1991

M. M. Fokkinga, E. Meijer

Program calculation properties of continuous algebras

Computer Science/Department of Algorithmics and Architecture Report CS-R9104 January

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Program Calculation Properties of Continuous Algebras

Maarten M Fokkinga* and Erik Meijer†

Defining data types as initial algebras, or dually as final co-algebras, is beneficial, if not indispensible, for an *algebraic calculus for program construction*, in view of the nice equational properties that then become available. It is not hard to render finite lists as an initial algebra and, dually, infinite lists as a final co-algebra. However, this would mean that there are two distinct data types for lists, and then a program that is applicable to both finite and infinite lists is not possible, and arbitrary recursive definitions are not allowed. We prove the existence of algebras that are both initial in one category of algebras and final in the closely related category of co-algebras, and for which arbitrary (continuous) fixed point definitions ("recursion") do have a solution. Thus there is a single data type that comprises both the finite and the infinite lists. The price to be paid, however, is that partiality (of functions and values) is unavoidable.

We derive, for any such data type, various laws that are useful for an algebraic calculus of programs.

CR categories and descriptors: D11 [**Software**]: Programming Techniques — Applicative Programming, D31 [**Software**]: Programming Languages — Semantics, F32 [**Theory of Computation**]: Logic and meanings of Programs — Algebraic approach to semantics, F33 [**Theory of Computation**]: Logic and meaning of programs — type structure.

General terms: algorithm, design, theory.

Additional keywords and phrases: algebraic calculus for program construction, data type, algebra, complete partial ordered set (cpo), recursion, initiality, finality, anamorphism, catamorphism.

* CWI, PO Box 4079, NL 1009 AB Amsterdam (until July 1991), and University of Twente, dept INF, PO Box 217, NL 7500 AE Enschede (from July 1991)

† University of Nijmegen, Toernooiveld, NL 6525 ED, Nijmegen

Report CS-R9104
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1 Introduction

There are several styles and methodologies for the construction of computer programs. Some of these can be collectively called Transformational Programming; see Partsch [32]. It is one method in this class for which our results are important: the algebraic style of programming that has already been mentioned by Burstall & Landin [13] and, later, Backus [4]. More specifically, we are considering the *practical* method developed by Bird [8, 9] and Meertens [29], now being explored and extended by a number of researchers, e.g., [3, 40]. In the Bird-Meertens style of programming one derives a program from its specification by algebraic calculation. Bird [7, 9, 10] has identified several laws for specific data types and he has shown the practicality of the approach by deriving (by calculation) algorithms for various problems. The work of Malcolm [24, 25, 26] shows that quite a number of very important laws (such as the unique extension property and the fusion and promotion laws) come for free for *any* data type, when one defines a data type to be an initial algebra, or dually, a final algebra. Initiality and finality give several equivalences for free, and, quoting Dijkstra [15, page 17],

equivalences lie at the heart of any practical calculus.

The importance of initiality as a principle of proof has already been observed by Lehmann and Smyth [22]. Also Goguen [17] observes that initiality allows proofs by induction to be formulated without induction, and Backhouse [3] and Meertens [30] show the advantage of this for a *practical calculus* of programs: the induction-less proof steps are more compact and purely calculational. Finality is dual to initiality, and has therefore the same benefits. In fact, an early attempt to exploit this phenomenom for actual programming has been made by Aiello et al. [1].

All research done so far into the Bird-Meertens style of programming has considered only *total* elements and *total* functions (= programs). That is, partial functions (undefined for some arguments) and partial elements (having some undefined constituents) have not been allowed in the theory. The restriction to total elements and functions does not preclude infinite elements and (possibly never ending) programs that operate on infinite elements, as Malcolm [24] and Hagino [21, 20] have shown. Yet we know from computability theory that the restriction to totality precludes a large set of computable —though partial— functions and elements. Also, in practice it occurs that one would like to define a function by recursive definition or a **while** program without being able to prove its being total. (A well-known example of this is the function f defined by $f\,x = x$ **if** $x \leq 1$, $f(x/2)$ **if** x is even, $f(3x+1)$ **otherwise**. At present it is unknown whether f is total. As another example, Bird et al. [10] use recursion that is not allowed by the theory that they —implicitly— say to adhere to.) For us a major motivation for extending the theory with full recursion is the development of a transformational approach to semantics directed compiler generation: Meijer [31] applies the Bird-Meertens way of program development to denotational semantics. The price to be paid for the introduction of full recursion is that the elements of the data type and the functions defined on the data type may be partial, including possibly totally undefined ("bottom"). Recursion has been studied extensively, in particular in the field of denotational semantics. The novelty of our results is the *purely equational laws* that we are able to isolate in a framework where recursion is allowed.

Another drawback of the approach of Malcolm [24, 25, 26] and Hagino [21, 20] is that there is no initial data type that comprises both the finite and the infinite lists. As a consequence there is no inductively defined program that works for both finite and infinite lists, e.g., a

"map" that doubles each element of a list. Clearly, this is unfortunate; there are for example a lot of stream processing functions that make sense for finite streams as well. Our notion of data type makes it possible that both finite and infinite elements are in one initial data type.

Technically speaking, our notion of data type comes very close to the notion of *continuous algebra*, introduced in the field of semantics of programming languages, by Goguen et al. [18] and Reynolds [36]. Briefly, a continuous algebra is just an algebra whose carrier is a pointed cpo (complete partially ordered set with a least element), whose operations are continuous functions, and for which the homomorphisms are *strict* continuous functions. Goguen et al. [18] and Reynolds [36] claim that for arbitrary signature an initial continuous algebra exists. (But this is not completely true: if the algebra has only one operation, it has to be strict as well.) Following Wand [44] and Smyth & Plotkin [38] we present our construction in a categorical framework, using the notion of *order-enriched* category instead of the particular category *CPO* of pointed cpo's for which Reynolds proof is valid. It is then not hard to show that the construction has also the desired finality property.

Our extension to the current theory has been suggested in part to us by Ross Paterson. Actually, Paterson himself has done similar work [33]. Our results are very close to his, the main difference being that he has not shown the initiality and finality of the data types, and consequently is forced to use Fixpoint Induction where we can simply invoke the uniqueness brought forward by initiality and finality. As soon as these proofs have been done, the laws derived by Paterson are very similar to ours, and in some cases even more general. By the way, once we have proven the initiality *and* finality property of our notion of data type, we can immediately derive all the laws already proven by, say, Malcolm [24, 25, 26] since he uses only the initiality, *or* the finality. In addition we can derive some results that depend on *both* the initiality *and* finality of the data type.

The remainder of the paper consists of (a remark on notation, an example that recalls some terminology about lists, and) five sections. First we review some concepts that are well-known in category theory and we explain their relevance for the algebraic style of programming. Important are the notions of (co-)homomorphism and natural transformation, and in particular the notion of *catamorphism* (= homomorphism on an initial data type) and its dual, called *anamorphism*. Second we devote a section to the Main Theorem that asserts for some categories the existence of algebras with the desired properties. In particular we recall the definition of the category *CPO* and $CPO_\perp$. The least fixed point of the so-called *envelope*, termed *hylomorphism*, is studied here. In our framework the fusion law for hylomorphisms turns out to be slightly stronger than the fusion law for cata- and anamorphisms. Third we derive various laws that are useful for program calculation. This is done in Section 4 for general laws, in Section 5 for laws for 'map', and in Section 6 for 'reduce' and 'generate'. As we have said above, many laws are already known, but not all of them, and some are certainly not widely known.

We do not consider data types (algebras) with equations, like associativity of an operation. We expect that most of the theory will go through in the presence of equations, but at present we are unable to give an elegant formal treatment.

Notation For ease of memorisation and *formal manipulation* in actual program calculation we wish to be very systematic in the way we write terms, (abbreviated) equations and type assertions. After some experimentation we have decided to write consistently in a formula the part denoting a source (input) at the left, and the part denoting a target (output) at the

right. As a consequence, we write composition of programs ("morphisms") f and g as $f; g$ (pronounced "*f then g*"). This —not entirely unconventional— way of writing turns out to be very convenient for actual calculation.[1] It is only in some examples that we need to apply a program on some input; for consistency we write then $x.f$ for "x subject to f".

Example: some list concepts In various examples we shall use some concepts and notions of so-called cons-lists to illustrate the formal definitions. Here we explain the concepts informally.

The type of cons-lists over A, denoted $A\mathsf{L}$ (usually $A*$), is defined by

$$A\mathsf{L} \quad = \quad nil\ A \quad | \quad cons\ (A \times A\mathsf{L}).$$

This also defines the functions $nil : A \rightarrow A\mathsf{L}$ and $cons : A \times A\mathsf{L} \rightarrow A\mathsf{L}$. The carrier $A\mathsf{L}$ consists of precisely all results yielded by repeated applications of nil and $cons$ (using arbitrary elements of A). (Bird [8] uses join-lists with an associative join-operation of type $A\mathsf{L} \times A\mathsf{L} \rightarrow A\mathsf{L}$. We do not treat data type with laws, such as the associativity of join, and therefore use cons-lists rather than the more elegant join-lists.) The following functions are defined on lists. In the explanation we write $cons$ as an infix semicolon that associates to the right, and we abbreviate $x_{i..j} = x_i : x_{i+1} : \ldots : x_{j-1} : nil$ (excluding x_j and ending in nil), and $x = x_{0..n}$.

$head$	:	$x_{0..(n+1)} \mapsto x_0$
$tail$	:	$x_{0..(n+1)} \mapsto x_{1..(n+1)}$
$f\mathsf{L}$	:	$x \mapsto (x_0.f) : (x_1.f) : \ldots : (x_{n-1}.f) : nil$
		this is the so-called f-map, usually denoted $f*$
$\oplus/_e$	:	$x \mapsto x_0 \oplus (x_1 \oplus (\cdots x_{n-1} \oplus e))$
		this is the so-called (right-) reduce
$\oplus/$	=	$\oplus/_e$ where e is the left identity of $\oplus$, if it exists and is unique
$join$	:	$(x, y) \mapsto x_0 : x_1 : \ldots : x_{n-1} : y$
$inits$	:	$x \mapsto x_{0..0} : x_{0..1} : x_{0..2} : \ldots : x_{0..n} : nil$
$tails$	:	$x \mapsto x_{0..n} : x_{1..n} : x_{2..n} : \ldots : x_{n..n} : nil$
$segs$	:	$x \mapsto$ the list containing all $x_{i..j}$ for $i \in 0..n$ and $j \in i..n$, in this order.

More formally, these functions may be defined by

$$\begin{array}{lcl} (a : x).head & = & a \\ (a : x).tail & = & x \\ nil.f\mathsf{L} & = & nil \\ (a : x).f\mathsf{L} & = & (a.f) : (x.f\mathsf{L}) \end{array}$$

[1] Now, the left-to-right direction of *writing* and *reading* coincides with the direction of source-to-target, which is convenient indeed. Moreover, if we write $f \circ g$ for "f after g", then, in order to ease formal manipulation (such as type checking), we have to reverse all arrows, saying $f : A \leftarrow B$ rather than $f : B \rightarrow A$. Consequently, formula $f : in \rightarrow \phi$ (i.e., equation $in; f = f; \phi$) becomes $f : \phi \leftarrow in$ (i.e., equation $f; \phi = in; f$). It will be shown that the equation is a recursive definition of f where in stands for the formal parameter (a pattern) and $f; \phi$ is the defining expression; with *our* notation the formal pattern comes at the left (as we are used to), and the defining expression comes at the right.

$$
\begin{array}{lcll}
nil.\ \oplus/_e & = & e \\
(a:x).\ \oplus/_e & = & a \oplus (x.\ \oplus/_e) \\
(x,y).join & = & x.\ cons/_y \\
nil.inits & = & (nil) : nil \\
(a:x).inits & = & x.\ (inits;\ (a{:})\mathsf{L};\ (nil{:})) \\
nil.tails & = & (nil) : nil \\
(a:x).tails & = & (a:x) : (x.tails) \\
segs & = & inits;\ tails\mathsf{L};\ join/ & \text{the identity of } join \text{ is } nil\ .
\end{array}
$$

In the notation that we shall use in the sequel, the definition of *head* reads $cons;\ head = \grave{\pi}$; and similarly $cons;\ tail = \acute{\pi}$. The reader may convince herself that *inits* and *tails* can be defined as right reduces, and that

$$
\begin{array}{lcll}
\mathsf{id}\mathsf{L} & = & \mathsf{id} \\
(f;\ g)\mathsf{L} & = & f\mathsf{L};\ g\mathsf{L} & \text{the map distribution law} \\
join/;\ f\mathsf{L} & = & f\mathsf{LL};\ join/ & \text{the map promotion law} \\
join/;\ \oplus/ & = & \oplus/\mathsf{L};\ \oplus/ & \text{the reduce promotion law} \\
head;\ f & = & f\mathsf{L};\ head \\
tail;\ f\mathsf{L} & = & f\mathsf{L};\ tail \\
inits;\ f\mathsf{LL} & = & f\mathsf{L};\ inits \\
tails;\ f\mathsf{LL} & = & f\mathsf{L};\ tails \\
segs;\ f\mathsf{LL} & = & f\mathsf{L};\ segs\ .
\end{array}
$$

With the laws presented in this paper, we shall be able to prove the map distribution and both promotion laws without induction. The proof of the last equation, given the preceding ones, is easy:

$$
\begin{array}{ll}
 & f\mathsf{L};\ segs \\
= & \\
 & f\mathsf{L};\ inits;\ tails\mathsf{L};\ join/ \\
= & \\
 & inits;\ f\mathsf{LL};\ tails\mathsf{L};\ join/ \\
= & \\
 & inits;\ (f\mathsf{L};\ tails)\mathsf{L};\ join/ \\
= & \\
 & inits;\ (tails;\ f\mathsf{LL})\mathsf{L};\ join/ \\
= & \\
 & inits;\ tails\mathsf{L};\ f\mathsf{LLL};\ join/ \\
= & \\
 & inits;\ tails\mathsf{L};\ join/;\ f\mathsf{LL} \\
= & \\
 & segs;\ f\mathsf{LL}
\end{array}
$$

In the sequel we shall meet more economic ways to carry out such proofs. (We owe this particular example to Roland Backhouse.)

2 Categories and Algebras

For an elegant formalisation of the notion of (many-sorted) algebra, avoiding subscripts, signatures, and families of operator symbols and operations, the notion of functor is indispensable. This leads us to some terminology of category theory. This has two further advantages. First, it allows for a far going generalisation since the basic ingredients ('object'

and 'morphism') are not interpreted and thus any theorem proved about them holds for all interpretations that satisfy the categorical axioms. Second, if one uses only categorical notions one can dualise and thus obtain several results (and ideas and concepts) for free.

It is helpful if the reader knows the basic notions from category theory: category, initiality, duality, and functor. Pierce [35] gives a very readable tutorial. A more extensive and also fairly readable treatment of these notions, and the approach that we follow, is given by Manes & Arbib [28, Chapter 2 and Part 3]. If the reader is not familiar with these terms, she may still go on reading: just interpret 'object' as 'type', 'morphism' as 'typed function', and 'category' as 'a collection of types and typed functions'. (There are some axioms; these are just what is needed to make their use meaningful, so we shall not explain them.) The meaning or consequences of the remaining terms will be explained.

Basic nomenclature We let K vary over categories, $A, B, \ldots$ over *objects*, and $f, g, \ldots$ $\phi, \psi, \ldots$ over *morphisms*. Formula $f : A \rightarrow B$ **in** K asserts that f is a morphism *in* K with *source* A and *target* B, and the indication of the category is omitted if no confusion can result. (We allow for the notation $f : B \leftarrow A$ but we will not use it, since we write a source always at the left.) Composition is denoted by $;$ (pronounced 'then') so that

$$f ; g : A \rightarrow C \qquad \text{whenever} \qquad f : A \rightarrow B \text{ and } g : B \rightarrow C\,.$$

One (the) final object is denoted $\mathbf{1}$; it is characterised by the fact that for any A there is precisely one morphism from A to $\mathbf{1}$, denoted $!_A$. (As a type $\mathbf{1}$ is the one-element type.)

We let $\mathsf{F}, \mathsf{G}, \ldots$ vary over functors [2] (actually, endofunctors on K), and write them with postfix notation, hence having the highest priority in the parsing of a term. The identity functor is denoted I, i.e., $x\mathsf{I} = x$ for any object and morphism x. Object A, when used as a functor, is the constant functor defined by $xA = A$ for all objects x and $fA = \mathsf{id}_A$ for all morphisms f. We let $\dagger$ vary over bi-functors, written with infix notation.

Product and co-product We postulate the existence of the categorical product and co-product. These are denoted $\times, \vartriangle, \grave{\pi}, \acute{\pi}$ respectively $+, \triangledown, \grave{\iota}, \acute{\iota}$. The conventional notation for $f \vartriangle g$ ("f split g") is $\langle f, g \rangle$; we prefer an infix symbol and not to use commas, brackets or parentheses (these are too important too waste for this particular purpose). The choice of the symbol will be explained in a moment. Similarly, our $f \triangledown g$ ("f junc g") is usually written $[f, g]$. We call $\times, \vartriangle, +, \triangledown$ *combinators*. So, for any A, B, C and $f : A \rightarrow B$ and $g : A \rightarrow C$, and any $h : A \rightarrow C$ and $j : B \rightarrow C$, there exist $A \times B$ and $A + B$ and

$$\begin{array}{lll} f \vartriangle g : A \rightarrow B \times C & \grave{\pi} : A \times B \rightarrow A & \acute{\pi} : A \times B \rightarrow B \\ h \triangledown j : A + B \rightarrow C & \grave{\iota} : A \rightarrow A + B & \acute{\iota} : B \rightarrow A + B\,. \end{array}$$

The remaining axioms asserting that they form a categorical product and co-product read:

$$(1) \quad f = g \vartriangle h \quad \equiv \quad f ; \grave{\pi} = g \ \wedge \ f ; \acute{\pi} = h$$

$$(2) \quad f = g \triangledown h \quad \equiv \quad \grave{\iota} ; f = g \ \wedge \ \acute{\iota} ; f = h$$

Combinators $\times$ and $+$ are made into a bi-functor by defining its action on morphisms by

$$(3) \quad f \times g \quad = \quad (\grave{\pi} ; f) \vartriangle (\acute{\pi} ; g)$$

$$(4) \quad f + g \quad = \quad (f ; \grave{\iota}) \triangledown (g ; \acute{\iota})\,.$$

[2] A functor F is a mapping from objects to objects, and from morphisms to morphisms, such that $f\mathsf{F} : A\mathsf{F} \rightarrow B\mathsf{F}$ whenever $f : A \rightarrow B$, and $(f ; g)\mathsf{F} = f\mathsf{F} ; g\mathsf{F}$ and $\mathsf{id}_A\mathsf{F} = \mathsf{id}_{A\mathsf{F}}$.

