Monadic Maps and Folds for Arbitrary Datatypes

Maarten Fokkinga,[*] University of Twente

Version of June 1, 1994

Each datatype constructor comes equiped not only with a so-called map and fold (*catamorphism*), as is widely known, but, under some condition, also with a kind of map and fold that are related to an arbitrary given monad.

This result follows from the preservation of initiality under lifting from the category of algebras in a given category to a certain other category of algebras in the Kleisli category related to the monad.

1 Introduction

1.1 Goal. A monad is a collection of specific operations that allows for a far-going structuring of programs. This has convincingly been demonstrated by Wadler [18] and others, and we shall not attempt to redo that here. The main contribution of this paper is the construction of some more, monad related, ways of structuring programs over an arbitrary datatype. In particular, a kind of *map* and *catamorphism* for the datatype, that are related to a given monad; a catamorphism is what other people sometimes call a *generalised fold*.

1.2 Formalism. In order to be truly general in our formulations and proofs, we need some suitable concepts and notation: category theory. The functional programmer not familiar with category theory may nevertheless still understand a large part of this paper, since each formula is just a functional program fragment. The only unconventional aspect is that the programs are expressed entirely "at the function level", that is, new functions are formed out of given ones *only by combining* functions (like $f \mathbin{;} g$, $f \times g$, and $f \triangledown g$), not by expressing the function result in terms of the function arguments (like $x \mapsto 2x+3$), although in examples and informal interpretations we may do so. The notation and some terminology is briefly introduced in Section 2; here it is also argued that the theory we set up, is quite general and covers all of the datatypes found in current functional languages. Moreover, the notion of monad is motivated and briefly illustrated. In order to help the

[*] Department INF, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands, e-mail: `fokkinga@cs.utwente.nl`

reader not familiar with category somewhat, we shall speak of 'type' and 'function', rather than of 'object' and 'morphism'. For an intuitive understanding, one may equate 'natural transformation' with 'polymorphic function'.

1.3 The method. In Section 3 we specify what we mean by monadic catamorphism and monadic map, and give the eventual definitions. The definitions can be obtained in a calculation of a few lines long (as shown in Appendix A), but we explore a different path: via an adjunction between the category of algebras and another category of algebras built upon the Kleisli category. The adjunction is constructed in Section 5, and in Section 6 we read off, so to say, the definitions and properties of the monadic map and catamorphism. In the construction we need a class of natural transformations $dist_F\colon FM^* \dot{\to} MF$, for arbitrary regular functor F and given monad M. In Section 4 the class $dist_F$ is constructed by induction on the formation of the *regular* functor F. Actually, the definition of the monadic map is but one step in this inductive construction of $dist$.

2 Preliminaries

In this section we explain some notation and the way datatypes are formalised in a general way, and we briefly motivate and define monads.

Datatypes have been studied in a category theoretical framework by, for example, Hagino [10, 9], Wraith [19, 20], Malcolm [11, 12], Fokkinga [5, 7], and others. We shall use their results, phrased in the calculational style explained by Fokkinga [6]. As a help to the reader, we very briefly explain the theory, first by means of some examples (the datatypes sum and product, also known as disjoint union and cartesian product, and the datatype list), and then more abstractly.

2.1 Sum, product, list. Consider the following declaration of $+$, inl, inr and $\triangledown$:

datatype $a + b$ **with** $\_ \triangledown \_$ **has constructors**
$inl\colon\ a \to a + b$
$inr\colon\ b \to a + b \quad .$

For each pair of types a, b, this declaration denotes an initial algebra with carrier $a + b$ and operations inl, inr typed as above. Operation $\triangledown$ has the following typing:

$$f\colon\ a \to c \ \wedge\ g\colon\ b \to c \quad \Rightarrow \quad f \triangledown g\colon\ a + b \to c \quad . \qquad \text{sum-}\textsc{Type}$$

To be precise, the datatype (initial algebra thus) is *completely* characterised by the statement:

$$h = f \triangledown g \quad \equiv \quad inl \mathbin{;} h = f \ \wedge\ inr \mathbin{;} h = g \quad . \qquad \text{sum-}\textsc{Charn}$$

The implication to the right asserts that there exists a solution for h in the right-hand side equations, while the implication to the left means that the solution is unique. Thus, for given f and g the right-hand side equations may be read as a *definition* of h. Indeed, the equations define h by case analysis on the $\mathit{inl}, \mathit{inr}$-construction of the argument; such an h is denoted $f \triangledown g$ (conventionally $[f,g]$). Interpreted in the category of sets with *total* functions, the set $a+b$ is (isomorphic to) the disjoint union of a and b.

Moreover, the declaration also defines the type former $+$ to be a functor:

$$\begin{array}{lcll} f+g & = & (f \mathbin{;} \mathit{inl}) \triangledown (g \mathbin{;} \mathit{inr}) & \\ & : & a+b \to a'+b' & \quad \text{whenever } f\colon\ a \to a', \quad g\colon\ b \to b' \quad . \end{array}$$

Some important consequences of the above definitions are:

$$f \triangledown g \mathbin{;} h \quad = \quad (f \mathbin{;} h) \triangledown (g \mathbin{;} h) \qquad \text{sum-Fusion}$$

$$f + g \mathbin{;} h \triangledown j \quad = \quad (f \mathbin{;} h) \triangledown (g \mathbin{;} j) \qquad \text{sum-Fusion}$$

$$f \triangledown g = h \triangledown j \quad \equiv \quad f = h \ \wedge\ g = j \quad . \qquad \text{sum-Decomp}$$

Product is the categorical dual of sum. So we can be brief here. The declaration:

> **datatype** $a \times b$ **with** $\_ \vartriangle \_$ **has destructors**
> $\mathit{exl}\colon\ a \times b \to a$
> $\mathit{exr}\colon\ a \times b \to b$

defines $\times, \mathit{exl}, \mathit{exr}, \vartriangle$ in such a way that:

$$f\colon\ c \to a \ \wedge\ g\colon\ c \to a \quad \Rightarrow \quad f \vartriangle g\colon\ c \to a \times b \qquad \text{prod-Type}$$

$$h = f \vartriangle g \quad \equiv \quad h \mathbin{;} \mathit{exl} = f \ \wedge\ h \mathbin{;} \mathit{exr} = g \qquad \text{prod-Charn}$$

$$\begin{array}{lcll} f \times g & = & (\mathit{exl} \mathbin{;} f) \vartriangle (\mathit{exr} \mathbin{;} g) & \qquad \text{prod-Ftr} \\ & : & a \times b \to a' \times b' & \quad \text{whenever } f\colon\ a \to a', \quad g\colon\ b \to b' \quad . \end{array}$$

There are again several corollaries, but we shall not formulate them explicitly here.

Next consider the following declaration of List, $\mathit{nil}, \mathit{cons}$ and foldr:

> **datatype** $\mathit{List}\, a$ **with** $\_\mathrm{foldr}\,\_$ **has constructors**
> $\mathit{nil}\colon\ () \to \mathit{List}\, a$
> $\mathit{cons}\colon\ a \times \mathit{List}\, a \to \mathit{List}\, a \quad .$

For each type a, this declaration denotes an initial algebra with carrier $\mathit{List}\, a$ and operations $\mathit{nil}, \mathit{cons}$ typed as above. Operation $\_\mathrm{foldr}\,\_$ has the following typing:

$$f\colon\ () \to a \ \wedge\ g\colon\ a \times b \to b \quad \Rightarrow \quad f \mathbin{\mathrm{foldr}} g\colon\ \mathit{List}\, a \to b \quad . \qquad \text{list-Type}$$

To be precise, the datatype (initial algebra thus) is *completely* characterised by the statement:

$$h = f \mathbin{\mathrm{foldr}} g \quad \equiv \quad \mathit{nil} \mathbin{;} h = f \ \wedge\ \mathit{cons} \mathbin{;} h = \mathit{id} \times h \mathbin{;} g \quad . \qquad \text{list-Charn}$$

The implication to the right asserts that there exists a solution for h in the right-hand side equations, while the implication to the left means that the solution is unique. Thus, for given f and g the right-hand side equations may be read as a *definition* of h. Indeed, the equations define h by induction on the $nil, cons$-construction of the argument; such an h is known as the 'foldright' of f and g. Interpreted in the category of sets with *total* functions, the set $\mathit{List}\, a$ is (isomorphic to) the set of finite sequences over a.

