\infty \circ (f, g) = (h \circ f, g)$.

Proof.

$$
\begin{aligned}
& h\infty \circ (f, g) \\
= \; & \{\text{Definition 4.11}\} \\
& (h \circ \mathsf{hd}, \mathsf{tl}) \circ (f, g) \\
= \; & \{(20), \text{Theorem 4.10}\} \\
& (h \circ \mathsf{hd} \circ (f, g), g) \\
= \; & \{(18)\} \\
& (h \circ f, g). \qquad \square
\end{aligned}
$$

In order to show that ∞ is a type-functor, we must show that it respects identity and composition. The proofs of these properties are straightforward and omitted.

Property 4.13. $I\infty = I \in A\infty \to A\infty$.

Property 4.14. $f\infty \circ g\infty = (f \circ g)\infty$.

Another general property is that the constructors or destructors of a data type are natural:

Property 4.15. $\mathsf{hd} : \infty \dot{\to} \mathsf{I}$ *and* $\mathsf{tl} : \infty \dot{\to} \infty$.

Proof. Immediate from Definition 4.11, (18) and (19). $\square$

And finally, infinite list homomorphisms preserve natural transformations:

Property 4.16. *If* $\eta : \mathsf{F} \dot{\to} \mathsf{G}$ *and* $\theta : \mathsf{F} \dot{\to} \mathsf{F}$, *then* $(\eta, \theta) : \mathsf{F} \dot{\to} \mathsf{G}\infty$.

4.4. Initial segments

In this section, we derive some transformations for functions on the initial segments of infinite lists. The goal is to derive "online" algorithms which produce output in constant time as each new input is read from a given infinite list.

Definition 4.17 (*Accumulation*). Let $\oplus \in B \parallel A \to B$. We define the *accumulation* of $\oplus$, denoted by $/\oplus/$, to be the function:

$$(\mathsf{fst}, \vec{\oplus}) \in B \parallel A\infty \to B\infty,$$

where

$$\vec{\oplus} \triangleq \langle \oplus \circ I \parallel \mathsf{hd}, \mathsf{tl} \circ \mathsf{snd} \rangle.$$

So, for all $\langle x, y \rangle \in B \parallel A\infty$,

$$\vec{\oplus}.\langle x, y \rangle = \langle x \oplus \mathsf{hd}.y, \mathsf{tl}.y \rangle.$$

The accumulation $/\oplus/$ takes $b \in B$ and infinite list $a_1, a_2, a_3, \ldots$ into the infinite list b, $b \oplus a_1$, $(b \oplus a_1) \oplus a_2$, $((b \oplus a_1) \oplus a_2) \oplus a_3, \ldots$. (This is, in fact, the "scan" function of Bird and Wadler [6].) We write $b/\oplus/$ for the function that takes $s \in A\infty$ to $/\oplus/.\langle b, s\rangle$. For example, $\mathsf{nil}/\rightarrowtail/$ gives the infinite list of initial segments of an infinite list.

Since $\vec{\oplus}$ is defined in terms of polymorphic functions, it preserves natural transformations:

Property 4.18. *If* $\oplus : F \parallel G \dot{\to} F$, *then* $\vec{\oplus} : F \parallel G\infty \dot{\to} F \parallel G\infty$.

Proof.

$$
\begin{aligned}
& \vec{\oplus} \circ f F \parallel f G\infty \\
= \; & \{\triangle\vec{\oplus}, \text{composition}\} \\
& \langle \oplus \circ I \parallel \mathsf{hd} \circ f F \parallel f G\infty, \mathsf{tl} \circ \mathsf{snd} \circ f F \parallel f G\infty \rangle \\
= \; & \{(1), \mathsf{hd} : \infty \dot{\to} \mathsf{I}, (1)\} \\
& \langle \oplus \circ f F \parallel f G \circ I \parallel \mathsf{hd}, \mathsf{tl} \circ f G\infty \circ \mathsf{snd} \rangle \\
= \; & \{\oplus : F \parallel G \dot{\to} F, \mathsf{tl} : \infty \dot{\to} \infty\} \\
& \langle f F \circ \oplus \circ I \parallel \mathsf{hd}, f G\infty \circ \mathsf{tl} \circ \mathsf{snd} \rangle \\
= \; & \{\text{composition, definition of } \vec{\oplus}\} \\
& f F \parallel f G\infty \circ \vec{\oplus}. \qquad \square
\end{aligned}
$$

The following property of $\vec{\oplus}$ is proven by a straightforward calculation, left to the reader.