Further we define

$$\begin{array}{llll} (5) & \Delta_A & = & \mathsf{id}_A \vartriangle \mathsf{id}_A \quad : \ A \to A \times A \\ (6) & \nabla_A & = & \mathsf{id}_A \triangledown \mathsf{id}_A \quad : \ A + A \to A \end{array}$$

so that $f \vartriangle g = \Delta; f \times g$ and $f \triangledown g = f + g; \nabla$, which explains our choice of the symbols $\vartriangle$ and $\triangledown$. One can prove various equations involving these combinators; here are some.

$$\begin{array}{rcl@{\qquad}rcl} f \times g; \grave{\pi} & = & \grave{\pi}; f & \grave{\iota}; f + g & = & f; \grave{\iota} \\ f \vartriangle g; \grave{\pi} & = & f & \grave{\iota}; f \triangledown g & = & f \\ f; g \vartriangle h & = & (f; g) \vartriangle (f; h) & f \triangledown g; h & = & (f; h) \triangledown (g; h) \\ \grave{\pi} \vartriangle \acute{\pi} & = & \mathsf{id} & \grave{\iota} \triangledown \acute{\iota} & = & \mathsf{id} \\ (h; \grave{\pi}) \vartriangle (h; \acute{\pi}) & = & h & (\grave{\iota}; h) \triangledown (\acute{\iota}; h) & = & h \\ f \vartriangle g; h \times j & = & (f; h) \vartriangle (g; j) & f + g; h \triangledown j & = & (f; h) \triangledown (g; j) \\ f \times g; h \times j & = & (f; h) \times (g; j) & f + g; h + j & = & (f; h) + (g; j) \\ f \vartriangle g = h \vartriangle j & \equiv & f = h \wedge g = j & f \triangledown g = h \triangledown j & \equiv & f = h \wedge g = j \end{array}$$

In parsing an expression, the combinators bind stronger than composition $;$ (and, in actual program texts, weaker than any other operation).

In some examples we postulate for a predicate p on object A the existence of a morphism

$$p? \quad : \quad A \to A + A$$

such that $p?$ maps its argument into the left component of $A + A$ if p holds for it, and into the right component otherwise. Thus $p?; f \triangledown g$ models the familiar **if** p **then** f **else** g. (One can construct such a $p?$ from $p : A \to \mathbf{1} + \mathbf{1}$ and the δ_R introduced below.)

Polynomial functors For mono-functors F and G and bi-functor $\dagger$ (like $\times$ and $+$) we define mono-functors FG and $(\mathsf{F} \dagger \mathsf{G})$:

$$\begin{array}{lcl} x(\mathsf{F}\,\mathsf{G}) & = & (x\mathsf{F})\mathsf{G} \\ x(\mathsf{F}\dagger\mathsf{G}) & = & x\mathsf{F} \dagger x\mathsf{G}\,. \end{array}$$

In view of the first equation we need not write parentheses in $x\mathsf{FG}$; it is clear from the font used for functors that the first 'juxtaposition' is functor application, and the second 'juxtaposition' is functor composition. Notice that in $(\mathsf{F} \dagger \mathsf{G})$ the bi-functor $\dagger$ is "lifted" to act on functors rather than objects or morphisms; $(\mathsf{F} \dagger \mathsf{G})$ itself is a mono-functor. The functors generated by

$$\mathsf{F} \quad ::= \quad \mathsf{I} \mid A \mid \mathsf{FG} \mid (\mathsf{F} \times \mathsf{G}) \mid (\mathsf{F} + \mathsf{G})$$

are called the *polynomial* functors. In Section 5 we shall see that also any data type definition induces a new functor; e.g., with $A\mathsf{L}$ the usual data type of lists over A, L is at the same time defined on morphisms in such a way that $f\mathsf{L}$ is the function that applies f to each member in the list. A frequently occurring functor is $\mathsf{I} \times \mathsf{I}$ which we denote by II. For example, a binary operation on A has type $A\mathsf{II} \to A$.

Algebras

Let K be a category and F be a functor, fixed throughout the sequel. Unless stated otherwise explicitly, any object is an object in K and any morphism is a morphism in K.

F-algebra An F-*algebra* is a pair (A, ϕ) where A is an object, called the *carrier* of the algebra, and $\phi : A\mathsf{F} \to A$ is a morphism, called the *operation* of the algebra. Thus we allow for only *one* carrier, only *one* operation, that takes only *one* argument. This makes the definition so simple, and the theorems and proofs so readable in comparison with traditional formalisations of many-sorted algebra; cf. Goguen et al. [18], Ehrig & Mahr [16], Reynolds [36] and so on. Yet, when the product and co-product exist, the definition *generalises* the traditional notion of algebra. Here are some examples.

- Suppose ϕ is to be a binary operation $A \times A \to A$. Then we can take $\mathsf{F} = \mathsf{II}$; indeed, $\phi : A\mathsf{F} \to A$.
- Suppose ϕ is to model a pair of operations, say $\psi : A\mathsf{G} \to A$ and $\chi : A\mathsf{H} \to A$. Then we can take $\mathsf{F} = \mathsf{G} + \mathsf{H}$ and $\phi = \psi \triangledown \chi$; indeed, $\phi : A\mathsf{F} \to A$.
- The carrier of a many-sorted algebra can be modeled as the sum of the constituent carriers plus an extra summand $\mathbf{1}$ that plays the role of of an "exception" carrier for the outcome of a function when it is supplied with an element not in its domain. For example, suppose A is to model two carrier sets B and C. Then we take $A = (B + C) + \mathbf{1}$. A function $f : ... \to B$ can now be modeled by $f ; \grave{\iota} ; \grave{\iota} : ... \to A$. A function $g : B \to ...$ can be modeled by $((g ; \grave{\iota}) \triangledown (!_C ; \acute{\iota})) \triangledown (!_{\mathbf{1}} ; \acute{\iota}) : A \to ... + \mathbf{1}$. ($!_X$ is the unique morphism from X to $\mathbf{1}$.)
- Suppose A is to model two carrier sets, B and C say, and ϕ is to model an operation $\psi : B \to C$. Then we take $A = (B + C) + \mathbf{1}$, as explained above. Further we take

  $$\phi = ((\psi ; \grave{\iota} ; \acute{\iota}) \triangledown (!_C ; \acute{\iota}) \triangledown (!_{\mathbf{1}} ; \acute{\iota})$$

  and $\mathsf{F} = \mathsf{I}$. Indeed, $\phi : A\mathsf{F} \to A$.
- Finally, suppose again that A is to model two carrier sets B and C, but now ϕ is to model an operation $\psi : B \times B \to C$. As explained above we take $A = (B + C) + \mathbf{1}$. Further we need to postulate a morphism $\delta_R : X \times (Y + Z) \to (X \times Y) + (X \times Z)$ satisfying $f \times (g + h) ; \delta_R = \delta_R ; (f \times g) + (f \times h)$. Omitting the details we claim that one can now construct a similar morphism δ_L (and vice versa), and then a similar morphism δ satisfying

  $$(f_1 + g_1 + h_1) \times (f_2 + g_2 + h_2) ; \delta = \delta ; (f_1 \times f_2) + \ldots + (h_1 \times h_2).$$

  Now we take $\mathsf{F} = \mathsf{II}$ and

  $$\phi = \delta ; (\psi ; \grave{\iota} ; \acute{\iota}) \triangledown \ldots \triangledown (!_{\mathbf{1} \times \mathbf{1}} ; \acute{\iota}).$$

  Indeed, $\phi : A\mathsf{F} \to A$.

Example: naturals Consider the usual algebra of *natural numbers* with one carrier and two operations; $\mathbb{N} = \{0, 1, \ldots\}$ is the carrier, the constant $0 \in \mathbb{N}$ is modeled with the 'nullary' operation $zero : \mathbf{1} \to \mathbb{N}$, and the successor operation with $suc : \mathbb{N} \to \mathbb{N}$. Taking $\mathsf{F} = \mathbf{1} + \mathsf{I}$ we have that $zero \triangledown suc : \mathbb{N}\mathsf{F} \to \mathbb{N}$ and $(\mathbb{N}, zero \triangledown suc)$ is an F-algebra indeed.

Example: lists The algebra of *cons-lists over* A has the form $(A\mathsf{L}, nil \triangledown cons)$ where $A\mathsf{L}$ is the set of lists over A, $nil : \mathbf{1} \rightarrow A\mathsf{L}$ models the nullary operation yielding the empty list, and $cons : A \times A\mathsf{L} \rightarrow A\mathsf{L}$ models the binary operation that from $a \in A$ and $l \in A\mathsf{L}$ constructs a list $cons(a, l) \in A\mathsf{L}$. Taking $\mathsf{F} = \mathbf{1} + (A \times \mathsf{I})$ we have that $nil \triangledown cons : A\mathsf{LF} \rightarrow A\mathsf{L}$, and $(A\mathsf{L}, nil \triangledown cons)$ is an F-algebra. (We shall later see that the L used here is a functor too.)

Example: rose trees The algebra of *rose trees* over A (multi-branching trees with elements from A at the tips) has the form $(A\mathsf{R}, tip \triangledown fork)$ where $tip : A \rightarrow A\mathsf{R}$ and $fork : A\mathsf{RL} \rightarrow A\mathsf{R}$ (with L from the previous example). Taking $\mathsf{F} = A + \mathsf{L}$ we have that $tip \triangledown fork : A\mathsf{RF} \rightarrow A\mathsf{R}$, and so $(A\mathsf{R}, tip \triangledown fork)$ is an F-algebra indeed.

Homomorphisms Intimately related to algebra is the notion of homomorphism. Given two F-algebras (A, ϕ) and (B, ψ), morphism h is an F*-homomorphism from* (A, ϕ) *to* (B, ψ) if $h : A \rightarrow B$ is a morphism in K and

$$(7) \qquad \phi; h \;=\; h\mathsf{F}; \psi, \qquad \text{which we denote by} \qquad h : \phi \xrightarrow{\mathsf{F}} \psi \qquad \textsc{Homo Def}$$

In our formalism the conventional lawless algebras become F-algebras with a polynomial functor F. So the reader may wish to check that for $\mathsf{F} = \mathsf{I}$, $\mathsf{F} = A$, $\mathsf{F} = \mathsf{II}$ and indeed for any polynomial functor, HOMO DEF (7) says that h "respects" or or "commutes with" the operation(s) of the algebra. For functors F that are not polynomial, the notion of F-algebra does not correspond to a familiar thing, and in this case HOMO DEF (7) is just the requirement that *by definition* declares h into a homomorphism from $(A, \phi : A\mathsf{F} \rightarrow A)$ to $(B, \psi : B\mathsf{F} \rightarrow B)$.

Homomorphisms play a very important rôle in program calculation: equation (7) gives two different ways of computing the same morphism (function). In particular, $\phi; h$ may be a part of a program, and the equation says that we may exchange it for $h\mathsf{F}; \psi$. Thus operation h is "promoted" (in the sense of Bird's [6] 'Promotion and Accumulation strategies', and Darlington's [14] 'filter promotion') from being a post-process of ϕ into being a pre-process for ψ. In view of this use we will pronounce $h : \phi \xrightarrow{\mathsf{F}} \psi$ as "h is $\phi \xrightarrow{\mathsf{F}} \psi$ promotable". The equation, when used from left to right in an actual program, may have a drastic, beneficial, effect on the computation time of the program, if ϕ and ψ are costly operations acting on sets, and h is a kind of filter that throws away part of its argument. On the other hand, using the equation from right to left can also be an efficiency improvement, e.g., when $\phi = \psi =$ summing the numbers in a list and h is multiplication by seven. But mostly the exchange of one side of the equation for the other side will be done only in order to make future transformations possible, without aiming at an immediate efficiency improvement.

Example Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ mapping n to 2^n, i.e., $zero; f = one$ and $suc; f = f; double$. This function is a $(\mathbf{1}+\mathsf{I})$-homomorphism from $(\mathbb{N}, zero \triangledown suc)$ to $(\mathbb{N}, one \triangledown double)$, and we write $f : zero \triangledown suc \xrightarrow{\mathbf{1}+\mathsf{I}} one \triangledown double$, since

$$\begin{array}{ll} & zero \triangledown suc; f \\ = & (zero; f) \triangledown (suc; f) \\ = & (\mathsf{id}_{\mathbf{1}}; one) \triangledown (f; double) \\ = & \mathsf{id}_{\mathbf{1}} + f; one \triangledown double \end{array}$$

$= \quad f(\mathbf{1}+\mathsf{I});\ one \triangledown double\,.$

Referring to the explanation of some list concepts in the introduction, we can formulate most of the given equations as homomorphism assertions in two ways, e.g.,

$$f \;:\; head \xrightarrow{\mathsf{L}} head \quad \text{and also:} \quad head \;:\; f\mathsf{L} \xrightarrow{\mathsf{I}} f$$
$$f\mathsf{L} \;:\; join/ \xrightarrow{\mathsf{L}} join/ \quad \text{and also:} \quad join/ \;:\; f\mathsf{LL} \xrightarrow{\mathsf{I}} f\mathsf{L}\,.$$

This phenomenom is formulated below as law HOMO SWAP (13).

We urge the reader to become intimately familiar with the notation $f : \phi \xrightarrow{\mathsf{F}} \psi$ since it will be used throughout the paper. To this end the reader may pause here and work out the equations of the introductory example. (A category theoretician might recognize a commuting square diagram in the formula $f : \phi \xrightarrow{\mathsf{F}} \psi$, even if ϕ, ψ, F are composite as in the laws below.)

The category of F-algebras We have argued that homomorphisms are computationally relevant. They are also calculationally attractive since they satisfy a lot of algebraic properties. The first two are very important and frequently used.

(8)	$\mathsf{id} : \phi \xrightarrow{\mathsf{F}} \phi$	HOMO ID
(9)	$f; g : \phi \xrightarrow{\mathsf{F}} \chi \quad \Leftarrow \quad f : \phi \xrightarrow{\mathsf{F}} \psi \;\wedge\; g : \psi \xrightarrow{\mathsf{F}} \chi$	HOMO COMPOSE

and moreover

(10)	$f\mathsf{F} : \phi\mathsf{F} \xrightarrow{\mathsf{I}} \psi\mathsf{F} \quad \Leftarrow \quad f : \phi \xrightarrow{\mathsf{I}} \psi$	HOMO DISTR
(11)	$f : \phi_1 \triangledown \phi_2 \xrightarrow{\mathsf{F}_1+\mathsf{F}_2} \psi_1 \triangledown \psi_2 \quad \equiv \quad f : \phi_i \xrightarrow{\mathsf{F}_i} \psi_i \;\; (i = 1, 2)$	HOMO SUMFCTR
(12)	$f : g; \phi \xrightarrow{A} g; \psi \quad \Leftarrow \quad f : \phi \xrightarrow{A} \psi$	HOMO AFCTR
(13)	$f : \phi \xrightarrow{\mathsf{F}} \phi \quad \equiv \quad \phi : f\mathsf{F} \xrightarrow{\mathsf{I}} f$	HOMO SWAP
(14)	$f : f\mathsf{F} \xrightarrow{\mathsf{F}} f \quad \Leftarrow \quad f : A\mathsf{F} \to A$	HOMO TRIV

And so on. HOMO COMPOSE (9) says that homomorphisms compose nicely; together with HOMO ID (8) it asserts that F-algebras form a category, called $\mathsf{F}\text{-Alg}(K)$. An object in this category $\mathsf{F}\text{-Alg}(K)$ is an F-algebra (A, ϕ), and a morphism $h : (A, \phi) \to (B, \psi)$ in $\mathsf{F}\text{-Alg}(K)$ is a morphism $h : A \to B$ in K that satisfies $h : \phi \xrightarrow{\mathsf{F}} \psi$. Composition is taken from K, and so are the identities. Initiality in $\mathsf{F}\text{-Alg}(K)$ turns out to be an important notion; we will define and discuss it below.

Example Let us write $\to$ for $\xrightarrow{\mathsf{I}}$. Then we have $inits, tails : f\mathsf{L} \to f\mathsf{LL}$ and $join/ : f\mathsf{LL} \to f\mathsf{L}$. We can now prove the equation $f\mathsf{L}; segs = segs; f\mathsf{LL}$ rather simply, thanks to the notation and laws for homomorphisms.