Moreover, the declaration also defines the type former List to be a functor:

$$\begin{array}{lll} \mathit{List}\, f & = & nil \,\mathrm{foldr}\, (f \times \mathit{List}\, f \,;\, cons) \\ & : & \mathit{List}\, a \to \mathit{List}\, a' \qquad\qquad\qquad \text{whenever } f\colon\ a \to a' \quad . \end{array}$$

The functor behaves as the well-know *map*: $\mathit{List}\, f = map\, f$. The reader may verify this by eliminating foldr by law list-CHARN, and then rewriting the equations at the point level, using arguments and function application explicitly. There are several important consequences of the above definitions, but we don't bother to mention them.

Below we shall explain how both the characterisations and the functor definitions are fully determined by the declaration alone. In order to keep the formulas simple, we consider datatypes with only one constructor. This is without loss of generality, since a collection of functions with the same target type can be combined into a single function, by means of a suitable use of disjoint union. In the case of lists, for example, we have that the three statements:

$$\begin{array}{lll} nil & : & () \to \mathit{List}\, a \\ cons & : & a \times \mathit{List}\, a \to \mathit{List}\, a \\ cstr & = & nil \triangledown cons \end{array}$$

are equivalent to:

$$\begin{array}{lll} nil & = & \mathit{inl} \,;\, cstr \\ cons & = & \mathit{inr} \,;\, cstr \\ cstr & : & () \;+\; a \times \mathit{List}\, a \;\to\; \mathit{List}\, a \quad , \end{array}$$

as the reader may wish to verify on the basis of the laws given so far. Thus we might alternatively declare the datatype of lists by:

datatype $\mathit{List}\, a$ **with** $foldr\_$ **has constructors** $cstr\colon\ () + a \times \mathit{List}\, a \to \mathit{List}\, a$

that is,

datatype $\mathit{List}\, a$ **with** $foldr\_$ **has constructors** $cstr\colon\ F(a, \mathit{List}\, a) \to \mathit{List}\, a \quad ,$

where $F(a,b) = () + a \times b \to b$. Having done so, we may put $nil = \mathit{inl} \,;\, cstr$ and $cons = \mathit{inr} \,;\, cstr$.

In the sequel we shall use $\alpha, \varphi, \psi, \ldots$ for functions of type $F(a,b) \to b$, for given functor F and arbitrary types a, b. So, effectively, each such function stands for a collection of functions, all with the same target type.

2.2 Datatypes in general. Let F be a 2-ary functor, and consider the declaration:

$$\textbf{datatype } Ta \textbf{ with } (\![\_]\!) \textbf{ has constructors } \alpha\colon\ F(a, Ta) \to Ta \quad .$$

For each a, the declaration defines the initial algebra with carrier Ta and operations α typed as above. Operation $(\![\_]\!)$ gives the unique homomorphism to any other algebra of the same signature. Thus, the entities $T, \alpha, (\![\_]\!)$ are completely characterised by:

$$\varphi\colon\ F(a,b) \to b \quad \Rightarrow \quad (\![\varphi]\!)\colon\ Ta \to b \qquad \text{cata-}\textsc{Type}$$

$$h = (\![\varphi]\!) \quad \equiv \quad \alpha \,;\, h = F(id, h) \,;\, \varphi \qquad \text{cata-}\textsc{Charn}$$

$$f\colon\ a \to b \quad \Rightarrow \quad Tf\colon\ Ta \to Tb \qquad \text{map-}\textsc{Type}$$

$$Tf \quad = \quad (\![\alpha \cdot F(f, id)]\!) \qquad \text{map-}\textsc{Def}$$

It follows that T is a functor, and that $(\![\_]\!)$ satisfies several other laws too:

$$\alpha \,;\, (\![\varphi]\!) \quad = \quad F(id, (\![\varphi]\!)) \,;\, \varphi \qquad \text{cata-}\textsc{Self}$$

$$(\![\alpha]\!) \quad = \quad id \qquad \text{cata-}\textsc{Id}$$

$$\varphi \,;\, f = F(id, f) \,;\, \psi \quad \Rightarrow \quad (\![\varphi]\!) \,;\, f = (\![\psi]\!) \qquad \text{cata-}\textsc{Fusion}$$

$$T\,id \quad = \quad id \qquad \text{map-}\textsc{Ftr}$$

$$T(f \,;\, g) \quad = \quad Tf \,;\, Tg \qquad \text{map-}\textsc{Ftr}$$

$$Tf \,;\, (\![\varphi]\!) \quad = \quad (\![F(f, id) \,;\, \varphi]\!) \qquad \text{map-cata-}\textsc{Fusion}$$

$$\varphi \,;\, f \ = \ F(f, f) \,;\, \psi \quad \Rightarrow \quad (\![\varphi]\!) \,;\, f \ = \ Tf \,;\, (\![\psi]\!) \qquad \text{cata-}\textsc{Trafo}$$

Law cata-Trafo asserts that $(\![\_]\!)$ is a transformer in the sense of Fokkinga [5]. For applications of these laws in actual program development we refer to Bird [3, 4], Meijer et al. [14], de Moor [15], and others.

Notice that the definitions make sense in any category; nowhere we have assumed that we are working in the category of sets with total functions. Of course, the declaration assumes that such entities exist; in ω-cocomplete categories the entities do exist for every regular functor F. (Sets with total functions form such a category, and the regular functors are defined in the sequel.)

Once more we stress the fact that the above declaration also covers datatypes with several constructors, in the way we have explained above. The reader may wish to generalise somewhat further by letting a be a variable ranging over a vector of n types, with n an arbitrary natural number. We also wish to mention that many-sorted datatypes are a special case of the above; see Fokkinga [7] and Wraith [20].

2.3 Monads. Let M be a functor. Say a function is M-*resultric* if it has type $a \to Mb$ for some types a and b. Suppose we wish to compose M-resultric functions, that is, we want to have a so-called M-*composition* $;_M$ satisfying the typing:

$$f\colon\ a \to Mb, \quad g\colon\ b \to Mc \quad \Rightarrow \quad f \,;_M\, g\colon\ a \to Mc \quad .$$

In view of the requirement the obvious definition of M-composition is:

$$f ;_M g \quad = \quad f \,;\, Mg \,;\, \mu \quad ,$$

where $\mu\colon\ MM \dot{\to} M$ is supposed to exist. Of course, one might wish the M-composition to be associative with some $\eta\colon\ I \dot{\to} M$ as unit. Given the definition above, that wish is exactly equivalent to the requirement that (M, η, μ) is a monad, that is, η and μ satisfy the following three so-called monad laws:

$$\begin{array}{lllll} \eta \,;\, \mu & = & id & = & M\eta \,;\, \mu \\ M\mu \,;\, \mu & = & \mu \,;\, \mu & & . \end{array}$$

The proof is easy and left to the reader. (The category with M-resultric functions as morphisms, η as identity, and $;_M$ as composition, is known as the *Kleisli* category.)

For later use we define the lifting operation $\_^M$ by:

$$f^M \quad = \quad f \,;\, \eta \quad : \quad a \to Mb \quad \text{for } f\colon\ a \to b \quad .$$

The lifting operation takes an arbitrary function into an M-resultric one in a trivial way. There are some nice algebraic properties involving $;_M$ and $\_^M$. For example:

$$\begin{array}{lll} g\colon\ M\text{-resultric} & \Rightarrow & f \,;\, g\colon\ M\text{-resultric} \\ (f \,;\, g) ;_M h & = & f \,;\, (g ;_M h), \quad \text{but in general } (f ;_M g) \,;\, h \neq f ;_M (g \,;\, h) \\ f \,;\, Mg ;_M h & = & f ;_M (g \,;\, h) \\ f^M ;_M g^M & = & (f \,;\, g)^M \\ f \,;\, g^M & = & (f \,;\, g)^M \\ f ;_M g^M & = & f \,;\, Mg \\ f^M ;_M g & = & f \,;\, g \quad , \end{array}$$

and presumably many more.