Property 4.19. *If* $f \circ \otimes = \oplus \circ f \parallel I$, *then* $f \parallel I : \vec{\oplus} \overset{\infty}{\mapsto} \vec{\otimes}$.

Accumulations preserve natural transformations, in the sense of the following property.

Property 4.20 (Natural transformations). *If*

$$\oplus : F \parallel G \dot{\to} F,$$

then

$$/\oplus/ : F \parallel G\infty \dot{\to} F\infty.$$

Proof. From Property 4.18, we have that $\vec{\oplus} : F \parallel G\infty \dot{\to} F \parallel G\infty$; moreover, $\mathsf{fst} : F \parallel G\infty \dot{\to} F$, so by Definition 4.17 and Property 4.16, $/\oplus/ : F \parallel G\infty \dot{\to} F\infty$ as desired. $\square$

Corollary 4.21. *Since* $\succ\!\!-\!: * \,[\!]\, 1 \stackrel{\cdot}{\to} *$, *we have* $/\succ\!\!-/ : * \,[\!]\, \infty \stackrel{\cdot}{\to} *\infty$ *and also* $\mathsf{nil}/\succ\!\!-/ : \infty \stackrel{\cdot}{\to} *\infty$.

Corollary 4.22. *Let* $(e,\oplus)_* \in A* \to B$ *be a snoc list homomorphism; the following holds*:

$$(e,\oplus)_*\infty \circ /\succ\!\!-/ = /\oplus/ \circ (e,\oplus)_* \,[\!]\, I.$$

Proof. Since by definition, $(e,\oplus)_* \circ \succ\!\!- = \oplus \circ (e,\oplus)_* \,[\!]\, I$, Property 4.19 gives that:

$$(e,\oplus)_* \,[\!]\, I : \vec{\oplus} \overset{x}{\mapsto} \succ\!\!-. \tag{23}$$

Hence:

$$\begin{aligned}
&(e,\oplus)_*\infty \circ /\succ\!\!-/ \\
&= \quad \{\text{Definition 4.17, Property 4.12}\} \\
&\quad ((e,\oplus)_* \circ \mathsf{fst}, \succ\!\!-) \\
&= \quad \{\text{property of fst}\} \\
&\quad (\mathsf{fst} \circ (e,\oplus)_* \,[\!]\, I, \succ\!\!-) \\
&= \quad \{(23), \text{Theorem 4.10}\} \\
&\quad (\mathsf{fst}, \vec{\oplus}) \circ (e,\oplus)_* \,[\!]\, I \\
&= \quad \{\text{Definition 4.17}\} \\
&\quad /\oplus/ \circ (e,\oplus)_* \,[\!]\, I. \qquad \square
\end{aligned}$$

Corollary 4.23. $(e,\oplus)_*\infty \circ \mathsf{nil}/\succ\!\!-/ = e/\oplus/$.

An important consequence of the above is that we can now use snoc list promotion in transforming programs involving accumulations. For example:

Corollary 4.24. *If* $f \circ \oplus = \otimes \circ f \,[\!]\, g$, *then* $f\infty \circ e/\oplus/ = f.e/\otimes/ \circ g\infty$.

Proof. From the above distributivity property of f, it is straightforward to show (using promotion) that

$$f \circ (e,\oplus)_* = (f.e,\otimes)_* \circ g* \tag{24}$$

and so:

$$\begin{aligned}
&f\infty \circ e/\oplus/ \\
&= \quad \{\text{Corollary 4.23, } \infty \text{ is a functor}\} \\
&\quad (f \circ (e,\oplus)_*)\infty \circ \mathsf{nil}/\succ\!\!-/ \\
&= \quad \{(24), \infty \text{ is a functor}\} \\
&\quad (f.e,\otimes)_*\infty \circ g*\infty \circ \mathsf{nil}/\succ\!\!-/ \\
&= \quad \{\text{Corollary 4.21}\} \\
&\quad (f.e,\otimes)_*\infty \circ \mathsf{nil}/\succ\!\!-/ \circ g\infty \\
&= \quad \{\text{Corollary 4.23}\} \\
&\quad f.e/\otimes/ \circ g\infty. \qquad \square
\end{aligned}$$

Note the role played in the last three corollaries by the natural transformation nil/$\rightarrowtail$/ in moving from an inefficient left-hand side to a more efficient right-hand side. Although none of the corollaries necessarily represents a significant increase in efficiency, the right-hand side in each case represents an "online" algorithm: a result is output in constant time when a new input is read.