$segs : f\mathsf{L} \to f\mathsf{LL}$

$\Leftarrow$ definition *segs*, HOMO COMPOSE (9)

$inits : f\mathsf{L} \to f\mathsf{LL}, \quad tails\mathsf{L} : f\mathsf{LL} \to f\mathsf{LLL}, \quad join/ : f\mathsf{LLL} \to f\mathsf{LL}$

$\Leftarrow$ for the middle conjunct: HOMO DISTR (10);
given equations (taking $f := f\mathsf{L}$ for the right conjunct)

$\mathsf{true}\,.$

Co-algebras By dualisation[3] we obtain the following concepts and definitions. An F*-co-algebra over* K is a pair (A, ϕ) with $\phi : A \to A\mathsf{F}$ in K. A morphism h is called an F*-co-homomorphism from* (A, ϕ) *to* (B, ψ) if $h : A \to B$ in K and

$$(15) \qquad \phi ; h\mathsf{F} = h ; \psi , \quad \text{which we denote by} \quad h : \phi \overset{\mathsf{F}}{\succ\!\!-} \psi \qquad \text{CoHomo Def}$$

The equations denoted by $h : \phi \overset{\mathsf{F}}{\to} \psi$ and $h : \phi \overset{\mathsf{F}}{\succ\!\!-} \psi$ are easy to remember: in both cases ϕ is followed by h (and h precedes ψ), and the position of $>$ on the arrow indicates which occurrence of h is subject to functor F. Notice also that in both cases, as a morphism in K, the source of h is the carrier that "belongs to" ϕ (and the target of h is the carrier that "belongs to" ψ). (We allow for the notation $h : \psi -\!\!\prec \phi$ but we will not use it, since we write sources at the left and targets at the right.)

We have

$$(16) \qquad \mathsf{id} : \phi \overset{\mathsf{F}}{\succ\!\!-} \phi \qquad \text{CoHomo Id}$$

$$(17) \qquad f ; g : \phi \overset{\mathsf{F}}{\succ\!\!-} \chi \;\Leftarrow\; f : \phi \overset{\mathsf{F}}{\succ\!\!-} \psi \;\wedge\; g : \psi \overset{\mathsf{F}}{\succ\!\!-} \chi \qquad \text{CoHomo Compose}$$

$$(18) \qquad f\mathsf{F} : \phi\mathsf{F} \overset{\mathsf{I}}{\succ\!\!-} \psi\mathsf{F} \;\Leftarrow\; f : \phi \overset{\mathsf{I}}{\succ\!\!-} \psi \qquad \text{CoHomo Distr}$$

$$(19) \qquad f : \phi_1 \vartriangle \phi_2 \overset{\mathsf{F}_1 \times \mathsf{F}_2}{\succ\!\!-} \psi_1 \vartriangle \psi_2 \;\equiv\; f : \phi_i \overset{\mathsf{F}_i}{\succ\!\!-} \psi_i \;\; (i = 1, 2) \qquad \text{CoHomo prodFctr}$$

$$(20) \qquad f : \phi ; g \overset{A}{\succ\!\!-} \psi ; g \;\Leftarrow\; f : \phi \overset{A}{\succ\!\!-} \psi \qquad \text{CoHomo AFctr}$$

$$(21) \qquad f : \phi \overset{\mathsf{F}}{\succ\!\!-} \phi \;\equiv\; \phi : f \overset{\mathsf{I}}{\succ\!\!-} f\mathsf{F} \qquad \text{CoHomo Swap}$$

$$(22) \qquad f : f \overset{\mathsf{F}}{\succ\!\!-} f\mathsf{F} \;\Leftarrow\; f : A \succ\!\!- A\mathsf{F} \qquad \text{CoHomo Triv}$$

Example: finite lists Consider the algebra of cons-lists discussed earlier: $(A\mathsf{L}, \mathit{nil} \triangledown \mathit{cons})$ is an F-algebra with $\mathsf{F} = \mathbf{1} + (A \times \mathsf{I})$. Let $\mathit{empty?} : A\mathsf{L} \to A\mathsf{L} + A\mathsf{L}$ be the test on emptyness. Then $(A\mathsf{L}, \;\; \mathit{empty?} ; !_L + (\mathit{head} \vartriangle \mathit{tail}))$ is an F-co-algebra. The operation of this co-algebra has type $A\mathsf{L} \to \mathbf{1} + (A \times A\mathsf{L})$ and decomposes a list into its constituents (the constituents of *nil* being the sole member of $\mathbf{1}$).

Example: infinite lists Consider the carrier $A\mathsf{M}$ of infinite lists and the two total operations $\mathit{head} : A\mathsf{M} \to A$ and $\mathit{tail} : A\mathsf{M} \to A\mathsf{M}$. Taking $\mathsf{F} = A + \mathsf{I}$, we see that $(A\mathsf{M}, \mathit{head} \triangledown \mathit{tail})$ is an F-co-algebra. Clearly, *head* and *tail* decompose a list into its two constituents.

As is the case with initiality for algebras, finality for co-algebras will turn out to be an important concept and will be discussed below.

Initiality and catamorphisms We explain (and define) here what initiality means in the category $\mathsf{F}\text{-Alg}(K)$. Let (L, in) be an object in $\mathsf{F}\text{-Alg}(K)$. By definition, (L, in) is initial in $\mathsf{F}\text{-Alg}(K)$ if: for any A and $\phi : A\mathsf{F} \to A$ there exists precisely one $f : L \to A$ satisfying

$$(\heartsuit) \quad \mathit{in} ; f \;=\; f\mathsf{F} ; \phi \qquad \text{or, equivalently,}$$

$$(\diamondsuit) \quad f \;=\; \mathit{out} ; f\mathsf{F} ; \phi$$

[3] Roughly said, dualisation is the process of interchanging everywhere the source and target, and the operands of composition.

where $out : L \to L\mathsf{F}$ is the inverse of in, which is easily shown to exist (see Section 4). We call in the *constructor* and out the *destructor* of the initial algebra. Interpreted in **Set** this equation may be read as an inductive definition of f: it says that the result of f on any element in L equals what one obtains by applying $f\mathsf{F}$ to the constituents of the element and subjecting these to ϕ. The phrase "there exists precisely one such f" means that for (L, in) this kind of "inductive definition" is well defined, i.e., does define uniquely a morphism f indeed. The unique solution for f in these equations is denoted $(\![\mathsf{F}|\ \ \phi]\!)$ or simply $(\![\phi]\!)$ if F is understood. So initiality of (L, in) in $\mathsf{F}\text{-Alg}(K)$ is —apart from typing— fully captured by the law

$$f = (\![\mathsf{F}|\ \ \phi]\!) \quad \equiv \quad f : in \xrightarrow{\mathsf{F}} \phi \qquad\qquad \text{CATA UNIQ}$$

$(\![\mathsf{F}|\ \ \phi]\!)$ is called an F*-catamorphism* ($\kappa\alpha\tau\alpha$ meaning 'downwards') since, interpreted as a computing agent, $(\![\phi]\!)$ descends along the structure of the argument (systematically replacing each in by ϕ, see the example below). So a catamorphism is nothing but a homomorphism on an initial algebra. It is useful to have a separate name for them, since in contrast to homomorphisms they are not closed under composition and have the uniqueness property CATA UNIQ.

There may exists other F-algebras (L', in') for which equation $(\heartsuit)$ has several solutions for f, or no one at all. In particular, only if in' is injective (i.e., "there is no confusion" and in' has a post-inverse) there is at least one solution for f, and only if in' is surjective (i.e., "there is no junk" and in' has a pre-inverse) there is at most one solution for f. One can prove that any two initial objects are isomorphic, so that one might speak of *the* initial F-algebra.

Notice that equation $(\heartsuit)$ is 'definition by pattern matching', as in functional languages. Equation $(\diamondsuit)$ on the other hand uses explicitly a destructor out to decompose an argument into its constituents.

Example: some catamorphisms For any F and initial F-algebra (L, in) the inverse out of in is $out = (\![\mathsf{F}|\ \ in\mathsf{F}]\!)$. In Section 4 we have the laws available by which we can (and will) derive this equation *by calculation.* For any ϕ we have

$$in\mathsf{F}^n;\ \ldots;\ in\mathsf{FF};\ in\mathsf{F};\ in;\ (\![\phi]\!) \quad = \quad (\![\phi]\!)\mathsf{F}^n\mathsf{F};\ \phi\mathsf{F}^n;\ \ldots;\ \phi\mathsf{FF};\ \phi\mathsf{F};\ \phi\,.$$

This shows clearly that "a catamorphism systematically replaces the constructor of the initial algebra". Now let $\mathsf{F} = \mathbf{1} + \mathsf{I}$ and let $(N, zero \triangledown suc)$ be a (the) initial F-algebra. Then

$$zero;\ suc;\ suc;\ \ldots;\ suc;\ (\![\mathsf{F}|\ \ a \triangledown f]\!) \quad = \quad a;\ f;\ f;\ \ldots;\ f\,.$$

The inverse of $zero \triangledown suc$ is $out_N = (\![\mathsf{F}|\ \ \mathsf{id}_\mathbf{1} + (zero \triangledown suc)]\!) : N \to \mathbf{1} + N$ We have that $zero;\ out_N = \grave{\imath};\ zero \triangledown suc;\ out_N = \grave{\imath}$ and $suc;\ out_N = \acute{\imath};\ zero \triangledown suc;\ out_N = \acute{\imath}$. So, $zero;\ suc^{n+1};\ out = zero;\ suc^n;\ \acute{\imath}$.

Finality and anamorphisms By dualising the previous discussion we have the following definition of finality in $\mathsf{F}\text{-co-Alg}(K)$. Suppose (L, out) is final in $\mathsf{F}\text{-co-Alg}(K)$. Then, for any A and $\phi : A \to A\mathsf{F}$ in K there exists precisely one $f : A \to L$ in K satisfying

$(\spadesuit)\quad \phi;\ f\mathsf{F} \quad = \quad f;\ out \qquad$ or, equivalently,

$(\clubsuit)\quad f \quad = \quad \phi;\ f\mathsf{F};\ in$

where in is the inverse of out, which can be shown to exist. In words, the element in L yielded by f is built from constituents that can equivalently be obtained from f's argument by first applying ϕ and then subjecting these results to $f\mathsf{F}$. The unique solution for f is denoted $[\![\mathsf{F}|\ \phi]\!]$ or simply $[\![\phi]\!]$ if F is understood; it is called an F-*anamorphism* (from $\alpha\nu\alpha$, meaning 'upward'). This is captured by:

$$f = [\![\mathsf{F}|\ \ \phi]\!] \quad \equiv \quad f : \phi \overset{\mathsf{F}}{\succ} out \qquad \text{ANA UNIQ}$$

Notice that a catamorphism "consumes" arguments from the initial algebra, whereas an anamorphism "produces" results in the final algebra. Anamorphisms are, currently, not as widely recognised as catamorphisms, though we expect them to play an important rôle in programming.

Example: some anamorphisms Let $\mathsf{F} = \mathbf{1} + (N \times \mathsf{I})$, the functor for cons-lists over N. Suppose (M, out) is the final F-co-algebra. The inverse in of out can be written $nil \triangledown cons$ with $nil : \mathbf{1} \longrightarrow M$ and $cons : N \times M \longrightarrow M$, namely define $nil = \grave{\iota}; in$ and $cons = \acute{\iota}; in$. (Warning: in $K = \mathbf{Set}$ we have that M contains both finite and infinite lists, whereas the initial F-algebra contains only finite lists. So in $\mathbf{Set}$ $(M, nil \triangledown cons)$ is not initial. In Section 3 we shall prove that there exists categories in which for the final F-co-algebra (M, out) it is true that (M, in) is an initial F-algebra as well.) Let $out_N : N \longrightarrow \mathbf{1} + N$ be as in the previous example. Now define the anamorphism

$$preds \quad = \quad [\![\mathsf{F}|\ \ out_N ; \Delta_N\mathsf{F}]\!] \qquad : N \longrightarrow M\,.$$

Then $preds$ yields the list of predecessors of its argument. This may also be seen by rewriting the definition of $preds$ according to (♣), obtaining the recursive equation

$$\begin{array}{rcl} preds & = & out_N ; \mathsf{id} + (\mathsf{id} \vartriangle \mathsf{id}) ; preds\mathsf{F} ; nil \triangledown cons \\ & = & out_N ; nil \triangledown (\mathsf{id} \vartriangle preds ; cons)\,. \end{array}$$

Writing n for $zero ; suc^n$ we have $\quad n ; suc ; preds \quad = \quad n ; \acute{\iota} ; nil \triangledown (\mathsf{id} \vartriangle preds ; cons) \quad =$ $n \vartriangle (n ; preds) ; cons$.
Let furthermore $f : N \longrightarrow N$ be arbitrary. Define the anamorphism

$$f^\omega \quad = \quad [\![\mathsf{id}_N \vartriangle f ; \acute{\iota}]\!] \qquad : N \longrightarrow M\,.$$

This f^ω is known as "iterate f", as may be seen by rewriting it according to (♣):

$$\begin{array}{rcl} f^\omega & = & \mathsf{id} \vartriangle f ; \acute{\iota} ; f^\omega\mathsf{F} ; nil \triangledown cons \\ & = & \mathsf{id} \vartriangle f ; \acute{\iota} ; \mathsf{id} + (\mathsf{id} \times f^\omega) ; nil \triangledown cons \\ & = & \mathsf{id} \vartriangle (f ; f^\omega) ; cons\,. \end{array}$$

Hence, writing $f \vartriangle g ; cons$ as $f{:}g$ (associating to the right), we have $f^\omega = \mathsf{id}{:}(f ; f^\omega) = \ldots = \mathsf{id}{:}f{:}(f ; f){:}\cdots{:}(f^n ; f^\omega) = f^0{:}f^1{:}f^2{:}\cdots{:}f^n{:}\cdots$.

We can now define the list of natural numbers by

$$\begin{array}{rcll} from & = & suc^\omega & : N \longrightarrow M \\ & = & [\![\mathsf{id} \vartriangle suc ; \acute{\iota}]\!] & : N \longrightarrow M \\ nats & = & zero ; from & : \mathbf{1} \longrightarrow M\,. \end{array}$$

Both *iterate* f and *from* produce an infinite list; a possibly finite, possibly infinite list is produced by a **while** construct:

$$f \textbf{ while } p \quad = \quad [\![p?; (\mathsf{id} \vartriangle f) + nil]\!] \qquad : N \to M$$

where p is a predicate on N. Thus f **while** p contains all repeated applications of f as long as predicate p holds of the elements.

Recursion In the previous discussion we have seen that equations ($\Diamond$) and ($\clubsuit$) have a unique solution. In general, one may wish to define morphisms by recursion equations that are more general than either ($\Diamond$) or ($\clubsuit$), say by a *fixed point equation* like

$$(\star) \quad f \quad = \quad f\mathsf{F}$$

where F is a unary morphism combinator (not necessarily a functor), i.e., $f\mathsf{F}$ is a term possibly containing f. It is impossible to guarantee unique solutions in general; for example there are many solutions for f in the equation $f = f$. Yet, as is well-known, under reasonable assumptions on F and the category, fixed point equations do have have a "least" solution, called the *least fixed point* of F and denoted $\mu\mathsf{F}$. Moreover the least fixed point is the solution computable by a straightforward operational rewrite interpretation of "definition" $(\star)$.

Since we intend to allow arbitrary fixed point equations, it seems that there is no point in paying attention to anamorphisms and catamorphisms. However, the advantage of ana- and catamorphisms over arbitrary least fixed points is that the former are easier to calculate with, since they are characterised fully equationally whereas the characterisation of the latter involves besides equations also a partial order on the morphisms, see Section 3 and 4.

Roughly speaking (details in the next section), the existence of a least fixed point can be guaranteed if the morphisms between any two objects form a so-called pointed cpo. This order may be induced by a partial order on the elements of objects: if each object is a pointed cpo, then the set of continuous functions between two objects form a pointed cpo too, and we can take these to be the morphisms. If we furthermore require homomorphisms to be strict, we have the so-called "continuous algebras", or rather, the category 'continuous F-algebras over K', cf. Goguen et al. [18] and Reynolds [36]. However, contrary to what is suggested in the literature, we will show in Section 3 that the operation of each algebra has to be strict as well, if an initial one is proven to exist. (There is no contradiction with the *proof* given by Reynolds since any junc operation $\phi = \psi \triangledown \chi$ is strict even if ψ and χ are not.)

> *Thus, the "universe of discourse" K shall be the category $K = CPO$ of cpo's with continuous, not necessarily strict, functions as morphisms (so that fixed point equations have a unique least solution), but the main theorem, (47), only asserts initiality in F-Alg$(K_\perp)$ (and finality in F-co-Alg(K)) where $K_\perp = CPO_\perp =$ the subcategory having only strict functions as morphisms.*

Notice also that the category F-Alg$(K_\perp)$ is a subcategory of the category of "continuous F-algebras over K".

Polymorphism and natural transformations Let F and G be functors. If, for all objects A, $\phi_A : A\mathsf{F} \to A\mathsf{G}$ is a morphism, we say ϕ is a *polymorphic morphism*, or briefly a *poly-morphism*, and write $\phi : \alpha\mathsf{F} \to \alpha\mathsf{G}$ (which abbreviates $\phi : \forall(A :: A\mathsf{F} \to A\mathsf{G})$). Actually, ϕ is a family of morphisms rather than a single one. An example is $\mathsf{id} : \alpha \to \alpha$. It is often

the case, and in this paper always, that a polymorphic morphism satisfies a "polymorphic equation". To be precise, $\phi : \alpha\mathsf{F} \to \alpha\mathsf{G}$ is called a *natural transformation* if

$$(23) \quad \forall(f{:}A{\to}B :: \; f\mathsf{F}; \phi_B = \phi_A; f\mathsf{G}) \quad \text{which we denote by} \quad \phi : \mathsf{F} \stackrel{.}{\to} \mathsf{G} \qquad \text{NTRF DEF}$$

Wadler [42] and de Bruin [12] prove that any polymorphic (lambda-definable?) function is a natural transformation, but the precise context in which their theorem holds is not clear to us. Inspired by their theorem we shall prove that the poly-morphisms that we encounter are natural transformations indeed; these proofs turn out to be quite simple.