The usefulness of monads for structuring functional programs is amply demonstrated by Wadler [18]. We cannot and will not try to do what Wadler has done so convincingly; the examples below are only given to help those readers who have not had the opportunity to read Wadler's paper.

Example: lists. To give just one example of a monad, and the usefulness of the new kind of composition, consider lists: $(\mathit{List}, [\_], \mathit{concat})$ is a monad. The reader may now readily recognise many hidden uses of List-composition in his own programming work:

$$f ;_{\mathit{List}} g \quad = \quad f \,;\, \mathit{List}\, g \,;\, \mathit{concat} \quad ,$$

that is, each element in the result list of f is subject to g, separately, and all the result lists of g together are concatenated into one big list. List comprehension can be expressed by List-composition:

$$f ;_{\mathit{List}} g \quad = \quad x \mapsto [z \mid\ y \leftarrow fx \,;\, z \leftarrow gy]$$

$$\eta \quad = \quad x \mapsto [x] \quad .$$

Actually, reading from right to left, and generalising *List* to M, we can take these equations as *defining* a comprehension notation in terms of M-composition, for arbitrary monad (M, η, μ). This is the proposal of Wadler in *Comprehending Monads* [18]. When all programs are expressed "at the function level", as we will do, there is no need for the comprehension notation since the expression $f \,;_M\, g$ is much shorter and clearer than the comprehension form.

Example: exception handling. We speak of exception handling if functions may "signal an exception", in place of delivering a normal result, and, in general, exceptions are "propagated to the final outcome of the program". It is straightforward but clumsy to express this idea in a functional setting by using case-distinctions in each and every function. A more structured way is as follows.

Let exc be a type, in which exceptions can be recorded. Change the type of each function from $a \to b$, say, to $a \to b + exc$, thus giving it the possibility to "signal an exception". Now use $;_{Exc}$ to compose the functions in such a way that exceptions signaled by the composed functions are propagated to an exception of the composite:

$$\begin{array}{lcll} f \,;_{Exc}\, g & = & f \,;\, g + id_{exc} \,;\, id \triangledown inr & \\ & : & a \to c + exc & \text{for } f\colon\ a \to b + exc, \quad g\colon\ b \to c + exc \quad . \end{array}$$

This amounts to using a monad, the monad of exception handling. To be precise, let Exc, η, μ be defined as follows:

$$\begin{array}{lcllll} Exc\ a & = & a + exc & & & \\ Exc\ f & = & f + id_{exc} & : & a + exc \to b + exc, & \text{for } f\colon\ a \to b \\ \eta & = & inl & : & a \to a + exc & \\ \mu & = & id \triangledown inr & : & (a + exc) + exc \to a + exc & . \end{array}$$

Then (Exc, η, μ) is a monad, and $;_{Exc}$ is its associated composition for Exc-resultric functions.

Using Exc-composition, exception handling is well structured: the error-prone case-distinctions are expressed once and for all in the definition of the monad, and need not be repeated throughout the program text.

Example: state-based programming. Let $state$ be a type, and consider functions of type $a \times state \to b \times state$, for varying types a and b. When the state is passed from one function to another in such a way that the state is single-threaded through the computation, we speak of *state-based* programming and *state-based* functions. The prime example is imperative programming: there the state is the collection of all global variables, and it is passed implicitly to each and every function (which are called one after the other).

Defining a type mapping St by:

$$\mathit{St}\,x \quad = \quad \mathit{state} \to x \times \mathit{state} \quad ,$$

we find that state-based functions are St-resultric:

$$\begin{array}{ll} & a \times \mathit{state} \to b \times \mathit{state} \\ \cong & \\ & a \to (\mathit{state} \to b \times \mathit{state}) \\ = & \\ & a \to \mathit{St}\,b \quad . \end{array}$$

So, state-based functions may be composed by St-composition, provided that St is a monad. This happens to be true indeed, as we shall sketch now. First, closely following the action of St on types x defined above, we define the action of St on *functions* x by:

$$\mathit{St}\,x \quad = \quad f \mapsto \mathit{state} \,;\, f \,;\, x \times \mathit{state} \quad ,$$

which we may as well notate by $\mathit{St}\,x = \mathit{state} \to x \times \mathit{state}$. Thus defined St is a functor, as the reader may verify. Next, we define the monad operations by:

$$\begin{array}{lcl} \eta\,x & = & s \mapsto (x, s) \\ \mu\,f & = & s \mapsto gs' \text{ where } (g, s') = fs \quad . \end{array}$$

These do satisfy the monad laws. (By the way, calculations involving this monad may be easier by using the equalities: $\mathit{St} f = (;\, f \times \mathit{state})$, and $\eta = \mathit{pair}$ and $\mu = (;\, \mathit{apply})$.) Finally, it so happens that by St-composition the state is single-threaded through the computation:

$$\begin{array}{lcl} (f \,;_{\mathit{St}}\, g)x & = & s \mapsto (z, s'') \text{ where } \ (y, s') = f\,x\,s, \quad (z, s'') = g\,y\,s' \\ & = & f\,x \,;\, g \times \mathit{state} \,;\, \mathit{apply} \quad . \end{array}$$

Using St-composition, state-based programming is well structured: the error-prone state manipulations of the right-hand side are expressed once and for all in the definition of the monad, and need not be repeated throughout the program text.

3 Monadic catas and maps

It is known that each datatype comes equiped with its 'cata' and 'map'; these satisfy several laws that are very useful for transformational programming in an algebraic fashion. We shall derive, for arbitrary monad M, a sufficient condition under which there is also a kind of cata and map for M-resultric functions; these satisfy similar laws, and can thus be used for programming with M-resultric functions.

3.1 A direct approach. Let a bifunctor F be given, and let T, α be induced by F:

$$\textbf{datatype } Ta \textbf{ with } (\!|\_|\!) \textbf{ has constructors } \alpha\colon\ F(a, Ta) \to Ta \quad .$$

(We leave it to the reader to consider also non-parametrised type declaration, taking a to be a null vector of types and writing t for Ta and Fy for $F(a, y)$.) Let furthermore (M, η, μ) be a monad, with its associated $;_M$ and lifting operation $\_^M$.

We set out to define a so-called *monadic cata* $(\!|\_|\!)_M$, and *monadic map* T_M, that act on M-resultric functions:

$$\begin{array}{lcl} \varphi\colon\ F(a,b) \to Mb & \Rightarrow & (\!|\varphi|\!)_M\colon\ Ta \to Mb \\ f\colon\ a \to Mb & \Rightarrow & T_M\, f\colon\ Ta \to M\,Tb \quad . \end{array}$$

Their behaviour should be "similar" to the normal cata and map, except that now care is taken of the M-resultricness of the functions. Compare the typing with the normal cata and map when applied to M-resultric (rather than arbitrary) functions:

$$\begin{array}{lcl} \varphi\colon\ F(a, Mb) \to Mb & \Rightarrow & (\!|\varphi|\!)\colon\ Ta \to Mb \\ f\colon\ a \to Mb & \Rightarrow & T\, f\colon\ Ta \to T\, M\, b \quad . \end{array}$$

So, one might try to define the new cata and map in terms of the old ones, by composing the various ingredients with suitable natural transformations between the types involved. Specifically, consider a class of transformations that distribute M over another functor in one way:

$$dist_F \quad : \quad F\, M^* \dot{\to} M\, F \quad ,$$

where $F\, M^*(x_0, \ldots, x_{n-1}) = F(Mx_0, \ldots, Mx_{n-1})$. For 2-ary functor F, we use the notation Fa to denote the functor $F(a, \_)$, that is:

$$Fa \quad = \quad x \mapsto F(a, x) \quad .$$

Observe that $dist_{Fa}\colon\ F(a, Mb) \to M(F(a, b))$ for all b, since Fa is a 1-ary functor. So, using these distribution functions one may define:

$$\begin{array}{lcll} (\!|\varphi|\!) & = & (\!|dist_{Fa}\ ;\ M\varphi\ ;\ \mu|\!) & \text{for } \varphi\colon\ F(a,b) \to Mb \\ T_M\, f & = & T\, f\ ;\ dist_T & \text{for } f\colon\ a \to Mb \end{array}$$

or

$$T_M\, f \quad = \quad (\!|F(f, id)\ ;\ dist_F\ ;\ M\alpha|\!) \quad .$$

The definitions are suggested by type considerations alone: they are the obvious way to get a well-typed right-hand side of the required type. Sheard [16, 17] indeed presents these definitions (in a far less general and abstract way than ours). In Appendix A the derivation is shown in detail for the second definition of T_M.