4.5. Infinite multiway trees

In this section we give a further example of an infinite data structure: multiway trees. A multiway tree consists of an internal label and a list of subtrees (thus the tree has an arbitrary branching factor).

Definition 4.25 (*Multiway trees*). We define the type of infinite multiway trees over A by $A\varpi \triangleq \nu(\mathrm{rt}: A\!\!\prec, \mathrm{sb}: *)$. Thus $A\varpi$ has destructors $\mathrm{rt} \in A\varpi \to A$ and $\mathrm{sb} \in A\varpi \to A\varpi*$ which return the root label and the subtrees respectively of a given tree, and for every $f \in P \to A$ and $g \in P \to P*$, there is a unique function $(f, g)^{\varpi} \in P \to A\varpi$ such that:

$$\mathrm{rt} \circ (f, g)^{\varpi} = f, \tag{25}$$

$$\mathrm{sb} \circ (f, g)^{\varpi} = (f, g)^{\varpi} * \circ g. \tag{26}$$

That is, the label at the root of the tree is generated by f, and the subtrees are generated by recursively mapping the homomorphism to the list given by g. (Note that such a tree may have finite depth if all branches eventually lead to empty lists of subtrees.)

Example 4.26 (*Game trees*). Let A be a type which represents valid states of some game and let $moves \in A \to A*$ be a function which returns the list of valid states which may be reached in one move from a given state. The homomorphism $(I, moves)^{\varpi} \in A \to A\varpi$ generates the game tree of all possible states which may be reached from a given state.

Henceforth, we shall drop the superscripted ϖ. We introduce the following notation for ϖ-promotability:

Notation 4.27 (ϖ-promotability). $(h : f \overset{\varpi}{\mapsto} g) \triangleq (f \circ h = h* \circ g)$.

It follows immediately from (26) that:

$$(f, g) : \mathrm{sb} \overset{\varpi}{\mapsto} g \tag{27}$$

for all homomorphisms (f, g). The notions of ∞- and ϖ-promotability are related by the following property:

Property 4.28. $((h : g \overset{\varpi}{\mapsto} f) \wedge (h* : j \overset{\infty}{\mapsto} i)) \Rightarrow (h : j \circ g \overset{\varpi}{\mapsto} i \circ f)$.

Proof.

$$
\begin{aligned}
& j \circ g \circ h \\
= \; & \{h : g \xrightarrow{\varpi} f\} \\
& j \circ h* \circ f \\
= \; & \{h* : j \xrightarrow{\infty} i\} \\
& h* \circ i \circ f. \qquad \square
\end{aligned}
$$

The promotion theorem for multiway trees is obtained by instantiating Theorem 4.8 and reformulating slightly, which gives:

$$(h : g \xrightarrow{\varpi} g') \Rightarrow ((f, g) \circ h = (f \circ h, g')). \tag{28}$$

Again, the effect of ϖ-promotion is to bring f inside the homomorphism brackets.

Definition 4.29 (*Map*). For $h \in A \to B$, define $h\varpi \triangleq (h \circ \mathrm{rt}, \mathrm{sb}) \in A\varpi \to B\varpi$.

As for infinite lists, it is possible to prove straightforwardly basic properties of ϖ, such as that ϖ is a type-functor, and the following property, cf. Property 4.12.

Property 4.30. $h\varpi \circ (f, g) = (h \circ f, g)$.

We shall give only one example of the use of ϖ-promotion, concerning a "pruning" operation on trees: before defining that operation, we introduce the notion of filters on finite lists.