As for homomorphisms, equation (23) is computationally relevant since it gives two operationally different ways of computing the same function. Moreover, natural transformations are calculationally attractive since we have the following nice, easily verified, laws.

$$(24) \quad \phi\mathsf{H} : \mathsf{FH} \stackrel{.}{\to} \mathsf{GH} \;\Leftarrow\; \phi : \mathsf{F} \stackrel{.}{\to} \mathsf{G} \qquad \text{NTRF DISTR}$$

$$(25) \quad \phi; \psi : \mathsf{F} \stackrel{.}{\to} \mathsf{H} \;\Leftarrow\; \phi : \mathsf{F} \stackrel{.}{\to} \mathsf{G} \;\wedge\; \psi : \mathsf{G} \stackrel{.}{\to} \mathsf{H} \qquad \text{NTRF COMPOSE}$$

$$(26) \quad \phi : \mathsf{HF} \stackrel{.}{\to} \mathsf{HG} \;\Leftarrow\; \phi : \mathsf{F} \stackrel{.}{\to} \mathsf{G} \qquad \text{NTRF POLY}$$

and moreover

$$(27) \quad \mathsf{id} : \mathsf{F} \stackrel{.}{\to} \mathsf{F} \qquad \text{NTRF ID}$$

$$(28) \quad f : A \stackrel{.}{\to} B \;\equiv\; f : A \to B \qquad \text{NTRF TRIV}$$

$$(29) \quad \phi_1 \dagger \phi_2 : \mathsf{F}_1 \dagger \mathsf{F}_2 \stackrel{.}{\to} \mathsf{G}_1 \dagger \mathsf{G}_2 \;\Leftarrow\; \phi_i : \mathsf{F}_i \stackrel{.}{\to} \mathsf{G}_i \;\; (i = 1, 2) \qquad \text{NTRF BI-DISTR}$$

$$(30) \quad f : g; \phi \stackrel{\mathsf{G}}{\to} g; \psi \;\Leftarrow\; g : \mathsf{F} \stackrel{.}{\to} \mathsf{G} \;\wedge\; f : \phi \stackrel{\mathsf{F}}{\to} \psi \qquad \text{NTRF TO HOMO}$$

$$(31) \quad \Delta : \mathsf{I} \stackrel{.}{\to} \mathbb{I} \qquad \nabla : \mathbb{I} \stackrel{.}{\to} \mathsf{I} \qquad \text{NTRF SPLIT JUNC}$$

$$(32) \quad \begin{array}{ll} \grave{\pi} : \mathsf{I} \times \mathsf{G} \stackrel{.}{\to} \mathsf{I} & \grave{\iota} : \mathsf{I} \stackrel{.}{\to} \mathsf{I} + \mathsf{G} \\ \acute{\pi} : \mathsf{F} \times \mathsf{I} \stackrel{.}{\to} \mathsf{I} & \acute{\iota} : \mathsf{I} \stackrel{.}{\to} \mathsf{F} + \mathsf{I} \end{array} \qquad \text{NTRF PROJ INJ}$$

$$(33) \quad \phi : \mathsf{F} \stackrel{.}{\to} \mathsf{I} \;\equiv\; \forall(f :: \; f : \phi \stackrel{\mathsf{F}}{\to} \phi) \qquad \text{NTRF FROM HOMO}$$

$$(34) \quad \phi : \mathsf{I} \stackrel{.}{\to} \mathsf{F} \;\equiv\; \forall(f :: \; f : \phi \stackrel{\mathsf{F}}{\succ} \phi) \qquad \text{NTRF FROM COHOMO}$$

For doing and verifying calculations the subscript to a poly-morphism is hardly needed, so we shall often omit them. However, the subscripts are very helpful when checking whether a term is meaningful (well-typed), and grasping its meaning; so in definitions and laws we do provide them.

Example The typing $\mathsf{id} : \alpha \to \alpha$ suggests $\mathsf{id} : \mathsf{I} \stackrel{.}{\to} \mathsf{I}$. This is true indeed, since $f; \mathsf{id} = \mathsf{id}; f$ for all f. Less trivially, recall the functions *head*, *tail*, *inits*, *tails* and *join/* explained informally in the beginning. (A completely formal treatment will be given in Section 6.) These are all polymorphic, and the equations that we have given for them assert that they are natural transformations indeed:

$$\begin{array}{lll} \mathit{head} & : & \mathsf{L} \stackrel{.}{\to} \mathsf{I} \\ \mathit{tail} & : & \mathsf{L} \stackrel{.}{\to} \mathsf{L} \\ \mathit{inits} & : & \mathsf{L} \stackrel{.}{\to} \mathsf{LL} \\ \mathit{tails} & : & \mathsf{L} \stackrel{.}{\to} \mathsf{LL} \\ \mathit{join/} & : & \mathsf{LL} \stackrel{.}{\to} \mathsf{L}\,. \end{array}$$

Using these facts we can now prove $f\mathsf{L}; \mathit{segs} = \mathit{segs}; f\mathsf{LL}$ for all f, i.e., $\mathit{segs} : \mathsf{L} \stackrel{.}{\to} \mathsf{LL}$, even slightly simpler than we did to illustrate homomorphisms:

$\quad inits; tails_{\mathsf{L}}; join/ : \mathsf{L} \dot{\rightarrow} \mathsf{LL}$

$\Leftarrow$ $\quad$ NTRF COMPOSE (25)

$\quad inits : \mathsf{L} \dot{\rightarrow} \mathsf{LL}, \quad tails_{\mathsf{L}} : \mathsf{LL} \dot{\rightarrow} \mathsf{LLL}, \quad join/ : \mathsf{LLL} \dot{\rightarrow} \mathsf{LL}$

$\Leftarrow$ $\quad$ for the middle conjunct: NTRF DISTR (24);
for the right conjunct: NTRF POLY (26);
given facts

$\quad \mathsf{true}$.

One should compare this with the proofs given earlier.

3 The Underlying Theory — The Main Theorem

In this section we define our notion of data type, and prove its existence. We also show that strictness of the operation of the algebras is a necessary condition. The properties asserted by the Main Theorem will be captured in the next section by laws for program calculation.

Fixed point theory We assume that the reader is familiar with elementary fixed point theory, see e.g., Stoy [39] or Schmidt [37]. A *pointed cpo* is a complete partial ordered set (the partial order denoted $\sqsubseteq$) with a least element denoted $\bot$ ("bottom"). The least upperbound with respect to $\sqsubseteq$ is denoted $\bigsqcup$. (Some authors call this just a cpo, and use the name pre-cpo for what we call a cpo. More specifically, by completeness we mean ω-completeness: every ascending *chain* has a least upper bound.) Functions that preserve $\bot$ are called *strict*. Kleene's Fixed Point Theorem says that for pointed cpo A and continuous function f from A to A, f has a *least fixed point*, denoted $\mu_A f$ or simply μf, given by

(35) $\mu f \quad = \quad \bigsqcup(n :: \bot . f^n)$.

(Recall that we write $x.f$ for "x subject to f" when morphism f is used as a function.) That μf is the least fixed point of f is fully captured by two implications:

(36) $\mu f = x \quad \Rightarrow \quad x = x.f$ FIXED POINT

(37) $\mu f \sqsubseteq x \quad \Leftarrow \quad x.f \sqsubseteq x \quad \{\Leftarrow \quad x.f = x\}$. LEAST

A predicate P on A is *inclusive* if $P(\bigsqcup(n :: a_n)) \Leftarrow \forall(n :: P(a_n))$ for any ascending chain $(n :: a_n)$. Predicates built from continuous functions, $=$ and $\sqsubseteq$, and have the form of a universally quantified conjunction of disjunctions, are inclusive. For inclusive predicates P we have the so-called Least Fixed Point Induction (sometimes called Computational Induction, or Scott Induction):

(38) $P(\mu f) \quad \Leftarrow \quad P(\bot) \quad \wedge \quad \forall(x :: P(x) \Rightarrow P(x.f))$ LFP IND

(39) $P(\mu f, \mu g) \quad \Leftarrow \quad P(\bot, \bot) \quad \wedge \quad \forall(x, y :: P(x, y) \Rightarrow P(x.f, y.g))$. LFP IND2

This allows us to prove properties about μf without using the definition of μ explicitly. Bird [5] gives many examples of its use.

Here is a derived law, called "fusion". We shall see in the sequel that this kind of law plays an important role in program calculations (and derivations of additional laws). For continuous functions f, g, h:

(40) $(\mu f).g = \mu h \quad \Leftarrow \quad f ; g = g ; h$ and g strict LFP FUSION

We prove the equality $(\mu f).g = \mu h$ by Fixed Point Induction, taking $P(x, y) \equiv (x.g = y)$. The base case $P(\bot, \bot)$ is simple since g is strict. For the induction step we have

$P(x.f, y.h)$

$\equiv$ unfold

$x.(f ; g) = y.h$

$\equiv$ premiss

$x.(g ; h) = y.h$

$\equiv$ induction hypothesis $P(x, y)$

true.

A slightly stronger version of LFP FUSION (40), and a large number of applications, have been investigated by Meyer [31]. The law has already been mentioned by Stoy [39]. Gunter et al. [19] call a fixed point operator *uniform* if law LFP FUSION (40) holds for it; they show that the least fixed point operator μ is the *unique* uniform fixed point operator.

The category CPO **and** $CPO_\perp$ The category CPO has as objects pointed cpo's, and as morphisms the continuous functions (with function composition as morphism composition, and the identity functions as identity morphisms). The category $CPO_\perp$ is the sub-category that has the same objects and has as morphisms only the strict morphisms of CPO. The set of continuous functions from a pointed cpo A to another one B is a pointed cpo itself; the order is given by $f \sqsubseteq_{A\to B} g \;\equiv\; \forall(x :: x.f \sqsubseteq_B x.g)$ and the least element $\perp_{A\to B}$ is $x \mapsto \perp_B$. Composition $;$ is monotonic and continuous in both arguments. The final object in CPO and $CPO_\perp$ is $\mathbf{1} = \{\perp\}$; it is initial in $CPO_\perp$ as well.

The product combinators are defined as follows.

$$
\begin{array}{lcl}
A \times B & = & \{(x,y) \mid x \in A \;\wedge y \in B\} \\
(x,y) \sqsubseteq (x',y') & \equiv & x \sqsubseteq x' \;\wedge\; y \sqsubseteq y' \\
f \vartriangle g & = & x \mapsto (x.f, x.g) \\
\grave{\pi} & = & (x,y) \mapsto x \\
\acute{\pi} & = & (x,y) \mapsto y
\end{array}
$$

and, following (3), $f \times g = (\grave{\pi}; f) \vartriangle (\acute{\pi}; g)$. Hence $\perp_{A\times B} = (\perp_A, \perp_B)$ and $\grave{\pi}, \acute{\pi}$ are strict, and $\times, \vartriangle$ preserve strictness, are monotonic and continuous in each of their arguments, and they satisfy the axioms of a product: (1) and (3), both for $K = CPO$ and for $K = CPO_\perp$.

It is known that CPO has no co-products, but $CPO_\perp$ does (by defining $A+B$ as the *coalesced sum* of A and B, thus identifying $\perp_A$ and $\perp_B$ as $\perp_{A+B}$). See e.g. Manes & Arbib [28]. Nevertheless we define $+$ as the *separated sum*, in which $\perp_A$ and $\perp_B$ are kept separate and a new bottom element is added. This definition corresponds closely to fully lazy functional languages (as explained below), and has as consequence that data type definitions yield carriers with infinite elements in cases where the coalesced sum would not. An extensive discussion of this phenomenom is given by Lehmann & Smyth [22]. Actually, they propose to provide both the separated and the coalesced sum since one might sometimes wish that a data type has no infinite elements. However, we consider the presence of two 'sums' (and also two 'products') too complicated. So we define

$$
\begin{array}{lcl}
A + B & = & \{0\}\times A \;\cup\; \{1\}\times B \;\cup\; \{\perp\} \\
x \sqsubseteq y & = & x = \perp \;\vee\; ((x)_1 = (y)_1 \;\wedge\; (x)_2 \sqsubseteq (y)_2) \\
f \triangledown g & = & \perp \mapsto \perp \;\cup\; (0,a) \mapsto a.f \;\cup\; (1,b) \mapsto b.g \\
\grave{\imath} & = & a \mapsto (0,a) \\
\acute{\imath} & = & b \mapsto (1,b)
\end{array}
$$

and, following (4), $f + g = (f; \grave{\imath}) \triangledown (g; \acute{\imath})$. Hence, $f \triangledown g$ and $f + g$ are strict for all f, g, and are monotonic and continuous in each of their arguments. Also, $+$ is a bi-functor on both CPO and $CPO_\perp$. Now recall the equation that characterises the categorical co-product:

$$
f = g \triangledown h \quad\equiv\quad \grave{\imath}; f = g \;\wedge\; \acute{\imath}; f = h
$$

The equation does not hold on CPO since CPO has no co-products; indeed, the left hand side determines f completely for given g, h but the right hand side does not determine the outcome of $\perp.f$. Surprisingly, the equation does hold on $CPO_\perp$, i.e., when f, g, h are taken to be strict morphisms, and gives us the nice calculation properties that we use so frequently. Yet $A + B$ is not a co-product in $CPO_\perp$; this is because $\grave{\iota}$ and $\acute{\iota}$ as defined above are not strict.

In an operational interpretation the difference between the separated and coalesced sum is explained as follows. In case of the separated sum, value $(0, \perp_A)$ as input for a program f means that the tag 0 is fully determined and f can use this information to produce already some part of its output, e.g., the tag of its result, or the complete result if it is independent of the actual tagged value. In case of the coalesced sum, however, $(0, \perp_A)$ is identified with $(1, \perp_B)$ into $\perp_{A+B}$, and for the program f there is no information at all, not even the information that the tag of the input is 0. Clearly both sums are implementable. It is the task of the language designer to choose the sum(s) that suits his purpose, e.g., having nice calculational properties or allowing infinite elements.

The polynomial functors are monotonic and continuous on both CPO and $CPO_\perp$.

Alternative interpretations for $\times$ and $+$ The coalesced sum $\oplus$ is a functor on $CPO_\perp$ but not on CPO since it does not distribute always over composition (though the weaker comparison $(f; g) \oplus (h; j) \sqsupseteq (f \oplus h); (g \oplus j)$ holds). In order to allow for alternative interpretations of $+$ and $\times$ (in particular coalesced sum, smashed product) in the main theorem, we shall take care to use the functors only on $CPO_\perp$.

O-categories and $K_\perp$ Our main theorem below is formulated in terms of the particular category CPO. In fact, nowhere we use the fact that the category K is CPO, except in the proof of Part 1 of the Theorem where we refer to the literature. All that is needed of K is captured by the notion of $O\perp$-category, which we explain here. Most of the notions have been introduced by Wand [44]. Smyth & Plotkin [38] have simplified and clarified Wand's ideas. We have profited much from the very precise and elementary (but unfortunately not widely accessible) treatment by Bos & Hemerik [11]. The recent tutorial by Pierce [35] gives a readable account of Smyth & Plotkin's work.

Let K be a category. K is an *O-category* (Ordered, or Order enriched) if for each A, B in K the set of morphisms $A \to B$ is a cpo, and the composition of K is continuous in each of its arguments. If in addition the least element $\perp_{A \to B}$ of the cpo of morphisms $A \to B$ is a post-zero of composition ($f; \perp = \perp$), then we say that K is an *O$\perp$-category*. For example, CPO is an O$\perp$-category.

Let K be an O$\perp$-category. We call a morphism $f : A \to B$ *strict* if for all C, $\perp_{C \to A}; f = \perp_{C \to B}$. We let $K_\perp$ be the sub-category of K with the same objects as K and as morphisms only the strict morphisms of K. For $K = CPO$ we have: morphism $f : A \to B$ is strict in K if and only if f as a continuous function from cpo A to cpo B is strict; so $K_\perp = CPO_\perp$.

The category '*continuous* F*-algebras over* K' is the subcategory of F-Alg(K) (with possibly non-strict operations) in which the (homo)morphisms are the *strict* F-homomorphisms in K. Notice that F-Alg$(K_\perp)$ is a subcategory of 'continuous F-algebras over K'.

Throughout the sequel we use F, G, H as postfix operations on morphisms, so that $f(\text{FG}) = (f\text{F})\text{G} = f\text{FG}$. A mapping (possibly functor) F on the morphisms of O-category K is called

locally continuous if for all A, B the restriction of F to the set of morphisms $A \rightarrow B$ is continuous. By the fixed point theory explained above, morphism μF exists if F is a locally continuous mapping on $O\bot$-category K. Notice that in an $O\bot$-category the fixed point laws and fixed point fusion hold on the level of morphisms and mappings on morphisms. A morphism μF is not necessarily strict.

(41) μF strict $\Leftarrow$ F preserves strictness MU STRICT

(42) Fact *Let K be an $O\bot$-category, and F be a mapping that is locally continuous on K and preserves strictness (so fF is in $K_\bot$ if f is; do not confuse this with strictness of F). Then the least fixed point of F in K equals the least fixed point in $K_\bot$: $\mu_K \mathrm{F} = \mu_{K_\bot} \mathrm{F}$. So we can just write μF.*

This fact is easily proved using the defining equations (36) and (37) of the least fixed point operator.

(43) Fact *Functors A and I are locally continuous, and FG, F × G and F + G are locally continuous if F, G and × and + are. So all polynomial functors on $CPO_\bot$ are locally continuous.*

Hylomorphisms For morphisms f and g we define the unary combinator $(f \hookrightarrow g)$ by $h(f \hookrightarrow g) = f;\, h;\, g$. This combinator is called an *envelope*, cf. MacKeag & Welsh [23], and is continuous. Further, define combinator $f \overset{\mathsf{F}}{\hookrightarrow} g = \mathsf{F}(f \hookrightarrow g)$, so that $h(f \overset{\mathsf{F}}{\hookrightarrow} g) = f;\, h\mathsf{F};\, g$. The least fixed point of an envelope will play an important role in the sequel, and hence deserves a name of its own. A *hylomorphism* is a morphism that can be written $\mu(\phi \overset{\mathsf{F}}{\hookrightarrow} \psi)$ for some morphisms ϕ, ψ and functor F. (The prefix hylo- comes from the Greek ὑλη meaning "matter", after the Aristotelian philosophy that form and matter(= data) are one. (In conjuction with -morphism the vowel η changes into o.)) Here are some useful laws for hylomorphisms.