It is not to be expected that such a $dist_F$ exists for arbitrary F. In the relational framework Backhouse et al. [1, Theorem 41] construct a natural transformation of type $FG \dot{\to} GF$ for a class of functors. In the construction they require that G belongs to a certain subclass of the regular functors; this seems not to be the case for the examples of $G = M$ that we have in mind. A direct construction of $dist_F$ for regular F will succeed, as we shall see in Section 4, but is unsatisfactory for the first tentative definition of T_M, since $dist_T$ will be defined in terms of T_M and so cannot be used to define T_M. This lurking circularity is avoided in the second tentative definition of T_M, since, in the formation of T according to the grammar for regular functors, F is a proper subterm of T.

3.2 An indirect approach. We shall indeed define the monadic cata and map as stated above, and prove the equality for both tentative definitions for T_M. But that is not the whole story. The next task is to investigate the properties they have. For this we might again proceed by analogy or similarity with the normal map and cata. That, however, is too naive: over and over again we would have to guess the formulas for the monadic case, and it is too easy to forget some laws. Thus we shall take a more fundamental approach. We shall compare the 'category of F-algebras (over the default category of functions)' with a certain 'category of F^M-algebras (over the category of M-resultric functions)', and establish an adjunction between the two, from which we can read off all the wanted definitions and laws, without further proof obligations!

3.3 Specialisation to lists. Before delving into the formal work, let us first consider an example, and judge whether we are on the right way. Specialising the above T_M to lists, that is, taking T and α the type functor and algebra induced by bifunctor $F(x, y) = () + x \times y$, we get exactly the monadic map for lists that one might have defined to begin with, as a kind of goal:

$$
\begin{array}{ll}
 & \mathit{Lst}_M\ f \\
= & \quad \text{proposed second definition of the monadic map} \\
 & (\![F(f, id)\ ;\ dist_F\ ;\ M\ cstr]\!) \\
= & \quad \text{definition of } F\text{, definition of } dist_F \text{ (given later)} \\
 & (\![id + f \times id\ ;\ (\eta\ ;\ M\ inl) \triangledown (dist_\times\ ;\ M\ inr)\ ;\ M\ cstr]\!) \\
= & \quad \text{sum-}\textsc{Fusion} \\
 & (\![(id\ ;\ \eta\ ;\ M\ inl\ ;\ M\ cstr) \triangledown (f \times id\ ;\ dist_\times\ ;\ M\ inr\ ;\ M\ cstr)]\!) \\
= & \quad \text{identity, functoriality } M\text{, and } nil \triangledown cons = cstr \\
 & (\![(\eta\ ;\ M\ nil) \triangledown (f \times id\ ;\ dist_\times\ ;\ M\ cons)]\!) \\
= & \quad \text{naturality } \eta\text{: } I \dot{\to} M\text{, definition } f^M = f\ ;\ \eta \\
 & (\![nil^M \triangledown (f \times id\ ;\ dist_\times\ ;\ M\ cons)]\!) \quad .
\end{array}
$$

That is,

$$\begin{array}{lcl} nil \,;\, \mathit{List}_M\, f & = & nil^M \\ cons \,;\, \mathit{List}_M\, f & = & f \times \mathit{List}_M\, f \,;\, dist_\times \,;\, M\, cons \quad . \end{array}$$

This is indeed what one might wish intuitively. If, in addition, we further specialise M to the monad St for state-based programming, it turns out that the state is single-threaded through the computation by the monadic map too:

$$\begin{array}{lcl} \mathit{List}_{St}\, f\, (nil()) & = & s \mapsto (nil(), s) \\ \mathit{List}_{St}\, f\, (cons(x, xs)) & = & s \mapsto f\, x\, s \,;\, id \times (\mathit{List}_{St}\, f)\, xs \,;\, assoc \,;\, cons \times id \quad , \end{array}$$

where $assoc$: $a \times (b \times c) \to (a \times b) \times c$, for all types a, b, c, is defined in the obvious way.

4 Construction of *dist*

The definition of the monadic maps and a class of natural transformations $dist$ will be mutually recursive. In this section we present the definition of $dist$. The following section has to be read in parallel, so to say.

4.1 Regular functors. The *regular* functors are built by composition from the constant functors, the extraction functors, the functors induced by regular (bi)functors, and the sum and product functors. A formal grammar for the n-ary regular functors $F^{(n)}$ with $n \geq 0$ is:

$$\begin{array}{lcll} F^{(n)} & ::= & \underline{a} & n\text{-ary constant functor} \\ & | & Ex_i^{(n)} & n\text{-ary extraction, } i = 0, \ldots, n-1 \\ & | & T \text{ induced by } F^{(n+1)} & \text{the type (map) functor induced by } F \\ & | & + \mid \times \qquad (\text{only if } n = 2) & \text{binary sum and product functor} \\ & | & F^{(k)}(F_0^{(n)}, \ldots, F_{k-1}^{(n)}) & \text{composition} \quad , \end{array}$$

where $F(G_0, \ldots, G_{k-1})$ is the functor that maps an n-tuple x to $F(G_0 x, \ldots, G_{k-1} x)$. Constant functor $\underline{a}$ maps each type to a and each function to id_a. For some extractions there are more familiar names: $Ex_0^{(1)}, Ex_0^{(2)}, Ex_1^{(2)} = I, Exl, Exr$.

4.2 Definition of *dist*. Let a monad (M, η, μ) be given. We shall define a class of natural transformations $dist$:

$$dist_F \quad : \quad F M^* \dot{\to} M F \quad ,$$

where

$$F G^* \quad = \quad x \mapsto F(Gx, \ldots, Gx) \quad .$$

So, here G^* stands for a k-tuple of identical functors G where k is the arity of F; and the arity of the composite FG^* is the arity of G. The definition is by induction on the formation of F according to the grammar above.

In the present paper we shall assume that $dist_\times$ is given; this one must be defined for each monad M separately, or, more generally, by induction for some inductively defined class of monads. Lambert Meertens (private communication) has shown that, if the category is cartesian closed and the monad functor M is strong, a function $dist_\times$ exists, but it has not the properties that one certainly wishes it to have.

We proceed by induction; the reader may wish to check, along the way, that the required naturality is achieved. On the right-hand side we give the typing requirement that follows from the naturality; it is to hold for all types (possibly tuples) a, b:

$$\begin{array}{lcll}
dist_{\underline{a}} & = & \eta & : a \to Ma \\
dist_I & = & id & : Ma \to Ma \\
dist_{Ex_i^{(n)}} & = & id & : Ma \to Ma \\
dist_+ & = & M\,inl \triangledown M\,inr & : Ma + Mb \to M(a+b) \\
dist_\times & & \text{assumed to be given} & \\
dist_{F(G,H)} & = & F(dist_G, dist_H) \,;\, dist_F & : F(G,H)M^*a \to MF(G,H)a \\
dist_T & = & T_M\, id & : TMa \to MTa \\
 & & \text{where } T \text{ is induced by bifunctor } F, \text{ and } id = id_{Ma} & .
\end{array}$$

For brevity we have defined $dist_{F(G,\ldots,H)}$ only for functors F with arity 2; the generalisation should be obvious. Notice that the monadic maps T_M will be defined in terms of $dist$, and now $dist_T$ is defined in terms of monadic map T_M. However, in the definition of T_M there will only occur the $dist_F$, and the $dist_F$ exists by the induction hypothesis for F. Notice that the monad operations η and μ are used (explicitly) in the first and (implicitly) in the last clause, respectively.