If $p \in A \to \mathrm{Bool}$ is a predicate on A then the filter of p is a function $p\lhd \in A* \to A*$ which removes all elements from a given list which do not satisfy p. The function is a snoc list homomorphism:

Definition 4.31 (*Filter*). $p\lhd \triangleq (\mathrm{nil}, \oplus)_*$, where for all $a \in A$ and $x \in A*$,

$$x \oplus a = \text{if } p.a. \text{ then } x \succ a \text{ else } x \text{ fi}.$$

We shall need the following property of $p\lhd$, called the "range translation" by Backhouse [1]: for all f,

$$p\lhd \circ h* = h* \circ (p \circ h)\lhd \tag{29}$$

that is,

$$h* : p\lhd \xrightarrow{\infty} (p \circ h)\lhd.$$

Combining this with (27), Property 4.28 gives:

$$(f, g) : p\lhd \circ \mathrm{sb} \xrightarrow{\varpi} (p \circ (f, g))\lhd \circ g \tag{30}$$

for all ϖ-homomorphisms (f, g).

Now, our pruning operation on infinite trees, which we shall also denote by $p\triangleleft$, is the homomorphism

$$p\triangleleft \triangleq (\mathsf{rt}, (p \circ \mathsf{rt})\triangleleft \circ \mathsf{sb}) \in A\varpi \to A\varpi,$$

which recursively removes all subtrees whose root labels do not satisfy the predicate p. The operation enjoys the following property:

Property 4.32. *$p\triangleleft \circ (f, g) = (f, (p \circ f)\triangleleft \circ g)$ for all f and g.*

Proof.

$$\begin{aligned}
&(\mathsf{rt}, (p \circ \mathsf{rt})\triangleleft \circ \mathsf{sb}) \circ (f, g)\\
=\ &\{(30) \text{ with } p := p \circ \mathsf{rt}, (28)\}\\
&(\mathsf{rt} \circ (f, g), (p \circ \mathsf{rt} \circ (f, g))\triangleleft \circ g)\\
=\ &\{(25), \text{twice}\}\\
&(f, (p \circ f)\triangleleft \circ g). \qquad \square
\end{aligned}$$

Corollary 4.33 (Range translation). *Range translation* (29) *also holds for ϖ-maps; i.e.,*

$$p\triangleleft \circ h\varpi = h\varpi \circ (p \circ h)\triangleleft.$$

Proof.

$$\begin{aligned}
&p\triangleleft \circ h\varpi\\
=\ &\{\text{Definition 4.29}\}\\
&p\triangleleft \circ (h \circ \mathsf{rt}, \mathsf{sb})\\
=\ &\{\text{Property 4.32}\}\\
&(h \circ \mathsf{rt}, (p \circ h \circ \mathsf{rt})\triangleleft \circ \mathsf{sb})\\
=\ &\{\text{Property 4.30; } \triangleq\triangleleft\}\\
&h\varpi \circ (p \circ h)\triangleleft. \qquad \square
\end{aligned}$$

5. Conclusion

As we said in the introduction, our starting point in this paper has been the work of Bird and Meertens. Our contribution, as we see it, is the generalisation of their work to other data structures in a uniform way. We proposed a uniform notion of homomorphism over arbitrary data structures, and derived promotion theorems for each type of homomorphism. These promotion theorems are more general than those given by Bird and Meertens, and moreover the promotion theorem for each data structure can be inferred from the definition of the data structure. One of the advantages of this generality is that it is possible to identify and reason about operators shared by large classes of data structures (see, for example, Malcolm [15], where map distributivity is proven for parameterised data structures).

To a great extent, the generality we have achieved is due to the use of category-theoretical notions like that of type-functor and initiality. Similar work to the present paper, but where concepts from category theory are used much more extensively to derive program transformations can be found in Spivey [22] and Malcolm [15]. In these papers, much is made of the notions of polymorphism and natural transformations: the links between the two notions, and several applications to the field of program transformation can be found in Wadler [23] and de Bruin [7]. Another advantage of the use of a categorical setting is the conciseness of many categorical concepts (exemplified above in the use of type functors in defining data structures): for example, as in [22], the uniqueness properties of initial algebras can be summarised by stating an adjointness property, or a distributivity property by stating that a function is a natural transformation; yet another example is that homomorphisms and promotion theorems for infinite data structures can be obtained from their counterparts for finite data structures via the categorical notion of duality, thus allowing finite and infinite structures to be handled in a coherent manner.