(44) $\mu(f \overset{\mathsf{F}}{\hookrightarrow} g)$ strict $\Leftarrow$ f, g strict and F preserves strictness HYLO STRICT

(45) $f;\, \mu(\phi \overset{\mathsf{F}}{\hookrightarrow} \psi);\, g = \mu(\phi' \overset{\mathsf{F}}{\hookrightarrow} \psi')$ $\Leftarrow$ $f : \phi' \overset{\mathsf{F}}{\succ\!\!-} \phi \;\wedge\; g : \psi \overset{\mathsf{F}}{\rightarrow} \psi' \;\wedge\; g$ strict HYLO FUSION

(46) $\mu(f \overset{\mathsf{F}}{\hookrightarrow} \grave{g});\, \mu(\acute{g} \overset{\mathsf{F}}{\hookrightarrow} h) = \mu(f \overset{\mathsf{F}}{\hookrightarrow} h)$ $\Leftarrow$ $\grave{g};\, \acute{g} = \mathsf{id}$ HYLO COMPOSE

For the first we argue

$\mu(f \overset{\mathsf{F}}{\hookrightarrow} g)$ strict

$\Leftarrow$ MU STRICT (41)

$(f \overset{\mathsf{F}}{\hookrightarrow} g)$ preserves strictness

$\equiv$ unfold

$\bot.(f;\, x\mathsf{F};\, g) = \bot$ for all strict x

$\equiv$ given strictness properties

true.

We prove the second by LFP IND2 (39), taking $P(x,y) \equiv f; x; g = y$. The base case $P(\perp,\perp)$ is immediate by strictness of g. For the induction step we argue

$$\begin{array}{ll} & f;\, x(\phi \overset{\mathsf{F}}{\circ\!\!\rightarrow} \psi);\, g = y(\phi' \overset{\mathsf{F}}{\circ\!\!\rightarrow} \psi') \\ \equiv & \quad \text{unfold} \\ & f;\, \phi;\, x\mathsf{F};\, \psi;\, g = \phi';\, y\mathsf{F};\, \psi' \\ \equiv & \quad \text{premiss} \\ & \phi';\, f\mathsf{F};\, x\mathsf{F};\, g\mathsf{F};\, \psi' = \phi';\, y\mathsf{F};\, \psi' \\ \equiv & \quad \text{functor property, induction hypothesis } P(x,y) \\ & \mathsf{true}\,. \end{array}$$

We prove HYLO COMPOSE (46) by LFP IND (38) taking $P(x,y,z) \equiv x; y = z$. The base case is immediate. For the induction step we argue

$$\begin{array}{ll} & x(f \overset{\mathsf{F}}{\circ\!\!\rightarrow} \grave{g});\, y(\acute{g} \overset{\mathsf{F}}{\circ\!\!\rightarrow} h) = z(\overset{\mathsf{F}}{\circ\!\!\rightarrow} h) \\ \equiv & \quad \text{unfold} \\ & f;\, x\mathsf{F};\, \grave{g};\, \acute{g};\, y\mathsf{F};\, h = f;\, z\mathsf{F};\, h \\ \equiv & \quad \text{premiss } \grave{g};\, \acute{g} = \mathsf{id}, \text{ functor axioms} \\ & f;\, (x;\, y)\mathsf{F};\, h = f;\, z\mathsf{F};\, h \\ \equiv & \quad \text{induction hypothesis } P(x,y,z) \\ & \mathsf{true}\,. \end{array}$$

We could do the last prove without least fixed point induction, using HYLO FUSION (45) instead [with $f, \phi, \psi, g := \mathsf{id}, f, \grave{g}, \mu(\acute{g} \overset{\mathsf{F}}{\circ\!\!\rightarrow} h)$], but for the fact that then strictness of h is required.

(47) Theorem (Main Theorem — simple version) *Let $K = CPO$, and let* F *be a locally continuous functor on* $K_\perp$. *Then*

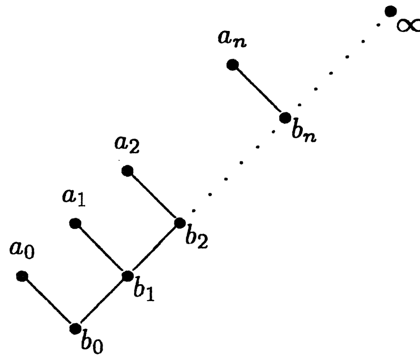
1. *there exists an object L in K and strict morphisms $in : L\mathsf{F} \rightarrow L$ and $out : L \rightarrow L\mathsf{F}$ which are each others inverses, and even $\mathsf{id}_L = \mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} in)$.*
2. *L is unique up to (a unique) isomorphism, and, given L, in and out are unique too.*
3. *(L, in) is initial in F-Alg$(K_\perp)$, and for any (A, ϕ) in F-Alg$(K_\perp)$ the unique strict homomorphism $(\!|\phi|\!) : in \overset{\mathsf{F}}{\rightarrow} \phi$ is the least solution for h of $h : in \overset{\mathsf{F}}{\rightarrow} \phi$ (in K as well as $K_\perp$).*
4. *(L, out) is final in F-co-Alg(K), and for any (A, ϕ) in F-co-Alg(K) the unique co-homomorphism $[\![\phi]\!] : \phi \overset{\mathsf{F}}{\succ\!\!-} out$ is the least solution for h of $h : \phi \overset{\mathsf{F}}{\succ\!\!-} out$ (in K, and for strict ϕ also in $K_\perp$).*

Proof of Part 1 and 2 Reynolds [36] proves entire Part 1 and 2 for the particular category $K = CPO$; a very readable and precise account of this proof is given by Schmidt [37, Chapter 11]. The crux of the proof is Scott's inverse limit construction. Informally, L is approximated

as much as you wish by repeated unfolding: $L \simeq L\mathsf{F} \simeq L\mathsf{FF} \simeq \ldots$, e.g., for $\mathsf{F} = \mathbf{1} + \mathsf{I}$ one obtains $L = N \simeq \mathbf{1} + (\mathbf{1} + (\mathbf{1} + \ldots))$. In the limit one has to adjoin a least upperbound for each ascending chain in order to obtain a *complete* partial order. In examples this informal procedure may be used to guess a solution for $L \simeq L\mathsf{F}$.

Actually, $(L, (out, in))$ is initial in the category of fixed points of F_{PR} in K_{PR}, where K_{PR} is the category of *PR*rojection pairs, or retractions, of K. A retraction from A to B is a pair $(f : A \to B, g : B \to A)$ satisfying $f; g = \mathsf{id}_A$ and $g; f \sqsubseteq \mathsf{id}_B$. It is easy to see that both components of a retraction are strict. In the inverse limit construction only strict functions play a rôle; F is applied only to strict functions, and the construction is carried out entirely in $CPO_\perp$. (Formally, $K_{PR} = (K_\perp)_{PR}$.)

Example: naturals plus infinity Let $\mathsf{F} = \mathbf{1} + \mathsf{I}$. A cpo N that solves $N \simeq N\mathsf{F} = \mathbf{1} + N$ is obtained informally by $N = \mathbf{1} + (\mathbf{1} + (\mathbf{1} + \ldots))$. This partial order is pictured here:



Note the upperbound ∞ that we have adjoined to make the partial order complete. The bijective morphism $in : \mathbf{1} + N \to N$ is given by

$$
\begin{array}{lcl}
\perp_{\mathbf{1}+N} & \mapsto & b_0 \\
(0, \perp_{\mathbf{1}}) & \mapsto & a_0 \\
(1, a_n) & \mapsto & a_{n+1} \\
(1, b_n) & \mapsto & b_{n+1} \\
(1, \infty) & \mapsto & \infty .
\end{array}
$$

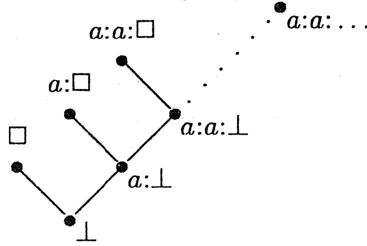
This also determines the inverse out.

Define $zero = \grave{\iota}; in : \mathbf{1} \to N$ and $suc = \acute{\iota}; in : N \to N$. Then $a_0 = \perp_{\mathbf{1}}.zero$ and $a_{n+1} = a_n.suc = \perp_{\mathbf{1}}.(zero; suc^n; suc)$, so the a_n may be interpreted as the natural numbers. Moreover there are "partial numbers" $b_0 = \perp_N$, and for all n, $b_{n+1} = b_n.suc = \perp_N.suc^{n+1}$. Finally we have $\infty = \bigsqcup b_n = \bigsqcup(\perp.suc^n)$ with $\infty.suc = \infty$; it is the infinite number.

We have $\mathsf{id}_N = \mu(out \overset{\mathsf{F}}{\hookrightarrow} in)$, i.e., id_N is the least fixed point of $f = out; \mathsf{id}_{\mathbf{1}} + f; in$; by induction on n it is easily proved that $a_n.f = a_n$ and $b_n.f = b_n$, and by continuity of f we have then that $\infty.f = (\bigsqcup b_n).f = \bigsqcup(b_n.f) = \infty$. By the theorem, any solution of $N \simeq \mathbf{1} + N$ with $\mathsf{id}_N = \mu(out \overset{\mathsf{F}}{\hookrightarrow} in)$ is isomorphic to the above one.

Let $h : N \to A$ be an F-homomorphism, say $h : zero \triangledown suc \xrightarrow{F} \phi \triangledown \psi$. Then h is uniquely determined (as it should be!), in particular the image of the infinite number (inaccessible via *suc* and *zero*) is determined by strictness and continuity of h: $\infty.h = (\bigsqcup b_n).h = \{\text{continuity}\}$ $\bigsqcup(b_n.h) = \bigsqcup(b_0.\ suc^n;\ h) = \bigsqcup(b_0.\ h;\ \psi^n) = \{\text{strictness}\}\ \bigsqcup(\bot_A.\psi^n)$.

Example: finite and infinite lists Let $F = \mathbf{1} + (A \times I)$. A (the) cpo L that solves $L \simeq LF = \mathbf{1} + (A \times L)$ is schematically pictured as follows:



Here a stands for a generic element of A. Formally, L consists of all $a_0{:}a_1{:}\ldots{:}a_{n-1}{:}\Box$ and all $a_0{:}a_1{:}\ldots{:}a_{n-1}{:}\bot$, completed with all $a_0{:}a_1{:}\ldots{:}a_{n-1}{:}\ldots$. The bijective morphism $in : \mathbf{1} + (A \times L) \to L$ is given by

$$\begin{array}{lll} \bot_{LF} & \mapsto & \bot \\ (0, \bot_1) & \mapsto & \Box \\ (1, (a, x)) & \mapsto & a{:}x \end{array}$$

for any x in L. Defining $nil = \grave{\iota};\ in : \mathbf{1} \to L$ and $cons = \acute{\iota};\ in : A \times L \to L$, we have that $\bot_1.nil = \Box$ and $(a, x).cons = a{:}x$. So, $\Box$ may be called the empty list, the $a_0{:}a_1{:}\ldots{:}\Box$ are the (totally determined) finite lists, the $a_0{:}a_1{:}\ldots{:}\bot$ are the partial lists, and the $a_0{:}a_1{:}\ldots$ are the (totally determined) infinite lists.

Example: infinite lists Let $F = A \times I$. A/the cpo that solves $L \simeq LF$ is the set $\{(a_0, a_1, \ldots) \mid \text{all } a_n \in A\}$, ordened componentwise, the least element $\bot_L$ being $(\bot_A, \bot_A, \ldots)$.

Define the "lifting" functor $\bot$ by $A\bot = A \cup \{\bot\}$ (the newly added $\bot$ being the least element), and $f\bot = (\bot \mapsto \bot_A \ \cup \ a \mapsto a.f)$. This lifting is a continuous functor indeed, on both *CPO* and $CPO_\bot$. Let $G = F\bot$, the composition of $F = A \times I$ with $\bot$. A (the) cpo that solves $M \simeq MG$ contains not only all $(a_0, a_1, \ldots)$ but also all finite approximations $(a_0, a_1, \ldots, \bot)$, where $\bot \sqsubseteq (a_i, a_{i+1}, \ldots, \bot) \sqsubseteq (a_i, a_{i+1}, \ldots)$.

Remark In practice one may wish to turn the right hand side in $M \simeq A \times M$ into a 1-ary separated sum with just one summand, viz. $A \times M$; compare the above F and G. Also, one might wish to consider $A{+}B{+}C$ as a 3-ary separated sum, rather than a cascaded 2-ary one, so that the amount of newly added $\bot$-elements is minimised. These wishes can be achieved easily if both the lifting functor and the *disjoint union* are provided, see Schmidt [37]. For the sake of simplicity in the formulas we refrain from doing so in this paper.

Proof of Part 3 One should note that the following part of the proof is entirely categorical, we do not use any property of K but for the fact that it is an $O\bot$-category. Let (A, ϕ) be any F-algebra over $K_\bot$. Put $\mathsf{G} = (out \overset{\mathsf{F}}{\hookrightarrow} \phi)$. We shall show for arbitrary morphism $h : L \to A$ in K

(a) $\mu\mathsf{G} \sqsubseteq h \quad \Leftarrow \quad in; h \sqsupseteq h\mathsf{F}; \phi \quad \{\Leftarrow h : in \overset{\mathsf{F}}{\to} \phi\}$

(b) $\mu\mathsf{G} = h \quad \Leftarrow \quad h : in \overset{\mathsf{F}}{\to} \phi \quad \wedge \quad h$ is strict

(c) $\mu\mathsf{G} : in \overset{\mathsf{F}}{\to} \phi \quad \wedge \quad \mu\mathsf{G}$ is strict.

Together these are equivalent to the assertion that $\mu\mathsf{G}$ is the unique strict solution for h of $h : in \overset{\mathsf{F}}{\to} \phi$ and the least solution for h of $h : in \overset{\mathsf{F}}{\to} \phi$ (without strictness requirement). Hence $(\!|\phi|\!) = \mu(out \overset{\mathsf{F}}{\hookrightarrow} \phi)$ (for any strict $\phi : A\mathsf{F} \to A$).

The proof of Part 3(a) is simple: by the premiss and $in^{-1} = out$, h satisfies the equation $h\mathsf{G} \sqsubseteq h$, so it is at least the least fixed point by LEAST (37).

For Part 3(b) we argue

$$
\begin{aligned}
& h = \mu\mathsf{G} \\
\equiv \quad & \text{recall from Part 1 that } \mathsf{id}_L = \mu(out \overset{\mathsf{F}}{\hookrightarrow} in) \\
& \mu(out \overset{\mathsf{F}}{\hookrightarrow} in); h = \mu\mathsf{G} \\
\equiv \quad & \text{HYLO FUSION (45)} \\
& h : in \overset{\mathsf{F}}{\to} \phi \quad \wedge \quad h \text{ strict} \\
\equiv \quad & \text{premiss} \\
& \mathsf{true}.
\end{aligned}
$$

Notice that an attempt to prove the first line immediately by, say, least fixed point induction, taking $P(x) \equiv h{=}x$, fails already in the base case $P(\bot)$. This motivates to bring in the μ-term in the left hand side. (Close inspection of the proof steps reveals that F is applied only to strict morphisms.)

Finally Part 3(b). By HYLO STRICT (44) we have that $\mu\mathsf{G}$ is strict; and by FIXED POINT (36) and $out = in^{-1}$ we have immediately $\mu\mathsf{G} : in \overset{\mathsf{F}}{\to} \phi$.

Proof of Part 4 This is dual to Part 3. Specifically, put $\mathsf{G} = (\phi \overset{\mathsf{F}}{\hookrightarrow} in)$ and show

(a) $\mu\mathsf{G} \sqsubseteq h \quad \Leftarrow \quad h : \phi \overset{\mathsf{F}}{\succ\!\!-} out$

(b) $\mu\mathsf{G} = h \quad \Leftarrow \quad h : \phi \overset{\mathsf{F}}{\succ\!\!-} out$

(c) $\mu\mathsf{G} : \phi \overset{\mathsf{F}}{\succ\!\!-} out$.

Together these are equivalent to the assertion that $\mu\mathsf{G}$ is the least, and unique, solution for h of $h : \phi \overset{\mathsf{F}}{\succ\!\!-} out$. Hence the required $[\![\phi]\!]$ equals $\mu\mathsf{G}$.

For the proof of Part 4(b), recall from Part 1 that $\mathsf{id}_L = \mu(out \overset{\mathsf{F}}{\hookrightarrow} in)$; so, prove $\mu\mathsf{G} = h; \mu(out \overset{\mathsf{F}}{\hookrightarrow} in)$ by HYLO FUSION (45). In contrast to Part 3(b) no strictness of h is needed here. We omit the details, since all is similar to Part 3.

Remark It seems that Part 4 is not entirely the dual of Part 3, since strictness occurs in one and not in the other. However, the dual of $\bot; f = \bot$ (f is strict) is $f; \bot = \bot$ and this is true for all f (by definition of $O\bot$-category). Clearly, if we restrict ourselves entirely to $K_\bot$ this asymmetry disappears and only then would 'dualising' be its own inverse.

Remark The inverse limit construction has been developed to solve domain equations like $D \cong D \to D$ (involving the function space combinator $\to$). So we may now incorporate the function space combinator into our framework, on equal footing with the product and sum combinators, and develop nice and relevant calculation laws for $\to$. This is left for future investigations.

The Main Theorem above refers to category *CPO* explicitly. We can abstract from this choice as follows.