4.3 Properties of $dist$. By induction on the formation of F one can easily prove that several properties of $dist_\times$ carry over to $dist_F$. In particular:

$$\begin{array}{lcl}
\eta \times \eta \,;\, dist_\times \;=\; \eta & \Rightarrow & F\eta^* \,;\, dist_F \;=\; \eta \\
\mu \times \mu \,;\, dist_\times \;=\; dist_\times \,;_M\, dist_\times & \Rightarrow & F\mu^* \,;\, dist_F \;=\; dist_F \,;_M\, dist_F \quad .
\end{array}$$

where $Fx^* = F(x, \ldots, x)$ with as many x's as the arity of F.

Another property that is worthwhile to notice:

$$dist_{Fa} \quad = \quad F(\eta, id) \,;\, dist_F \quad ,$$

since $Fa = (x \mapsto F(a, x)) = F(\underline{a}, I)$.

4.4 Monad dependent *dist*-part. Consider the monad (St, η, μ) for state-based programming, discussed earlier. A definition for some $dist_\times$: $St\,X \times St\,Y \dot{\rightarrow} St\,(X \times Y)$, where $X, Y = Exl\,, Exr$, suggests itself:

$$dist_\times\,(f, g) \quad = \quad f\,;\,id \times g\,;\,assoc \quad ,$$

where $assoc$: $a \times (b \times c) \rightarrow (a \times b) \times c$ is defined in the obvious way. Notice that $dist_\times$ is not uniquely determined; another choice is:

$$dist_\times\,(f, g) \quad = \quad g\,;\,id \times f\,;\,swap \quad ,$$

where $swap$: $b \times (a \times c) \rightarrow (a \times b) \times c$ for all types a, b, c.

Similarly, in the category of Sets with total functions it is not hard to find a $dist_\times$ for the list monad and the exception monad.

5 Establishing an adjunction

Let a category $\mathcal{A}$ be given, for example $\mathcal{S}et$. Take this one as the *default* category, that is, whenever an indication of a category is missing, it is '$\mathcal{A}$' that has to be supplied. Let also a monad (M, η, μ) be given (in the default category thus). We shall establish an adjunction between certain categories of algebras. In the next section this adjunction is exploited by reading off the definition and properties of the monadic cata and map.

5.1 Assumption. Assume that there exists a natural transformation $dist_\times$:

$$dist_\times \quad : \quad MX \times MY \dot{\rightarrow} M(X \times Y) \qquad \text{where} \quad X, Y = Exl\,, Exr \quad ,$$

satisfying

$$\eta \times \eta\,;\,dist_\times \;=\; \eta \qquad \wedge \qquad \mu \times \mu\,;\,dist_\times \;=\; dist_\times \,;_M\, dist_\times \quad .$$

Alas, for $M = St$ the assumption is false. We return to this point in the conclusion.

5.2 Lifted and algebra categories. Four categories will play a role and need be well distinguished:

$\mathcal{A}$	:	the "universe of discourse"
$\mathcal{A}^M$	:	the M-Kleisli category, or "lifted" category
$\mathcal{A}lg(F)$	:	the category of F-algebras (in $\mathcal{A}$)
$\mathcal{A}lg^M(G)$	:	the category of G-algebras in $\mathcal{A}^M$.

The lifted category $\mathcal{A}^M$ has the same objects as $\mathcal{A}$, and its morphisms are M-resultric functions, its composition is the M-composition $;_M$, and its identity id_M is the M-unit η. In the term '$\mathcal{A}lg(F)$', F is required to be an endofunctor on $\mathcal{A}$; in the term

'$\mathcal{A}lg^M(G)$', G is required to be an endofunctor on $\mathcal{A}^M$. In the sequel we shall always take F^M for G, where F^M is a functor constructed from an endofunctor F on $\mathcal{A}$. The typing in $\mathcal{A}^M$ is denoted with $\to_M$, the typing in $\mathcal{A}lg(F)$ is denoted with $\to_F$:

$$\begin{array}{lcl}
f\colon a \to_M b & \equiv & f\colon a \to Mb \\
f\colon \varphi \to_F \psi & \equiv & \varphi \mathbin{;} f = Ff \mathbin{;} \psi \\
f\colon \varphi \to_{M_G} \psi & \equiv & \varphi \mathbin{;_M} f = Gf \mathbin{;_M} \psi \\
 & \equiv & \varphi \mathbin{;} Mf \mathbin{;} \mu = Gf \mathbin{;} M\psi \mathbin{;} \mu \quad .
\end{array}$$

The notations $a \to_M b$ and $a \to Mb$ are chosen similar, since they are semantically equal.

Barr and Wells [2] present an adjunction between $\mathcal{A}$ and $\mathcal{A}^M$. The two functors are:

$$\begin{array}{lcll}
_^M & : & \mathcal{A} \to \mathcal{A}^M & \text{"lifting" of } \mathcal{A} \\
a^M & = & a & \\
f^M & = & f \mathbin{;} \eta \quad (= \eta \mathbin{;} Mf) & \\
 & : & a \to_M b \qquad \text{whenever } \; fa \to b &
\end{array}$$

and

$$\begin{array}{lcl}
U & : & \mathcal{A}^M \to \mathcal{A} \\
Ua & = & Ma \\
Uf & = & Mf \mathbin{;} \mu \\
 & : & Ma \to Mb \qquad \text{whenever } \; f\colon a \to_M b \quad .
\end{array}$$

Thus, lifting raises the target of a function from b to Mb, and U "rebalances" this by further raising the source of an M-resultric function from a to Ma. The reader may wish to verify that $\_^M$ and U are functors indeed (and that $U \circ (\_^M) = M$). The unit and co-unit of the adjunction are η and id; it is trivial to verify the adjunction property:

$$f = \eta \mathbin{;} Ug \quad \equiv \quad f^M \mathbin{;_M} id = g \quad .$$

So, since $\_^M$ is a left adjoint, it preserves initiality. However, we are not interested in initiality in $\mathcal{A}$, but in $\mathcal{A}lg(F)$. (The initial object in $\mathcal{A} = \mathcal{S}et$ is the empty set.)

So, we set out to find an adjunction between $\mathcal{A}lg(F)$ and $\mathcal{A}lg^M(F^M)$. Such an adjunction gives a translation of an initial F-algebra with its associated cata $(\!|\_|\!)$ to an initial F^M-algebra and its associated cata in $\mathcal{A}^M$. Moreover, all the "standard" laws for initial algebras, like fusion properties, then also hold in $\mathcal{A}^M$. Thus it only remains to rephrase the definitions and properties so obtained in $\mathcal{A}^M$ in terms of $\mathcal{A}$, the universe of discourse we are interested in.

5.3 Lifting of functors. In order to follow the approach explained above, we must be able to "lift" an endofunctor F on $\mathcal{A}$ to an endofunctor F^M on $\mathcal{A}^M$. Type considerations alone suggest the definition:

$$F^M \quad : \quad \mathcal{A}^M \to \mathcal{A}^M \qquad \text{"lifting" of functor } F$$

$$
\begin{array}{lcl}
F^M a & = & Fa \\
F^M f & = & Ff \,;\, dist_F \\
 & : & F^M a \to M F^M b \quad \text{whenever} \quad f\colon\, a \to Mb \quad .
\end{array}
$$

Since $dist_F$ has been defined for regular functors F only, we can only lift regular functors in this way. One can easily show that F^M is a functor and that it relates nicely to F:

$$
\begin{array}{lcl}
F\mu \,;\, dist_F = dist_F \,;_M\, dist_F & \Rightarrow & F^M \text{ is a functor} \\
F\eta \,;\, dist_F = \eta & \Rightarrow & F^M f^M = (Ff)^M \quad .
\end{array}
$$

From the assumption (5.1) and property (4.3) we know that the premisses are true. For the record we mention that $I^M = I$ and $(FG)^M = F^M G^M$.

The definition of a 2-ary $F^M\colon \mathcal{A}^M \times \mathcal{A}^M \to \mathcal{A}^M$ for given bifunctor $F\colon \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is similar: generalise $a, f,$ and G to n-tuples, and place a * at appropriate places in the formulas.