An interesting area for future research comes from a paper by Backhouse [2], in which the results of the present paper are generalised to relations: i.e., Backhouse constructs relational homomorphisms where we have been concerned with functional homomorphisms. For example, we have defined map operators which apply a function "componentwise" to a given structure, preserving the shape of the structure; the corresponding relational homomorphism relates "componentwise" two structures of the same shape. Interestingly, the promotion theorem still holds for these relational homomorphisms, and in fact a stronger formulation can be obtained by replacing equalities by containment inequalities. This generalisation carries on work by de Moor [8] in opening the way to reasoning calculationally about inverses and nondeterminism within the Bird-Meertens formalism, and allows more freedom in formulating specifications. An application of Backhouse's relational calculus which we have begun to investigate [16] is the use of "guards" as a means of reasoning about subsets of data types: a guard effectively introduces context into a calculation, functioning somewhat like a comment or assertion in a program text. The technique has been used earlier by Hesselink in the context of process and specification algebras [10, 11], and we believe it could play an important role in the specification of programs on infinite data structures.

References

[1] R.C. Backhouse, An exploration of the Bird-Meertens formalism, Tech. Rept. CS8810, Department of Mathematics and Computing Science, University of Groningen, Groningen, Netherlands (1988).

[2] R.C. Backhouse, Naturality of homomorphisms, *Lecture Notes, International Summer School on Constructive Algorithmics* 3 (1989).

[3] R.S. Bird, The promotion and accumulation strategies in transformational programming, *ACM Trans. Programming Languages Syst.* 6 (1984) 487-504.

[4] R.S. Bird, An introduction to the theory of lists, in: M. Broy, ed., *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series 36 (Springer, Berlin, 1987).

[5] R.S. Bird, Constructive functional programming, International Summer School on Constructive Methods in Computing Science, Marktoberdorf (1988).

[6] R.S. Bird and P. Wadler, *Introduction to Functional Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1988).

[7] P.J. de Bruin, Naturalness of polymorphism, Department of Mathematics and Computing Science, University of Groningen, Groningen, Netherlands (1989).

[8] O. de Moor, Indeterminacy in optimization problems, *Lecture Notes, International Summer School on Constructive Algorithmics* 2 (1989).

[9] T. Hagino, A typed lambda calculus with categorical type constructors, in: D.H. Pitt, A. Poigne, and D.E. Rydeheard, eds., *Category Theory and Computer Science*, Lecture Notes in Computer Science 283 (Springer, Berlin, 1988) 140-157.

[10] W.H. Hesselink, An algebraic calculus of commands, Tech. Rept. CS 8808, Department of Computing Science, Groningen University, Groningen, Netherlands (1988).

[11] W.H. Hesselink, Command algebras, recursion and program transformation, Tech. Rept. CS 8812, Department of Computing Science, Groningen University, Groningen, Netherlands (1988).

[12] J. Jeuring, Deriving algorithms on binary labelled trees, in: *Proceedings Conference on Computing Science in the Netherlands* (1989).

[13] S. Mac Lane, *Categories for the Working Mathematician*, Graduate Texts in Mathematics 5 (Springer, Berlin, 1971).

[14] G. Malcolm, An application of the abstraction theorem, Department of Computing Science, Groningen University, Groningen, Netherlands (1989).

[15] G. Malcolm, Factoring homomorphisms, Computing Science Notes CS 8908, Department of Computing Science, Groningen University, Groningen, Netherlands (1989).

[16] G. Malcolm, Squiggoling in context, Department of Computing Science, Groningen University, Groningen, Netherlands (1989).

[17] L. Meertens, Algorithmics: Towards programming as a mathematical activity, in: *Proceedings CWI Symposium on Mathematics and Computer Science* (North-Holland, Amsterdam, 1986) 289-334.

[18] L. Meertens, First steps towards the theory of rose trees, Draft Rept., CWI, Amsterdam (1988).

[19] N.P. Mendler, Inductive definitions in type theory, Ph.D. Thesis, Cornell University, Ithaca, NY (1987).

[20] J. Meseguer and J.A. Goguen, Initiality, induction and computability, in: M. Nivat and J.C. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, Cambridge, 1985) 459-542.

[21] J.C. Reynolds, Types, abstraction and parametric polymorphism, in: R.E. Mason, ed., *IFIP '83* (Elsevier Science Publishers, Amsterdam, 1983) 513-523.

[22] M. Spivey, A categorical approach to the theory of lists, in: J.L.A. van de Snepscheut, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science 375 (Springer, Berlin, 1989) 399-408.

[23] P. Wadler, Theorems for free! Draft Rept., Department of Computing Science, University of Glasgow, Glasgow, Scotland (1989).