(48) Theorem (Main Theorem — general version) *Let K be a localized $\mathrm{O}\bot$-category with K_{PR} being an ω-category, and let F be a locally continuous functor on $K_\bot$. Then the same statements 1 through 4 hold true as for the simple version.*

Proof The proof of Parts 3 and 4 is literally the same as for the simple version. For Parts 1 and 2 we refer to Smyth & Plotkin [38] and Bos & Hemerik [11]; they prove all of it (and define the notion 'localized') except for the equality $\mathsf{id}_L = \mu(out \circ\!\!\rightarrow in)$. This equality is derivable from their by-products. [Details to be supplied ...] □

Strictness is necessary

The equality $(\![\phi]\!) = \mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi)$, for strict ϕ, suggests either to *define* $(\![\phi]\!) = \mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi)$ for non-strict ϕ as well, or to restrict attention entirely to $K_\bot$. The latter option is chosen by Paterson [34]; it has the drawback that in actual programs one can no longer substitute equals for equals (since this requires lazy evaluation, which gives non-strict functions). Therefore we choose the former option. Unfortunately it destroys the equational characterisation of $(\![\ ]\!)$, as shown below. So we have to guard several laws by an explicit strictness requirement; in some cases a weaker requirement suffices. (If we refrain from extending $(\![\ ]\!)$ to non-strict ϕ, then we have to guard each law in which $(\![\phi]\!)$ occurs by the requirement that ϕ be strict!)

There is no simple modification to the equational characterisation of $(\![\phi]\!)$ that makes it valid for non-strict ϕ as well. In particular we shall prove the following discrepancies, for some $K, \mathsf{F}, (L, in, out)$ and some ϕ, h that meet the conditions of the Theorem. (Combinator \$ is 'strictify'; we assume that it satisfies at least $\phi\$ = \phi$ for strict ϕ, and that $\phi\$$ is strict for any ϕ.)

(a) $\mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi) = h \quad \not\equiv \quad h : in \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi$

(b) $\mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi) = h \quad \not\equiv \quad h : in \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi \ \wedge\ h$ strict

(c) $\mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi) = h \quad \not\equiv \quad h : in \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi\$$

(d) $\mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi) = h \quad \not\equiv \quad h : in \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi\$ \ \wedge\ h$ strict

We do have for all ϕ and h

(e) $\mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi\$) = h \quad \equiv \quad h : in \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi\$ \ \wedge\ h$ strict

but in view of the assumed properties of \$ this is equivalent to saying that (L, in) is initial in F-Alg$(K_\perp)$.

For (a) and (c) observe that in general a fixed point (h in the rhs) need not be the least fixed point (h in the lhs). Specifically, let $K = CPO$ and $\mathsf{F} = \mathsf{I}$ (hence $L = \{\perp\}$, $in = \perp_{L \to L} = out$), let A satisfy $\perp_A \neq a \in A$, and take $\phi = \mathsf{id}_A$ (which is strict). Then both $h = \lambda(x :: \perp_A)$ and $h = \lambda(x :: a)$ satisfy the rhs, but since they differ they can not both satisfy the equality of the lhs.

For (b) and (d) observe that the lhs does not imply 'ϕ strict' (an easy consequence of the rhs). Specifically, let $K = CPO$, let A be such that $\perp_A \neq a \in A$, and take $\mathsf{F} = A$ and $\phi = \lambda(x :: a) : A \to A$. Then $h = \mu(out \overset{\mathsf{F}}{\hookrightarrow} \phi) = \phi$ which is not strict.

(49) Corollary *For K, F, L, in as in the Theorem, it is* not *true that (L, in) is initial in the category 'continuous* F*-algebras over K'.*

This is just claim (b) above. It does not contradict the proof given by Reynolds [36] since he treats the (possibly infinite and possibly singleton) collection ϕ_s (for $s \in S$) of operations of the algebra, as a single junc combination $\nabla_{s \in S} \phi_s$. We have already seen that $\phi \triangledown \psi$ is strict, even if ϕ and ψ are not.

4 Data Types and some General Laws

Throughout the remainder of the paper we let K be any category that meets the requirements of the Main Theorem (the simple or general version); $K = CPO$ is an example. We say that a functor is a *data type functor* if it meets the requirements of the Main Theorem; the polynomial functors on $CPO_\perp$ are data type functors.

(50) Definition *Let* F *be a data type functor. Then the* F-data type, *denoted* $[\![\mathsf{F}]\!]$, *is the* (L, in, out) *asserted to exist by the Main Theorem.*

Rather than let the programmer use the construction of $[\![\mathsf{F}]\!]$ given in the Main Theorem, we list here the the most important laws for the data type $[\![\mathsf{F}]\!]$. So, throughout the section we have

$$(L, in, out) \quad = \quad [\![\mathsf{F}]\!] .$$

The basic laws are immediate from the Main Theorem; the derived laws require proof. All the proofs are merely equational reasoning; in particular least fixed point induction is not used explicitly anymore. This is quite remarkable, since many morphisms are actually least fixed points (see law CATA LFP (55) and ANA LFP (59)), and least fixed point induction is used so frequently in the literature, especially denotational semantics. We are mainly interested in equational laws, so we refrain from (attempting to be complete and) formulating laws involving $\sqsubseteq$.

4.1 Basic Laws: summary of previous section

Laws for cata- and anamorphisms The following laws just summarize (part of) the statement of the Main Theorem.

(51)	$in : L\mathsf{F} \to L$ (is strict)	$out : L \to L\mathsf{F}$ (is strict)	IN OUT TYPE
(52)	$in ; out = \mathsf{id}_{L\mathsf{F}}$	$out ; in = \mathsf{id}_L$	IN-OUT INVERSE

Let furthermore $(A, \phi : A\mathsf{F} \to A)$ be arbitrary. Then

(53)	$(\!|\phi|\!) : L \to A \quad \Leftarrow \quad \phi : A\mathsf{F} \to A$	CATA TYPE
(54)	$\phi \text{ strict} \ \wedge \ f = (\!|\phi|\!) \quad \equiv \quad f : in \xrightarrow{\mathsf{F}} \phi \ \wedge \ f \text{ strict}$	CATA UNIQ
(55)	$(\!|\phi|\!) = \mu(out \overset{\mathsf{F}}{\circ\!\!\to} \phi)$	CATA LFP
(56)	$(\!|\phi|\!) \text{ strict} \quad \Leftarrow \quad \phi \text{ strict}$	STRICTNESS CATA

Dually, for arbitrary $(A, \phi : A \to A\mathsf{F})$,

(57)	$[\!(\phi)\!] : A \to L \quad \Leftarrow \quad \phi : A \to A\mathsf{F}$	ANA TYPE
(58)	$f = [\!(\phi)\!] \quad \equiv \quad f : \phi \overset{\mathsf{F}}{\succ\!\!-} out$	ANA UNIQ
(59)	$[\!(\phi)\!] = \mu(\phi \overset{\mathsf{F}}{\circ\!\!\to} in)$	ANA LFP
(60)	$[\!(\phi)\!] \text{ strict} \quad \Leftarrow \quad \phi \text{ strict}$	STRICTNESS ANA

Laws for Product and Sum For completeness we list the postulates for the sum and product combinators too, in particular the strictness requirements.

(61)	$f\,\text{strict} \;\wedge\; f = g \triangledown h \;\equiv\; \grave{\iota}; f = g \;\wedge\; \acute{\iota}; f = h \;\wedge\; g, h\,\text{strict}$	JUNC UNIQ
(62)	$f = g \vartriangle h \;\equiv\; g = f; \grave{\pi} \;\wedge\; h = f; \acute{\pi}$	SPLIT UNIQ
(63)	$f + g \;=\; (f; \grave{\iota}) \triangledown (g; \acute{\iota})$	SUM DEF
(64)	$f \times g \;=\; (\grave{\pi}; f) \vartriangle (\acute{\pi}; g)$	PRODUCT DEF
(65)	$f \triangledown g$ strict	STRICTNESS JUNC
(66)	$f \vartriangle g$ strict $\Leftarrow$ f, g strict	STRICTNESS SPLIT
(67)	$\grave{\pi}$ and $\acute{\pi}$ are strict	STRICTNESS PROJS

Laws for Mu See FIXED POINT (36), LEAST (37), LFP IND (38), MU STRICT (41) of the previous section. But notice that all is lifted from the level of 'elements' to the level of 'morphisms' (which are supposed to form a pointed cpo indeed). Also HYLO STRICT (44), HYLO FUSION (45) and HYLO COMPOSE (46) are important to remember; these are already formulated on the right level.

4.2 Derived Laws

From the equivalences CATA UNIQ (54) and ANA UNIQ (58) one immediately obtains some laws by choosing the variables in such a way that the left hand side, or right hand side, is valid. For example, by substituting $h := (\!|\phi|\!)$ in CATA UNIQ (54) we get CATA HOMO (68) below, and by substituting $h, \phi := \mathsf{id}, in$ and using also HOMO ID (8) we get CATA ID (69):

(68)	$(\!|\phi|\!) \;:\; in \xrightarrow{\mathsf{F}} \phi$	CATA HOMO
(69)	$(\!|in|\!) \;=\; \mathsf{id}_L$	CATA ID

The dual laws hold of anamorphisms.

(70)	$[\!(\phi)\!] \;:\; \phi \stackrel{\mathsf{F}}{\succ\!\!-} out$	ANA COHOMO
(71)	$[\!(out)\!] \;=\; \mathsf{id}_L$	ANA ID

Law CATA HOMO (68) is sometimes called Computation (or Evaluation) Rule since a straightforward implementation will use this very equation from left to right as rewrite rule in the evaluation of a program.

Unique extension and Fusion Immediately from the uniqueness property of cata- and anamorphisms, CATA UNIQ (54) and ANA UNIQ (58), we obtain a useful weakened version: the Unique Extension Property.

(72)	$f = g \;\Leftarrow\; \phi\,\text{strict} \;\wedge\; \text{both } f \text{ and } g \;:\; in \xrightarrow{\mathsf{F}} \phi$	CATA UEP
(73)	$f = g \;\Leftarrow\; \text{both } f \text{ and } g \;:\; \phi \stackrel{\mathsf{F}}{\succ\!\!-} out$	ANA UEP

Meertens [30] shows nicely that the use of law CATA UEP (72) is more compact than an old-fashioned proof by induction: the schematic set-up of an inductive proof is done here once and for all, and built in into the law CATA UEP (72), so that a calculation need not be interrupted by the standard rituals of declaring the base case, declaring the induction step, and declaring that, by induction, the proof is complete. The proof obligations of both the base case and the induction step are captured by the premiss. This may be clearer in Section 5 when we specialise the law to CATA UEP SPECD (113) for a specialised data type functor.

Another useful property is the following fusion law.

(74) $\quad (\!|\phi|\!); f = (\!|\psi|\!) \quad \Leftarrow \quad f : \phi \xrightarrow{\mathsf{F}} \psi \;\wedge\; f \text{ strict}$ $\quad$ CATA FUSION

(75) $\quad [\![\phi]\!] = f; [\![\psi]\!] \quad \Leftarrow \quad f : \phi \overset{\mathsf{F}}{\succ\!\!-} \psi$ $\quad$ ANA FUSION

The law is called 'Promotion' by Backhouse [2]; we reserve the name 'Promotion' for what Bird [6, 8] has called so. Notice that the 'follows from' part of law CATA UNIQ (54) is a consequence of law CATA FUSION (74); take $\phi, \psi := in, \phi$ and use CATA ID (69). Similarly for its dual.

Here is a proof of CATA FUSION (74).

$(\!|\phi|\!); f = (\!|\psi|\!)$

$\equiv$ CATA LFP (55)

$\mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi); f = \mu(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \psi)$

$\Leftarrow$ HYLO FUSION (45), taking G to be $(; f)$

$(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \phi)(; f) = (; f)(out \overset{\mathsf{F}}{\circ\!\!\rightarrow} \psi) \quad \wedge \quad (; f)$ is strict

$\equiv$ strictness of f is given;
extensionality: for all g

$out; g\mathsf{F}; \phi; f = out; (g; f)\mathsf{F}; \psi$

$\equiv$ functor property, premiss

true .

Malcolm [25] derives this law immediately from the uniqueness property, but in our setting that would give an additional strictness requirement.

Both CATA UEP (72) and CATA FUSION (74) can be unified into the following property of the initiality of (L, in): For F-homomorphisms f, g, h, j such that both $f; h$ and $g; j$ have source (L, in) and target, say, (C, χ), we have $f; h = g; j$. (We leave it to the reader to draw a commuting square diagram.) In formulas,

$$f; h = g; j \quad \Leftarrow \quad \begin{array}{lll} f : in \xrightarrow{\mathsf{F}} \phi & h : \phi \xrightarrow{\mathsf{F}} \chi & f, h \text{ strict} \\ g : in \xrightarrow{\mathsf{F}} \psi & j : \psi \xrightarrow{\mathsf{F}} \chi & g, j \text{ strict} \end{array}$$

Taking $h = j = \mathsf{id}$ (and $\phi = \psi = \chi$) yields law CATA UEP (72), and taking $j = \mathsf{id}$ (and $\psi = \chi$) yields law CATA FUSION (74).

Example: derivation of *out* Although the Main Theorem asserts the existence of *out*, the inverse of *in*, we show here how one might derive the inverse of *in*, using the calcu-

lus developed so far. First we observe that type considerations alone already suggest the definition.

$$
\begin{array}{ll}
 & out : L \longrightarrow L\mathsf{F} \\
\equiv & \quad \text{guess that } out = (\!|\phi|\!) \text{ (what else?); CATA TYPE (53)} \\
 & \phi : L\mathsf{FF} \longrightarrow L\mathsf{F} \qquad \text{and } \phi \text{ strict} \\
\Leftarrow & \quad \text{guess that } \phi = \psi\mathsf{F}\text{; functor axiom} \\
 & \psi : L\mathsf{F} \longrightarrow L \qquad \text{and } \psi \text{ strict} \\
\equiv & \quad \text{guess that } \psi = in\text{; IN OUT TYPE (51)} \\
 & \mathsf{true}\,.
\end{array}
$$

Thus, taking $out = (\!|in\mathsf{F}|\!)$ we have $out : L \longrightarrow L\mathsf{F}$. (Similarly we could obtain $in = [\![out\mathsf{F}]\!]$ if we were to derive an expression for in given out.) Now we prove that out is the pre-inverse of in; again we phrase the proof as a *derivation* of $out = (\!|in\mathsf{F}|\!)$.

$$
\begin{array}{ll}
 & out;\, in = \mathsf{id}_L \\
\equiv & \quad \text{guess that } out = (\!|\phi|\!)\text{; CATA ID (69), CATA FUSION (74)} \\
 & in : \phi \xrightarrow{\mathsf{F}} in \qquad \text{and } \phi \text{ strict} \\
\equiv & \quad \text{take } \phi = in\mathsf{F}\text{; IN OUT TYPE (51), HOMO TRIV (14)} \\
 & \mathsf{true}\,.
\end{array}
$$

Finally, we need to show that out is a post-inverse of in as well.

$$
\begin{array}{ll}
 & in;\, out \\
= & \quad \text{definition } out = (\!|in\mathsf{F}|\!)\text{, CATA HOMO (68)} \\
 & out\mathsf{F};\, in\mathsf{F} \\
= & \quad \text{functor axiom; above derived: } out \text{ is pre-inverse of } in \\
 & \mathsf{id}\,.
\end{array}
$$

Relating cata, ana and mu to each other There are several laws relating cata- and anamorphisms to each other. Notice that a condition like $f;\, g = \mathsf{id}$ means that f is a pre-inverse of g, and g is a post-inverse of f; in **Set** it follows that f is injective and g is surjective. In CATA ANA EQV (79) we use the abbreviation

$$
\phi\langle \mathsf{F} \dot{\longrightarrow} \mathsf{G}\rangle\psi \quad \equiv \quad \forall f :: f\mathsf{F};\, \psi = \phi;\, f\mathsf{G}
$$

(note the reversal of ϕ and ψ). As a special case we have $\phi\langle \mathsf{F} \dot{\longrightarrow} \mathsf{G}\rangle\phi \equiv \phi : \mathsf{F} \dot{\longrightarrow} \mathsf{G}$.

$$
\begin{array}{lrcll}
(76) & [\!(\phi)\!];\ (\![\psi]\!) & = & \mu(\phi \overset{\mathsf{F}}{\circ\!\!\rightarrow} \psi) & \text{ANA CATA HYLO} \\
(77) & (\![\phi]\!);\ [\!(\psi)\!] = \mathsf{id}_L & \Leftarrow & \phi;\ \psi = \mathsf{id}_{A\mathsf{F}} & \text{CATA ANA ID} \\
(78) & [\!(\phi)\!];\ (\![\psi]\!) \sqsubseteq \mathsf{id}_A & \Leftarrow & \phi;\ \psi \sqsubseteq \mathsf{id}_A & \text{ANA CATA ID} \\
(79) & (\![\mathsf{F}|\ \ \phi;\ in_{\mathsf{G}}]\!) = [\!(\mathsf{G}|\ \ out_{\mathsf{F}};\ \psi)\!] & \Leftarrow & \psi\langle \mathsf{F} \dot{\rightarrow} \mathsf{G}\rangle\phi & \text{CATA ANA EQV} \\
(80) & (\![\mathsf{F}|\ \ \phi;\ \psi]\!) = [\!(\mathsf{G}|\ \ out_{\mathsf{F}};\ \phi)\!];\ (\![\mathsf{G}|\ \ \psi]\!) & \Leftarrow & \phi : \mathsf{F} \dot{\rightarrow} \mathsf{G} & \text{CATA DECOMPOSE1} \\
(81) & (\![\mathsf{F}|\ \ \phi;\ \psi]\!) = (\![\mathsf{F}|\ \ \phi;\ in_{\mathsf{G}}]\!);\ (\![\mathsf{G}|\ \ \psi]\!) & \Leftarrow & \phi : \mathsf{F} \dot{\rightarrow} \mathsf{G} & \text{CATA DECOMPOSE2} \\
(82) & [\!(\mathsf{G}|\ \ \phi;\ \psi)\!] = [\!(\mathsf{F}|\ \ \phi)\!];\ (\![\mathsf{F}|\ \ \psi;\ in_{\mathsf{G}}]\!) & \Leftarrow & \psi : \mathsf{F} \dot{\rightarrow} \mathsf{G} & \text{ANA DECOMPOSE1} \\
(83) & [\!(\mathsf{G}|\ \ \phi;\ \psi)\!] = [\!(\mathsf{F}|\ \ \phi)\!];\ [\!(\mathsf{G}|\ \ out_{\mathsf{F}};\ \psi)\!] & \Leftarrow & \psi : \mathsf{F} \dot{\rightarrow} \mathsf{G} & \text{ANA DECOMPOSE2}
\end{array}
$$

The last few laws deal with $(L_{\mathsf{F}}, in_{\mathsf{F}}, out_{\mathsf{F}}) = [\![\mathsf{F}]\!]$ as before, but also with $(L_{\mathsf{G}}, in_{\mathsf{G}}, out_{\mathsf{G}}) = [\![\mathsf{G}]\!]$. Recall that the $\phi : \mathsf{F} \dot{\rightarrow} \mathsf{G}$ implies that ϕ is polymorphic $: \alpha\mathsf{F} \rightarrow \alpha\mathsf{G}$. In CATA ANA EQV (79) the occurrences within the catamorphism (of type $L_{\mathsf{F}} \rightarrow L_{\mathsf{G}}$) are typed: $\phi : L_{\mathsf{G}}\mathsf{F} \rightarrow L_{\mathsf{G}}\mathsf{G}$ and $in_{\mathsf{G}} : L_{\mathsf{G}}\mathsf{G} \rightarrow L_{\mathsf{G}}$. Those within the anamorphism (of type $L_{\mathsf{F}} \rightarrow L_{\mathsf{G}}$) are typed: $out_{\mathsf{F}} : L_{\mathsf{F}} \rightarrow L_{\mathsf{F}}\mathsf{F}$ and $\psi : L_{\mathsf{F}}\mathsf{F} \rightarrow L_{\mathsf{F}}\mathsf{G}$.