Notice that for 2-ary functor F the 1-ary mapping $Ff = F(f, \_)$ is not a functor; in particular, it cannot sensibly map objects to objects in such a way that the functor axioms are satisfied. So the above definition do not define $(Ff)^M = (F(f,\_))^M$, and we are free to do that now. We put

$$
(Ff)^M \quad = \quad F^M f \quad = \quad F^M(f, \_) \quad ,
$$

so that for both types and functions x we have $(Fx)^M = F^M x$, hence $(Fx)^M y = F^M(x, y)$ and functor $(Fx)^M$ is functorial in x. Moreover, $(Fa)^M y = (F\,id_a)^M y$.

Remark. There is some circularity lurking: the definition of $dist$ makes use of the monadic maps for which we are trying to develop a definition just here! However, the definition of $dist_F$ and the monadic map T_M (for datatype T induced by regular functor F) is by induction on the construction of F according to the grammar for regular functors: arriving here at the task of defining the monadic map T_M, we may already assume the existence of $dist_F$ as an inductive hypothesis.

5.4 Lifting on the level of algebras. Since each object and morphism in $\mathcal{A}lg(F)$ is a morphism in $\mathcal{A}$, the lifting functor $\_^M$ also applies to algebras and homomorphisms. Moreover, since lifting distributes over compositions of morphisms, it also distributes over compositions of homomorphisms; and similarly for the identity. In addition, it preserves the other categorical structure as well, that is, F-algebras become F^M-algebras, and homomorphisms become homomorphisms:

$$
\begin{array}{lcll}
\varphi^M & : & F^M a^M \to_M a^M & \text{if } \varphi\colon\, Fa \to a \\
f^M & : & \varphi^M \to_{M F^M} \psi^M & \text{if } f\colon\, \varphi \to_F \psi \quad .
\end{array}
$$

So, lifting is also a functor on the level of algebras:

$$
\_^M \quad : \quad \mathcal{A}lg(F) \to \mathcal{A}lg^M(F^M) \quad .
$$

5.5 U on the level of algebras. We wish to define a functor $V\colon \mathcal{A}lg^M(F^M) \to \mathcal{A}lg(F)$. Type considerations alone suggest its action on objects, that is, on F^M-algebras in $\mathcal{A}^M$:

$$\begin{array}{rcl} V\varphi & = & dist_F \,;\, M\varphi \,;\, \mu \qquad (= dist_F \,;\, U\varphi \quad = \quad dist_F \,;_M\, \varphi\,) \\ & : & FMa \to Ma \quad \text{whenever } \varphi\colon F^M a \to Ma \quad . \end{array}$$

For its action on morphisms we argue as follows. Let F^M-algebras φ and ψ in $\mathcal{A}^M$ be arbitrary, say:

$$\begin{array}{rcl} \varphi & : & Fa \to Ma \\ \psi & : & Fb \to Mb \quad , \end{array}$$

and consider morphism $f\colon \varphi \to_{M_{F^M}} \psi$. Then, from the given typing of f in $\mathcal{A}lg^M(F^M)$ we find:

$$\begin{array}{l} \quad f\colon \varphi \to_{M_{F^M}} \psi \\ \equiv \\ \quad \varphi \,;_M\, f = F^M f \,;_M\, \psi \\ \equiv \\ \quad \varphi \,;\, Mf \,;\, \mu = Ff \,;\, dist_F \,;\, M\psi \,;\, \mu \\ \Rightarrow \\ \quad a = \operatorname{src} f \quad \wedge \quad \operatorname{tgt} f = Mb \quad , \end{array}$$

whereas, from the desired typing of Vf in $\mathcal{A}lg(F)$ we find:

$$\begin{array}{l} \quad Vf\colon V\varphi \to_F V\psi \\ \equiv \\ \quad V\varphi \,;\, f = FVf \,;\, V\psi \\ \equiv \\ \quad dist_F \,;\, M\varphi \,;\, \mu \,;\, Vf = FVf \,;\, dist_F \,;\, M\psi \,;\, \mu \\ \Rightarrow \\ \quad Ma = \operatorname{src} Vf \quad \wedge \quad \operatorname{tgt} Vf = Mb \quad . \end{array}$$

These type considerations suggest the action of V on morphisms:

$$\begin{array}{rcl} Vf & = & Mf \,;\, \mu \qquad (= Uf\,) \\ & : & Ma \to Mb \qquad\qquad \text{whenever } f\colon a \to Mb \quad , \end{array}$$

and, indeed, thus defined V preserves the typing:

$$f\colon \varphi \to_{M_{F^M}} \psi \qquad\qquad \Rightarrow \quad Vf\colon V\varphi \to_F V\psi$$

that is,

$$\varphi \,;_M\, f = F f \,;\, dist_F \,;_M\, \psi \quad \Rightarrow \quad dist_F \,;\, M\varphi \,;\, \mu \,;\, Vf = FVf \,;\, dist_F \,;\, M\psi \,;\, \mu \quad ,$$

as the reader may wish to check. Since V acts on morphisms the same as $U\colon \mathcal{A}^M \to \mathcal{A}$, it is immediate that V distributes over compositions and preserves the identity. So, V is a functor:

$$V \quad : \quad \mathcal{A}lg^M(F^M) \to \mathcal{A}lg(F) \quad .$$

5.6 Adjunction on the level of algebras. Taking η as the unit, and id as the co-unit, it is readily verified that $\_^M$ is left-adjoint to V:

$$f = \eta \mathbin{;} Vg \qquad \equiv \qquad f^M \mathbin{;_M} \mu = g \quad ,$$

for all $f\colon\ \varphi \to_F V(\psi)$ and $g\colon\ \varphi^M \to M_{F^M}\ \psi$.

6 Exploiting the adjunction

Having established the adjunction between $\mathcal{A}lg^M(F^M)$ and $\mathcal{A}lg(F)$, for regular functors F, we shall now exploit the adjunction, in particular the preservation of initiality by left adjoints: the initial F-algebra is mapped to an initial F^M-algebra. Thus we get definitions and properties of the monadic cata and map for free.

6.1 Monadic cata. Let F be a regular endofunctor, and t its induced type:

datatype t **with** $(\!|\_|\!)$ **has constructors** $\alpha\colon\ Ft \to t$.

Since lifting $\_^M\colon\ \mathcal{A}lg(F) \to \mathcal{A}lg^M(F^M)$ is a left adjoint, it preserves initiality, and so:

$$\begin{array}{ll} & \textbf{datatype}\ t\ \textbf{with}\ (\!|\_|\!)\ \textbf{has constructors}\ \alpha\colon\ Ft \to t \\ \equiv & \\ & \alpha\colon\ Ft \to t \text{ initial in } \mathcal{A}lg(F) \\ \Rightarrow & \\ & \alpha^M\colon\ F^M t^M \to M\ t^M \text{ initial in } \mathcal{A}lg^M(F^M) \\ \equiv & \\ & \alpha \mathbin{;} \eta\colon\ Ft \to Mt \text{ initial in } \mathcal{A}lg^M(F^M) \quad . \end{array}$$

Moreover, from the proof of 'left adjoints preserve initiality' we know, for example from Fokkinga and Meertens [8], how the cata in $\mathcal{A}lg^M(F^M)$, denoted $(\!|\_|\!)_M$, is expressed in terms of $(\!|\_|\!)$:

$$\begin{array}{lll} (\!|\varphi|\!)_M & = & [\![(\!|V\varphi|\!)]\!] \\ & = & (\!|V\varphi|\!)^M \mathbin{;_M} id \\ & = & (\!|V\varphi|\!) \\ & = & (\!|dist_F \mathbin{;_M} \varphi|\!) \qquad\qquad \text{Mcata-}\textsc{Def} \\ & = & (\!|dist_F \mathbin{;} M\varphi \mathbin{;} \mu|\!) \\ & : & t \to Mb \qquad \text{whenever } \varphi\colon\ Fb \to Mb \quad . \qquad \text{Mcata-}\textsc{Type} \end{array}$$

This is exactly the definition that one may find by type considerations alone. Since, for initial F-algebra α with cata $(\!|\_|\!)$, the F^M-algebra α^M is initial in $\mathcal{A}lg^M(F^M)$, we have immediately the laws for initial algebras in general:

$$f = (\!|\varphi|\!)_M \qquad \equiv \qquad \alpha^M \mathbin{;_M} f = F^M f \mathbin{;_M} \varphi \qquad \text{Mcata-}\textsc{Charn}$$

$$\alpha^M ;_M (\![\varphi]\!)_M = F^M (\![\varphi]\!)_M ;_M \varphi \qquad \text{Mcata-Self}$$

$$(\![\alpha^M]\!)_M = id_M \qquad \text{Mcata-Id}$$

$$\left.\begin{array}{l} \alpha^M ;_M f = F^M f ;_M \varphi \\ \alpha^M ;_M g = F^M g ;_M \varphi \end{array}\right\} \quad \Rightarrow \quad f = g \qquad \text{Mcata-Uniq}$$

$$\varphi ;_M f = F^M f ;_M \psi \quad \Rightarrow \quad (\![\varphi]\!)_M ;_M f = (\![\psi]\!)_M \quad . \qquad \text{Mcata-Fusion}$$

One may work out these formulas in entirely in terms of $\mathcal{A}$, the universe of discourse. But it is probably better to consider M-composition as an available derived operation, like the monadic comprehensions of Wadler, and also keep the lifting operation $\_^M$. So, it only remains to eliminate $F^M = f \mapsto Ff ; dist_F$ and $id_M = \eta$. This is left to the reader.