Law CATA ANA ID (77) may alternatively be formulated as follows. Denote the pre-inverse of ϕ by $\phi^{`}$, if it exists, and the post-inverse by ϕ', if it exists. Then, assuming that the right hand sides are well defined,

$$
\begin{array}{lcl}
[\!(\phi)\!] & = & (\![\phi^{`}]\!)' \\
(\![\phi]\!) & = & [\!(\phi')\!]^{`}
\end{array}
$$

We have given ANA CATA ID (78) only to stress that there is no equality in the conclusion. Before delving into the proofs we give some examples.

Example: reverse (To illustrate CATA ANA EQV (79) and CATA DECOMPOSE2 (81).) Let $\mathsf{F} = A + \mathbb{I}$, so that $(B, in, out) = [\![\mathsf{F}]\!]$ is the data type of nonempty binary trees with A-elements at the tips. Define

$$
\begin{array}{lcll}
reverse & = & (\![\mathsf{F}|\ \ \mathsf{id}_A + swap_B;\ in]\!) & : B \rightarrow B \\
swap & = & \acute{\pi} \vartriangle \grave{\pi} & : B \times B \rightarrow B \times B\,.
\end{array}
$$

Actually, *reverse* and *swap* are poly-morphisms. We can write *reverse* as an anamorphism by law CATA ANA EQV (79), since $\mathsf{id}_A + swap : A + \mathbb{I} \dot{\rightarrow} A + \mathbb{I}$ (as is easily verified by the NTRF laws):

$$
reverse \;=\; [\!(\mathsf{F}|\ \ out;\ \mathsf{id}_A + swap)\!] \quad : B \rightarrow B\,.
$$

As an illustration of CATA DECOMPOSE2 (81) we have that $reverse;\ reverse = \mathsf{id}$:

$$
\begin{array}{cl}
 & reverse;\ reverse \\
= & \quad \text{unfold} \\
 & (\![\mathsf{id} + swap;\ in]\!);\ (\![\mathsf{id} + swap;\ in]\!) \\
= & \quad \text{CATA DECOMPOSE2 (81), and } \mathsf{id} + swap : A + \mathbb{I} \dot{\rightarrow} A + \mathbb{I} \\
 & (\![\mathsf{id} + swap;\ \mathsf{id} + swap;\ in]\!) \\
= & \quad \text{easy: } swap;\ swap = \mathsf{id}
\end{array}
$$

$$\llparenthesis in \rrparenthesis$$
$=$ Cata Id (69)
$$\mathsf{id}\,.$$

Example: paramorphisms (To illustrate Hylo Compose (46).) Let $(L, in, out) = [\![\mathsf{F}]\!]$ be given, as well as $\phi : (L \times A)\mathsf{F} \to A$. Consider the equation in $f : L \to A$

(a) $in;\, f \;=\; (\mathsf{id}_L \vartriangle f)\mathsf{F};\, \phi\,.$

Notice that the occurrence of ϕ in the equation not only recieves the results of f on the constituents, but also the constituents themselves. An example of this scheme is the usual definition of the factorial:

$$zero \triangledown suc;\, factorial \;=\; (\mathsf{id} \vartriangle factorial)(1 + \mathsf{I});\, zero \triangledown (suc \times \mathsf{id};\, times)$$

We define combinator $(\mathsf{id}\vartriangle)$ by $f(\mathsf{id}\vartriangle) = \mathsf{id} \vartriangle f$, and $\{\phi\}$ by

$$\{\phi\} \;=\; \mu((\mathsf{id}\vartriangle)(out \overset{\mathsf{F}}{\hookrightarrow} \phi)) \qquad : L \to A$$

so that $\{\phi\}$ is a solution for f in (a). (Our $\{\phi\}$ serves the same purpose as Meertens' [30] paramorphisms.) Now define $\mathsf{G} = (L \times \mathsf{I})\mathsf{F}$, so that $\phi : A\mathsf{G} \to A$, and put $(M, In, Out) = [\![\mathsf{G}]\!]$. Further define

$$preds \;=\; [\![\mathsf{G}|\;\; out;\, \Delta\mathsf{F}]\!] \qquad : L \to M\,.$$

Then $\{\phi\} = preds;\, \llparenthesis \mathsf{G}|\;\; \phi \rrparenthesis$, as shown by this calculation:

$$preds;\, \llparenthesis \mathsf{G}|\;\; \phi \rrparenthesis$$
$=$ unfold
$$[\![\mathsf{G}|\;\; out;\, \Delta\mathsf{F}]\!];\, \llparenthesis \mathsf{G}|\;\; \phi \rrparenthesis$$
$=$ Ana LFP (59), Cata LFP (55)
$$\mu(out;\, \Delta\mathsf{F} \overset{\mathsf{G}}{\hookrightarrow} In);\, \mu(Out \overset{\mathsf{G}}{\hookrightarrow} \phi)$$
$=$ Hylo Compose (46), In-Out Inverse (52)
$$\mu(out;\, \Delta\mathsf{F} \overset{\mathsf{G}}{\hookrightarrow} \phi)$$
$=$ see below
$$\mu((\mathsf{id}\vartriangle)(out \overset{\mathsf{F}}{\hookrightarrow} \phi))$$
$=$ fold
$$\{\phi\}\,.$$

The one but last step is proved by showing that the combinators are extensionally equal:

$$f(out;\, \Delta\mathsf{F} \overset{\mathsf{G}}{\hookrightarrow} \phi)$$
$=$
$$out;\, \Delta\mathsf{F};\, f\mathsf{G};\, \phi$$
$=$
$$out;\, \Delta\mathsf{F};\, \mathsf{idF} \times f\mathsf{F};\, \phi$$
$=$
$$out;\, (\mathsf{id} \vartriangle f)\mathsf{F};\, \phi$$
$=$
$$f(\mathsf{id}\vartriangle)(out \overset{\mathsf{F}}{\hookrightarrow} \phi)\,.$$

Example (To illustrate CATA ANA ID (77).) Let $\mathsf{F} = \mathbf{1} + N \times \mathsf{I}$ and $(L, in, out) = [\![\mathsf{F}]\!] =$ the data type of cons-lists over N. Let $suc, pred : N \rightarrow N$ such that $suc; pred = \mathsf{id}_N$. Then, by CATA ANA ID (77) we have

$$(\![\mathsf{id}_1 + suc \times \mathsf{id}_L ; in]\!) ; [\!(out ; \mathsf{id}_1 + pred \times \mathsf{id}_L)\!] \quad = \quad \mathsf{id}_L .$$

We shall later see that the catamorphism is *suc*-map and the anamorphism is *pred*-map. (If $(N, zero \triangledown suc, out_N) = [\![\mathbf{1} + \mathsf{I}]\!]$, we can define *pred* by $pred = out_N ; zero \triangledown \mathsf{id}$.)

4.3 The proofs of the laws

For ANA CATA HYLO (76) we calculate

$$\begin{array}{ll} & [\!(\phi)\!] ; (\![\psi]\!) \\ = & \quad \text{ANA LFP (59), CATA LFP (55)} \\ & \mu(\phi \overset{\mathsf{F}}{\multimap} in) ; \mu(out \overset{\mathsf{F}}{\multimap} \psi) \\ = & \quad \text{IN-OUT INVERSE (52), HYLO COMPOSE (46)} \\ & \mu(\phi \overset{\mathsf{F}}{\multimap} \psi) . \end{array}$$

For CATA ANA ID (77) we calculate

$$\begin{array}{ll} & (\![\phi]\!) ; [\!(\psi)\!] \\ = & \quad \text{CATA LFP (55), ANA LFP (59)} \\ & \mu(out \overset{\mathsf{F}}{\multimap} \phi) ; \mu(\psi \overset{\mathsf{F}}{\multimap} in) \\ = & \quad \text{HYLO COMPOSE (46), premiss: } \phi ; \psi = \mathsf{id} \\ & \mu(out \overset{\mathsf{F}}{\multimap} in) \\ = & \quad \text{CATA LFP (55), CATA ID (69)} \\ & \mathsf{id} . \end{array}$$

For ANA CATA ID (78) we calculate

$$\begin{array}{ll} & [\!(\phi)\!] ; (\![\psi]\!) \\ = & \quad \text{ANA LFP (59), CATA LFP (55)} \\ & \mu(\phi \overset{\mathsf{F}}{\multimap} in) ; \mu(out \overset{\mathsf{F}}{\multimap} \psi) \\ = & \quad \text{IN-OUT INVERSE (52), HYLO COMPOSE (46)} \\ & \mu(\phi \overset{\mathsf{F}}{\multimap} \psi) \\ \sqsubseteq & \quad \text{AUX LAW1 (84) below, premiss: } \phi ; \psi \sqsubseteq \mathsf{id} \\ & \mathsf{id} . \end{array}$$

The auxiliary law used above reads

$$\text{(84)} \qquad \mu(f \overset{\mathsf{F}}{\multimap} g) \sqsubseteq \mathsf{id} \quad \Leftarrow \quad f ; g \sqsubseteq \mathsf{id} \qquad \text{AUX LAW1}$$

The proof is simple:

$\mu(f \overset{\mathsf{F}}{\hookrightarrow} g) \sqsubseteq \mathsf{id}$

$\Leftarrow$ LEAST (37)

$\mathsf{id}(f \overset{\mathsf{F}}{\hookrightarrow} g) \sqsubseteq \mathsf{id}$

$\equiv$ unfold, premiss

true.

Also for CATA ANA EQV (79) and CATA DECOMPOSE1 (80) we have an auxiliary law:

$$(85) \qquad f \overset{\mathsf{F}}{\hookrightarrow} (\psi; g) \;=\; (f; \phi) \overset{\mathsf{G}}{\hookrightarrow} g \quad \Leftarrow \quad \phi\langle \mathsf{F} \dot{\to} \mathsf{G}\rangle \psi \qquad \text{AUX LAW2}$$

where, again, the premiss abbreviates $\forall f :: f\mathsf{F}; \psi = \phi; f\mathsf{G}$. The proof is simple and omitted. Now for CATA ANA EQV (79) we calculate

$(\!|\mathsf{F}| \;\; \phi; in_{\mathsf{G}}|\!)$

$=$ CATA LFP (55)

$\mu(out_{\mathsf{F}} \overset{\mathsf{F}}{\hookrightarrow} \phi; in_{\mathsf{G}})$

$=$ AUX LAW2 (85)

$\mu(out_{\mathsf{F}}; \psi \overset{\mathsf{G}}{\hookrightarrow} in_{\mathsf{G}})$

$=$ ANA LFP (59)

$[\![\mathsf{G}| \;\; out_{\mathsf{F}}; \psi]\!]$.

For CATA DECOMPOSE1 (80) we calculate

$(\!|\mathsf{F}| \;\; \phi; \psi|\!)$

$=$ CATA LFP (55)

$\mu(out_{\mathsf{F}} \overset{\mathsf{F}}{\hookrightarrow} \phi; \psi)$

$=$ AUX LAW2 (85), premiss

$\mu(out_{\mathsf{F}}; \phi \overset{\mathsf{G}}{\hookrightarrow} \psi)$

$=$ ANA CATA HYLO (76)

$[\![\mathsf{G}| \;\; out_{\mathsf{F}}; \phi]\!]; (\!|\mathsf{G}| \;\; \psi|\!)$.

For CATA DECOMPOSE2 (81) we calculate

$(\!|\mathsf{F}| \;\; \phi; \psi|\!)$

$=$ CATA DECOMPOSE1 (80), premiss

$[\![\mathsf{G}| \;\; out_{\mathsf{F}}; \phi]\!]; (\!|\mathsf{G}| \;\; \psi|\!)$

$=$ CATA ANA EQV (79), premiss

$(\!|\mathsf{F}| \;\; \phi; in_{\mathsf{G}}|\!); (\!|\mathsf{G}| \;\; \psi|\!)$.

The proofs of ANA DECOMPOSE1 (82) and ANA DECOMPOSE2 (83) are the duals of the proof of CATA DECOMPOSE1 (80) and CATA DECOMPOSE2 (81).

5 Map for any Data Type

Sometimes the data type of lists over A is denoted $A*$, and for $f : A \to B$ there is defined a function f-map, denoted $f* : A* \to B*$. Two laws for $f*$ are: $\mathsf{id}_A* = \mathsf{id}_{A*}$ and $(f; g)* = f*; g*$. These statements together precisely assert that $*$ is a functor. Another characteristic property of $f*$ is that it leaves the "shape" of its argument unaffected, and only affects the constituents of its argument that have type A. We shall now show that any data type comes equiped with such a map functor. Its action on morphisms can be defined both as a catamorphism and as an anamorphism — which is not true in Malcolm's [25] approach.

Let F_A be a data type functor that depends functorially on A, i.e., F_A can be written $A\dagger\mathsf{I}$ for some bi-functor $\dagger$. (For example, the functor for cons-lists over A is $\mathsf{F}_A = \mathbf{1}+(A\times\mathsf{I}) = A\dagger\mathsf{I}$ where $\dagger$ is defined by $x\dagger y = \mathbf{1}+(x\times y)$. Actually, any functor F can so be written; define $\dagger$ by $x\dagger y = y\mathsf{F}$, then $\mathsf{F} = A\dagger\mathsf{I}$.) Define a mapping L on objects, together with poly-morphisms $in : \alpha\mathsf{F} \to \alpha\mathsf{L}$ and $out : \alpha\mathsf{L} \to \alpha\mathsf{F}$, by

$$(A\mathsf{L}, in_A, out_A) = [\![A\dagger\mathsf{I}]\!] .$$

Mapping L plays the rôle of $*$ above. We wish to define L on morphisms as well, in such a way that L becomes a functor (on K). The requirement '$f\mathsf{L} : A\mathsf{L} \to B\mathsf{L}$ whenever $f : A \to B$' together with '$f\mathsf{L}$ is a catamorphism' does not leave much room for the definition of L: (at each step "there is only one thing you can do")

$f\mathsf{L} : A\mathsf{L} \to B\mathsf{L}$

$\equiv$ we wish $f\mathsf{L} = (\![\phi]\!)$ say;
typing rules

$f\mathsf{L} = (\![A\dagger\mathsf{I}|\ \phi]\!) \ \wedge\ \phi : B\mathsf{L}(A\dagger\mathsf{I}) \to B\mathsf{L}$

$\equiv$ since $in_B : B\mathsf{L}(B\dagger\mathsf{I}) \to B\mathsf{L}$ guess that $\phi = \psi; in_B$;
typing rules

$\psi : B\mathsf{L}(A\dagger\mathsf{I}) \to B\mathsf{L}(B\dagger\mathsf{I})$

$\equiv$ since $f : A \to B$ guess that $\psi = f(\mathsf{I}\dagger B\mathsf{L}) = f\dagger\mathsf{id}_{B\mathsf{L}}$;
typing rules

true.

(As shown below in law MAP DUAL (87), the wish that map be an anamorphism leads to an equivalent definition.) So we define, for $f : A \to B$,

$$f\mathsf{L} \;=\; (\![A\dagger\mathsf{I}|\ \ f\dagger\mathsf{id}_{B\mathsf{L}}; in_B]\!) \;:\; A\mathsf{L} \to B\mathsf{L} \qquad (86)\ \text{MAP DEF}$$

The L so defined is called *the map functor induced by* $\mathsf{F}_A = A\dagger\mathsf{I}$ or, more precisely, by $\dagger$. The functor properties will be proved later on. Notice that the subterm $f\dagger\mathsf{id}_{B\mathsf{L}}$ has the effect of subjecting the A-constituents of the argument to f, and leaving the argument unaffected otherwise; here the "functoriality" of the dependence on A is exploited.