6.2 Monadic map. Let F be a regular bifunctor. Substituting $F := Fa$ in the above, and writing t_a as Ta, we have:

$$\begin{array}{ll} & \textbf{datatype } Ta \textbf{ with } (\![\_]\!) \textbf{ has constructors } \alpha\colon F(a, Ta) \to Ta \\ \Rightarrow & \\ & \alpha^M\colon (Fa)^M (Ta)^M \to\!M\ (Ta)^M \text{ is initial in } \mathcal{A}lg^M((Fa)^M) \\ \equiv & \\ & \alpha ; \eta\colon F(a, Ta) \to\!M\ Ta \text{ is initial in } \mathcal{A}lg^M(F^M a) \quad , \end{array}$$

and $a \mapsto Ta$ can be extended to a functor $T_M\colon \mathcal{A}^M \to \mathcal{A}^M$, by defining:

$$\begin{array}{rcll} T_M\, a & = & Ta & \\ T_M\, f & = & (\![F^M(f, id_M) ;_M \alpha^M]\!)_M & \\ & = & (\![F(f, \eta) ; dist_F ;_M \alpha^M]\!)_M & \\ & = & (\![F(f, id) ; dist_F ; M\alpha]\!) & \text{Mmap-Def} \\ & : & Ta \to\!M\ Tb \qquad \text{whenever } f\colon a \to\!M\ b \quad , & \text{Mmap-Type} \end{array}$$

where the last equality is calculated as follows:

$$\begin{array}{ll} & (\![F(f, \eta) ; dist_F ;_M \alpha^M]\!)_M \\ = & \qquad \text{definition monadic cata} \\ & (\![dist_{Fa} ;_M (F(f, \eta) ; dist_F ;_M \alpha^M)]\!) \\ = & \qquad \text{property } dist_{Fa} = F(\eta, id) ; dist_F \\ & (\![F(\eta, id) ; dist_F ;_M (F(f, \eta) ; dist_F ;_M \alpha^M)]\!) \\ = & \qquad \text{property } f ;_M (g ; h) = f ; Mg ;_M h \text{, and } ; \text{-} ;_M\text{-associativity} \\ & (\![F(\eta, id) ; dist_F ; M\, F(f, \eta) ;_M dist_F ;_M \alpha^M]\!) \\ = & \qquad \text{naturality } dist_F\colon FM^* \to MF \\ & (\![F(\eta, id) ; F(Mf, M\eta) ; dist_F ;_M dist_F ;_M \alpha^M]\!) \\ = & \qquad \text{functoriality } F \text{, and } ; \text{-} ;_M\text{-associativity, and } f ;_M g^M = f ; Mg \end{array}$$

$$
\begin{array}{ll}
 & (\![F(f,id) \,;\, F(\eta, M\eta) \,;\, (dist_F \,;_M\, dist_F) \,;\, M\alpha]\!) \\
= & \qquad \text{property } dist \\
 & (\![F(f,id) \,;\, F(\eta, M\eta) \,;\, F(\mu,\mu) \,;\, dist_F \,;\, M\alpha]\!) \\
= & \qquad \text{functoriality } F\text{, monad properties} \\
 & (\![F(f,id) \,;\, dist_F \,;\, M\alpha]\!) \quad .
\end{array}
$$

From the laws for maps in general we thus find:

$$
\begin{array}{lcll}
T_M\, id_M & = & id_M & \text{Mmap-}\textsc{Id} \\
T_M\, f \,;_M\, T_M\, g & = & T_M\,(f \,;_M\, g) & \text{Mmap-}\textsc{Distr} \\
T_M\, f \,;_M\, (\![\varphi]\!)_M & = & (\![F^M(f,id) \,;_M\, \varphi]\!)_M & \text{Mmap-Mcata-}\textsc{Fusion} \\
\varphi \,;_M\, f = F^M(f,f) \,;_M\, \psi & \Rightarrow & (\![\varphi]\!)_M \,;_M\, f = T_M\, f \,;_M\, (\![\psi]\!)_M \quad . & \text{Mcata-}\textsc{Trafo}
\end{array}
$$

Keeping again M-composition and M-cata as available derived operations, it only remains to eliminate the occurrences of $id_M = \eta$ and F^M; this is left to the reader.

6.3 Further laws. Since monadic maps are catamorphisms in the default category, one may use the laws for catamorphisms to derive more laws for the monadic map. For example:

$$
\begin{array}{lcll}
T f \,;\, T_M g & = & T_M(f \,;\, g) & \text{map-Mmap-}\textsc{Fusion} \\
T_M & = & T^M \quad . & \text{Mmap-mapM-}\textsc{Eq}
\end{array}
$$

So, from now on one might confuse T_M and T^M, since they are semantically the same. As a corollary it follows that $T_M\, f^M = T^M\, f^M = (T f)^M$. The proofs are easy; for the former law we calculate:

$$
\begin{array}{ll}
 & T f \,;\, T_M g \\
= & \qquad \text{definition monadic map} \\
 & T f \,;\, (\![F(g,id) \,;\, dist_F \,;\, M\alpha]\!) \\
= & \qquad \text{map-cata-}\textsc{Fusion} \\
 & (\![F(f,id) \,;\, F(g,id) \,;\, dist_F \,;\, M\alpha]\!) \\
= & \qquad \text{functoriality } F\text{, definition monadic map} \\
 & T_M(f \,;\, g) \quad .
\end{array}
$$

For the latter law, the equality $T_M\, x = T^M\, x$ for types x is immediate; for functions x we calculate, from right to left:

$$
\begin{array}{ll}
 & T^M f \\
= & \qquad \text{definition of lifting}
\end{array}
$$

$$
\begin{array}{cl}
 & T\,f\ ;\ dist_T \\
= & \quad \text{definition } dist_T \\
 & T\,f\ ;\ T_M\ id \\
= & \quad \text{previous law: map-Mmap-}\textsc{Fusion} \\
 & T_M\,(f\ ;\ id) \\
= & \quad \text{identity} \\
 & T_M\,f \quad .
\end{array}
$$

Notice that $T_M f = Tf\ ;\ dist_T$ has been proved along the way. So the two proposed definitions for T_M in paragraph 1 are equal.

6.4 Other applications. Here are some other applications of our fundamental approach to defining monadic catas and maps.

First, consider the following law, taken from Fokkinga [7]. Let F and G be functors, and let t, α and u, β be defined by:

$$
\begin{array}{l}
\textbf{datatype}\ t\ \textbf{with}\ (\!|t \mid \_|\!)\ \textbf{has constructors}\ \alpha\colon\ Ft \to t \\
\textbf{datatype}\ u\ \textbf{with}\ (\!|u \mid \_|\!)\ \textbf{has constructors}\ \beta\colon\ Gu \to u \quad .
\end{array}
$$

Then:

$$
\varepsilon\colon\ F \dot{\to} G \quad \Rightarrow \quad (\!|t \mid \varepsilon\ ;\ \beta|\!)\ ;\ (\!|u \mid \varphi|\!) = (\!|t \mid \varepsilon\ ;\ \varphi|\!) \quad . \qquad \text{cata-}\textsc{Compose}
$$

The "problem" is to find the generalisation to the case of monadic catas. Using the formulation for $\mathcal{A}lg^M(F^M)$ (which is an *instantiation* rather than a generalisation of the formula), we immediately have solved the problem:

$$
\varepsilon\colon\ F^M \dot{\to} M\ G^M \quad \Rightarrow \quad (\!|t \mid \varepsilon\ ;_M\ \beta^M|\!)_M\ ;_M\ (\!|u \mid \varphi^M|\!)_M = (\!|t \mid \varepsilon\ ;_M\ \varphi^M|\!)_M \quad .
$$

Fully worked out the premiss $\varepsilon\colon\ F^M \dot{\to} M\ G^M$ reads:

$$
\begin{array}{l}
\varepsilon\colon\ Fa \to MGa \\
Ff\ ;\ dist_F\ ;_M\ \varepsilon \quad = \quad \varepsilon\ ;\ MGf\ ;_M\ dist_G \quad ,
\end{array}
$$

for all types a and M-resultric functions f.

Second, we remind the reader of the fact that a lot more properties have been proved about catamorphisms and maps in general, that is, for arbitrary categories and datatypes. For example, Meertens [13] investigates the notion of paramorphism (functions inductively defined by primitive recursion), and Fokkinga [7] discusses prepromorphisms (functions inductively defined by recursion schemes that are more complex than primitive recursion), and presents the Banana Split law (asserting that tupling of catas yields a cata again), and so on. All their definitions and theorems do hold in $\mathcal{A}lg^M(F^M)$ as well, and so they translate to definitions and statements in $\mathcal{A}$.

6.5 Conclusion. The theorem proved in this paper gives the existence of a monadic map and fold. Thus "higher order" techniques are made available for programming tasks.

However, the theorem contains an assumption on monad M, given in paragraph 5.1, that is not valid for several known monads, in particular it is not valid for the monad of St for state-based programming. Hence, for such monads the mapping F^M is not a functor, and the notion of $\mathcal{A}lg^M(F^M)$, and therefore the whole reasoning, doesn't make sense!

Nevertheless, for some monads the assumption may be valid. Moreover, if the functor F that determines the datatype, doesn't involve the product functor $\times$, then too the assumed function $dist_F$ exists. In addition, some of results are true in general, for arbitrary monad and datatype. For example, whether or not the assumption is satisfied, one may *define* $(\!|\_|\!)_M$ and T_M in the way we have done, and then check what properties still hold. So it turns out that Mcata-CHARN, -SELF, -ID, and -UNIQ do hold in general! But functoriality of F is used to prove law cata-FUSION, hence functoriality of F^M will be needed for law Mcata-FUSION; this law is not valid when $M = \mathit{St}$. Similarly, in the calculation for law Mmap-DEF there is an application of the properties of $dist$; these are no longer valid if the assumption for M doesn't hold.

Acknowledgement. This paper started with an attempt to understand the direct construction by Tim Sheard [16, 17] of a monadic map and fold. His paper and subsequent discussion has been a great stimulus. Erik Meijer pointed out a slip of the pen, and suggested some notational improvements.

References

[1] R. Backhouse, H. Doornbos, and P. Hoogendijk. A class of commuting relators. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*. Utrecht University, Dept Computer Science, 1992.

[2] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[3] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer Verlag, 1987. Also Technical Monograph PRG–56, Oxford University, Computing Laboratory, Programming Research Group, October 1986.

[4] R.S. Bird. Lecture notes on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*. International Summer School directed by F.L. Bauer [et al.], Springer Verlag, 1989. NATO Advanced Science Institute Series (Series F: Computer and System Sciences Vol. 55).

[5] M.M. Fokkinga. Datatype laws without signatures. In *Computing Science in The Netherlands*, pages 231–248. Stichting Mathematisch Centrum, Amsterdam, 1991. Also Tech Report CS-R9133, CWI. Submitted for publication.

[6] M.M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992. An abbreviated version has appeared in Computing Science in The Netherlands 1991, published by Stichting Mathematisch Centrum, Amsterdam, pages 211–230, 1991.

[7] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[8] M.M. Fokkinga and L. Meertens. Adjunctions. Technical Report To appear, University of Twente, Enschede, The Netherlands, 1992.

[9] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[10] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 283 in Lect. Notes in Comp. Sc., pages 140–157. Springer Verlag, 1987.

[11] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, University of Groningen, The Netherlands, 1990.

[12] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, September 1990.

[13] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1990.

[14] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *Lect. Notes in Comp. Sc.*, pages 124–144. Springer Verlag, 1991.

[15] O. de Moor. *Categories, Relations and dynamic programming*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, April 1992. Also: Technical Monograph PRG-98.

[16] T. Sheard. Adding algebraic methods to traditional functional languages by using reflection. In *AMAST'93 (Algebraic Methods And Software Technology)*. Springer Verlag, June 1993.

[17] T. Sheard. Type parametric programming with compile-time reflection. Oregon Graduate Institute of Science and Technology, Beaverton OR, USA, June 1993.

[18] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, June 1990. To appear in Mathematical Structures in Computer Science.

[19] G.C. Wraith. Categorical datatypes — a critique of Hagino's thesis. Unpublished note, November 1988.

[20] G.C. Wraith. A note on categorical datatypes. In D.E. Rydeheard, P. Dybjer, A.M. Pits, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lect. Notes in Comp. Sc.*, pages 118–127. Springer Verlag, 1989.

A A direct construction of the monadic map

Let us try and construct T_M directly, for given bifunctor F. Type considerations alone suggest a satisfactory definition. Let $f\colon a \to M\,b$ be arbitrary. Then, using $\varphi, \psi, \chi, \mathit{dist}$ as auxiliary unknows:

$$
\begin{array}{ll}
 & T_M\,f\colon\ T a \to M\,T b \\
\equiv & \qquad \{\text{guess } T_M\,f \text{ is a catamorphism on } T a \text{ (what else?)}\} \\
 & \qquad \textbf{put } T_M\,f = (\!|\varphi|\!) \\
 & (\!|\varphi|\!)\colon\ T a \to M\,T b \\
\Leftarrow & \qquad \text{cata-Type} \\
 & \varphi\colon\ F(a, M\,T b) \to M\,T b \\
\equiv & \qquad \{\text{guess } \varphi = \ldots \mathbin{;} M\,\alpha \text{ since the result type matches}\} \\
 & \qquad \textbf{put } \varphi = \psi \mathbin{;} M\,\alpha \\
 & \psi \mathbin{;} M\,\alpha\colon\ F(a, M\,T b) \to M\,T b \\
\Leftarrow & \qquad \text{typing } \alpha \text{ and } M \\
 & \psi\colon\ F(a, M\,T b) \to M\,F(b, T b) \\
\equiv & \qquad \{\text{guess } \psi = F(f, \ldots) \mathbin{;} \ldots \text{ since } f \text{ has to be used on } a\ \} \\
 & \qquad \textbf{put } \psi = F(f, \chi) \mathbin{;} \mathit{dist} \\
 & F(f, \chi) \mathbin{;} \mathit{dist}\colon\ F(a, M\,T b) \to M\,F(b, T b) \\
\Leftarrow & \qquad \text{typing rules, and } f\colon\ a \to M\,b \\
 & F(\mathit{id}, \chi) \mathbin{;} \mathit{dist}\colon\ F(M\,b, M\,T b) \to M\,F(b, T b) \\
\equiv & \qquad \{\text{guess and}\}\ \textbf{put } \chi = \mathit{id} \\
 & \mathit{dist}\colon\ F(M\,b, M\,T b) \to M\,F(b, T b) \\
\Leftarrow & \qquad \{\text{guess } \mathit{dist} \text{ is independent of } T \text{ (and natural!)}\} \\
 & \mathit{dist}\colon\ F(M\,a, M\,b) \to M\,F(a, b) \qquad \text{for all types } a, b \quad .
\end{array}
$$

So, provided that such a dist exists, the typing requirement is met if we define:

$$
T_M\,f \quad = \quad (\!|F(f, \mathit{id}) \mathbin{;} \mathit{dist} \mathbin{;} M\,\alpha|\!) \quad .
$$