Example: map for cons-lists Recall the data type $(A\mathsf{L}, nil_A \triangledown cons_A, out_A) = [\![\mathsf{F}_A]\!]$ of cons-lists over A. The data type functor is $\mathsf{F}_A = \mathbf{1}+(A\times\mathsf{I}) = A\dagger\mathsf{I}$ where $x\dagger y = \mathbf{1}+(x\times y)$; and actually nil and $cons$ are polymorphic. For any $f : A \to B$ we find

$$
\begin{array}{ll}
& \mathsf{true} \\
\equiv & \quad \text{MAP DEF (86)} \\
& f\mathsf{L} = (\!| A \dagger \mathsf{I} | \quad f \dagger \mathsf{id}_{B\mathsf{L}}; in_B |\!) \\
\Rightarrow & \quad \text{CATA UNIQ (54), noticing that } f \dagger \mathsf{id}; in \text{ is strict} \\
& in_A; f\mathsf{L} \;=\; f\mathsf{L}(A \dagger \mathsf{I}); f \dagger \mathsf{id}_{B\mathsf{L}}; in_B \\
\equiv & \quad \text{definition of } \dagger, \text{ and } nil, cons \\
& nil_A \triangledown cons_A; f\mathsf{L} \;=\; \mathsf{id}_1 + (\mathsf{id}_A \times f\mathsf{L}); \mathsf{id}_1 + (f \times \mathsf{id}_{B\mathsf{L}}); nil_B \triangledown cons_B \\
\equiv & \quad \text{combinator and functor laws} \\
& nil_A; f\mathsf{L} \;=\; nil_B \\
& cons_A; f\mathsf{L} \;=\; f \times f\mathsf{L}; cons_B
\end{array}
$$

This is the usual definition of f-map for cons-lists; compare with the example in the introduction.

Here are some useful laws; some explanation and type assumptions follow the list. In each cata/anamorphism the functor is $\mathsf{F}_A = A \dagger \mathsf{I}$. First we relate the cata- and anamorphism definition:

$$(87) \qquad (\!| A \dagger \mathsf{I} | \quad f \dagger \mathsf{id}_{B\mathsf{L}}; in_B |\!) \;=\; f\mathsf{L} \;=\; [\!(B \dagger \mathsf{I} | \quad out_A; f \dagger \mathsf{id}_{A\mathsf{L}})\!] \qquad \text{MAP DUAL}$$

Here are the two functor axioms:

$$
\begin{array}{lrcll}
(88) & \mathsf{id}_A\mathsf{L} & = & \mathsf{id}_{A\mathsf{L}} & \text{MAP ID} \\
(89) & (f; g)\mathsf{L} & = & f\mathsf{L}; g\mathsf{L} & \text{MAP DISTRIBUTION}
\end{array}
$$

and for the catamorphism definition we have:

$$
\begin{array}{lcl}
(90) & f\mathsf{L} \;:\; in_A \overset{A\dagger\mathsf{I}}{\rightarrow} f \dagger \mathsf{id}_{B\mathsf{L}}; in_B & \text{MAP HOMO} \\
(91) & in : \mathsf{I} \dagger \mathsf{L} \dot{\rightarrow} \mathsf{L} & \text{IN NTRF} \\
(92) & f\mathsf{L}; (\!|\phi|\!) \;=\; (\!| f \dagger \mathsf{id}; \phi |\!) & \text{CATA FACTORN1} \\
(93) & f : (\!|\phi|\!) \overset{\mathsf{L}}{\rightarrow} (\!|\psi|\!) \;\Leftarrow\; f : \phi \overset{\mathsf{I}\dagger\mathsf{I}}{\rightarrow} \psi \;\wedge\; f \text{ strict} & \text{PROMO OVER CATA} \\
(94) & (\!|\phi|\!) : \mathsf{L} \dot{\rightarrow}_\perp \mathsf{I} \;\Leftarrow\; \phi : \mathsf{I} \dagger \mathsf{I} \dot{\rightarrow}_\perp \mathsf{I} & \text{CATA NTRF}
\end{array}
$$

and dually for the anamorphism definition:

$$
\begin{array}{lcl}
(95) & f\mathsf{L} \;:\; out_A; f \dagger \mathsf{id}_{A\mathsf{L}} \overset{A\dagger\mathsf{I}}{\succ\!\!-} out_B & \text{MAP COHOMO} \\
(96) & out \;:\; \mathsf{L} \dot{\rightarrow} \mathsf{I} \dagger \mathsf{L} & \text{OUT NTRF} \\
(97) & [\!(\phi)\!]; f\mathsf{L} \;=\; [\!(\phi; f \dagger \mathsf{id})\!] & \text{ANA FACTORN1} \\
(98) & f : [\!(\phi)\!] \overset{\mathsf{L}}{\succ\!\!-} [\!(\psi)\!] \;\Leftarrow\; f : \phi \overset{\mathsf{I}\dagger\mathsf{I}}{\succ\!\!-} \psi & \text{PROMO OVER ANA} \\
(99) & [\!(\phi)\!] : \mathsf{I} \dot{\rightarrow} \mathsf{L} \;\Leftarrow\; \phi : \mathsf{I} \dot{\rightarrow} \mathsf{I} \dagger \mathsf{I} & \text{ANA NTRF}
\end{array}
$$

The name MAP DISTRIBUTION (89) has become standard. Law CATA FACTORN1 (92) is Verwer's [41] factorisation theorem; it says not only that $f\mathsf{L}; (\!|\phi|\!)$ is a catamorphism, but also that any catamorphism of the form $(\!| f \dagger \mathsf{id}; \phi |\!)$ can be factored in a map followed by a

catamorphism (which is always possible by choosing $f = \mathsf{id}$). We shall later state a more specialised factorisation law. Fully typed the law reads: for $f : A \to B$ and $\phi : B \dagger C \to B$,

$$f\mathsf{L} ; (\![B \dagger \mathsf{I}| \quad \phi]\!) \quad = \quad (\![A \dagger \mathsf{I}| \quad f \dagger \mathsf{id}_C ; \phi]\!) \quad : A\mathsf{L} \to C .$$

Law PROMO OVER CATA (93) captures the informal meaning expressed by Bird [6] in his 'Promotion and Accumulation Strategies': in $(\![\phi]\!) ; f = f\mathsf{L} ; (\![\psi]\!)$ morphism f is promoted from being a post-process of $(\![\phi]\!)$ into being a pre-process of $(\![\psi]\!)$. We shall later specialise this law for a particular $\dagger$ and various f, ϕ, ψ, thus obtaining MAP PROMOTION (120), REDUCE PROMOTION (121), and CATA PROMOTION (122). The ingredients of law PROMO OVER CATA (93) are typed as follows:

$$\Rightarrow \frac{f : A \to B \qquad \phi : A \dagger A \to A \qquad \psi : B \dagger B \to B}{(\![A \dagger \mathsf{I}| \quad \phi]\!) : A\mathsf{L} \to A \qquad (\![B \dagger \mathsf{I}| \quad \psi]\!) : B\mathsf{L} \to B}$$

Before delving into the proofs, we give some examples.

Example (To illustrate MAP DISTRIBUTION (89).) For arbitrary $\dagger$, $\mathsf{F}_A = A \dagger \mathsf{I}$ and $(A\mathsf{L}, in_A, out_A) = [\![\mathsf{F}_A]\!]$, we define the "shape" of elements from $A\mathsf{L}$ to be elements of $\mathbf{1}\mathsf{L}$:

$$shape_A \quad = \quad !_{A\mathsf{L}} \quad : A\mathsf{L} \to \mathbf{1}\mathsf{L} .$$

Then we can prove that "map preserves shape": for any $f : A \to B$ we have

$$\begin{array}{ll} & f\mathsf{L} ; shape_B \\ = & f\mathsf{L} ; !_{B\mathsf{L}} \\ = & (f ; !_B)\mathsf{L} \\ = & !_{A\mathsf{L}} \\ = & shape_A . \end{array}$$

Example (To illustrate CATA FACTORN1 (92).) Let $x \dagger y = \mathbf{1} + x \times y$ so that $\mathsf{F}_A = A \dagger \mathsf{I} = \mathbf{1} + A \times \mathsf{I}$ = the functor for cons-lists over A, and let L, in and out be induced by $\dagger$. Suppose further that N has been defined, as well as morphisms $zero : \mathbf{1} \to N$, $add : N \times N \to N$, and $sq : N \to N$. Consider now

$$sumsquares \quad = \quad sq\mathsf{L} ; (\![zero \triangledown add]\!) \quad : N\mathsf{L} \to N .$$

We can calculate a single catamorphism for the composition *sumsquares*; usually this is done by the Fold-Unfold technique, but we can do it simpler. By CATA FACTORN1 (92) we have

$$\begin{array}{ll} & sumsquares \\ = & sq\mathsf{L} ; (\![zero \triangledown add]\!) \\ = & (\![sq \dagger \mathsf{id} ; zero \triangledown add]\!) \\ = & (\![\mathsf{id} + sq \times \mathsf{id} ; zero \triangledown add]\!) \\ = & (\![zero \triangledown (sq \times \mathsf{id} ; add)]\!) . \end{array}$$

Example: Promotion over iterate (To illustrate PROMO OVER ANA (98).) Recall that for $f : A \to A$, "iterate f" $= f^\omega = [\![A \dagger \mathsf{I} | \quad \mathsf{id} \vartriangle f; \iota]\!] : A \to A\mathsf{L}$, where $x \dagger y = \mathbf{1} + (x \times y)$ and $\mathsf{F}_A = A \dagger \mathsf{I}$. For iterate there is a promotion law similar to PROMO OVER ANA (98):

$$(100) \qquad f : g^\omega \overset{\mathsf{L}}{\succ\!\!-} h^\omega \quad \Leftarrow \quad f : g \overset{\mathsf{I}}{\succ\!\!-} h \qquad \text{PROMO OVER ITERATE}$$

Here is the proof.

$$
\begin{array}{ll}
 & f : [\![\mathsf{id} \vartriangle g; \iota]\!] \overset{\mathsf{L}}{\succ\!\!-} [\![\mathsf{id} \vartriangle h; \iota]\!] \\
\Leftarrow & \quad \text{PROMO OVER ANA (98)} \\
 & f : (\mathsf{id} \vartriangle g; \iota) \overset{\mathsf{I}\dagger\mathsf{I}}{\succ\!\!-} (\mathsf{id} \vartriangle h; \iota) \\
\equiv & \quad \text{definition } \dagger \text{ and observation below} \\
 & f : g \overset{\mathsf{I}}{\succ\!\!-} h .
\end{array}
$$

In the last step we used the following equivalence, easily proved using only the combinator and functor laws. We also give its dual.

$$
\begin{array}{lcl}
f : (g; \iota) \overset{\mathsf{F}+\mathsf{G}}{\succ\!\!-} (h; \iota) & \equiv & f : g \overset{\mathsf{G}}{\succ\!\!-} h \\
f : (\grave{\pi}; g) \overset{\mathsf{F}\times\mathsf{G}}{\to} (\grave{\pi}; h) & \equiv & f : g \overset{\mathsf{F}}{\to} h .
\end{array}
$$

Map is a data type functor Earlier we gave the example that rose trees $(A\mathsf{R}, \mathit{tip} \triangledown \mathit{fork})$ form a $A + \mathsf{L}$-algebra. Actually, one would define them as the data type $[\![A + \mathsf{L}]\!]$. However, here we use L as a component of a data type functor. So we should prove that functor L is a data type functor whenever F_A is. This is easy: the Main Theorem requires only that L is a locally continuous functor. We know already that L is a functor, if F_A is, so it remains to show that L is locally continuous, i.e., $f\mathsf{L}$ is continuous in f. For this we argue

$$
\begin{array}{ll}
 & f\mathsf{L} \text{ is continuous in } f \\
\equiv & \quad \text{MAP DEF (86), CATA LFP (55)} \\
 & \mu(\mathit{out} \overset{\mathsf{F}}{\circ\!\!\to} f \dagger \mathsf{id}; \mathit{in}) \text{ is continuous in } f \\
\Leftarrow & \quad \text{see below} \\
 & \mathit{out} \overset{\mathsf{F}}{\circ\!\!\to} f \dagger \mathsf{id}; \mathit{in} \quad \text{is continuous in } f \\
\equiv & \quad \text{composition etc. and } \mathsf{F} \text{ are continuous} \\
 & \mathsf{true} .
\end{array}
$$

For the one but last step we argue as follows (quite standard in fact) to show that $\mu(\mathrm{F}(f))$ is continuous in f if $\mathrm{F}(f)$ is continuous in f. Let $\mathcal{F}$ be an ascending chain. Then

$$
\begin{array}{ll}
 & \bigsqcup_{f\in\mathcal{F}}(\mu(\mathrm{F}(f))) \\
= & \bigsqcup_{f\in\mathcal{F}} \bigsqcup_{n\in N} \bot.(\mathrm{F}(f))^n \\
= & \bigsqcup_{n\in N} \bigsqcup_{f\in\mathcal{F}} \bot.(\mathrm{F}(f))^n \\
= & \bigsqcup_{n\in N} \bot. \bigsqcup_{f\in\mathcal{F}}(\mathrm{F}(f))^n \\
= & \bigsqcup_{n\in N} \bot.(\bigsqcup_{f\in\mathcal{F}} \mathrm{F}(f))^n \\
= & \bigsqcup_{n\in N} \bot.(\mathrm{F}(\bigsqcup_{f\in\mathcal{F}} f))^n \\
= & \mu(\mathrm{F}(\bigsqcup_{f\in\mathcal{F}} f)) .
\end{array}
$$

This is a surprisingly simple proof in comparison with the corresponding one by Malcolm [25], the reason being that the implication "F locally continuous implies F_{PR} ω-co-continuous" is already implicit in (the proof of) the Main Theorem.

5.1 Proofs of the laws

The following fact is needed several times.

(101) $\qquad f \dagger \mathsf{id} : A \dagger \mathsf{I} \dot{\rightarrow} B \dagger \mathsf{I} \qquad$ AUX LAW3

The proof is easy:

$f \dagger \mathsf{id} : A \dagger \mathsf{I} \dot{\rightarrow} B \dagger \mathsf{I}$

$\Leftarrow$ NTRF BI-DISTR (29)

$f : A \dot{\rightarrow} B \;\wedge\; \mathsf{id} : \mathsf{I} \dot{\rightarrow} \mathsf{I}$

$\equiv$ for left conjunct: NTRF TRIV (28) and $f : A \rightarrow B$;
for right conjunct: NTRF ID (27)

true.

For MAP DUAL (87) we argue

$(\!|A \dagger \mathsf{I}| \;\; f \dagger \mathsf{id}; in_B|\!) = [\![B \dagger \mathsf{I}| \;\; out_A; f \dagger \mathsf{id}]\!]$

$\Leftarrow$ CATA ANA EQV (79)

$f \dagger \mathsf{id} : A \dagger \mathsf{I} \dot{\rightarrow} B \dagger \mathsf{I}$

$\equiv$ AUX LAW3 (101)

true.

For MAP ID (88) we calculate

$\mathsf{id}_A \mathsf{L}$

$=$ MAP DEF (86)

$(\!|\mathsf{id} \dagger \mathsf{id}; in|\!)$

$=$ functor properties: $\mathsf{id} \dagger \mathsf{id} = \mathsf{id}$; CATA ID (69)

id.

For MAP DISTRIBUTION (89) we argue

$f\mathsf{L}; g\mathsf{L}$

$=$ MAP DEF (86)

$(\!|f \dagger \mathsf{id}; in|\!); (\!|g \dagger \mathsf{id}; in|\!)$

$=$ CATA DECOMPOSE2 (81), above AUX LAW3 (101)

$(\!|f \dagger \mathsf{id}; g \dagger \mathsf{id}; in|\!)$

$=$ functor axiom, MAP DEF (86)

$(f; g)\mathsf{L}$

We remark that if one tries to prove the equation $f\mathsf{L}; g\mathsf{L} = (f; g)\mathsf{L}$ by unfolding $f\mathsf{L}$ and $(f; g)\mathsf{L}$ according to MAP DEF (86), and then applying CATA FUSION (74), one succeeds only if $g\mathsf{L}$ is strict.
For MAP HOMO (90) we argue

$$f\mathsf{L} : in_A \overset{A\dagger\mathsf{I}}{\rightarrow} f \dagger \mathsf{id}_{B\mathsf{L}}; in_B$$

$\equiv$ MAP DEF (86), CATA HOMO (68)

true.

For IN NTRF (91) we argue

$$in : \mathsf{I} \dagger \mathsf{L} \dot{\rightarrow} \mathsf{L}$$

$\equiv$ unfold: for all f

$$f(\mathsf{I} \dagger \mathsf{L}); in = in; f\mathsf{L}$$

$\equiv$ exchange lhs and rhs, use $f(\mathsf{I} \dagger \mathsf{L}) = f\mathsf{L}(A \dagger \mathsf{I}); f \dagger \mathsf{id}$

$$in; f\mathsf{L} = f\mathsf{L}(A \dagger \mathsf{I}); f \dagger \mathsf{id}; in$$

$\equiv$ MAP HOMO (90)

true.

For CATA FACTORN1 (92) we argue

$$f\mathsf{L}; (\!|\phi|\!)$$

$=$ MAP DEF (86)

$$(\!|f \dagger \mathsf{id}; in|\!); (\!|\phi|\!)$$

$=$ CATA DECOMPOSE2 (81), AUX LAW3 (101)

true.

Again we remark that a proof with MAP DEF (86) followed by CATA FUSION (74) gives the condition that ϕ be strict.
For PROMO OVER CATA (93) we argue

$$f : (\!|\phi|\!) \overset{\mathsf{L}}{\rightarrow} (\!|\psi|\!)$$

$\equiv$ unfold

$$(\!|\phi|\!); f = f\mathsf{L}; (\!|\psi|\!)$$

$\equiv$ CATA FACTORN1 (92)

$$(\!|\phi|\!); f = (\!|f \dagger \mathsf{id}; \psi|\!)$$

$\Leftarrow$ CATA FUSION (74) (recall $\mathsf{F}_A = A \dagger \mathsf{I}$), f is strict

$$f : \phi \overset{A\dagger\mathsf{I}}{\rightarrow} f \dagger \mathsf{id}; \psi$$

$\equiv$ given $f : \phi \overset{\mathsf{I}\dagger\mathsf{I}}{\rightarrow} \psi$

$$f(A \dagger \mathsf{I}); f \dagger \mathsf{id} = f(\mathsf{I} \dagger \mathsf{I})$$

$\equiv$ functor properties

true.

For CATA NTRF (94) we argue

$$(\!|\phi|\!) : \mathsf{L} \stackrel{.}{\to}_{\perp} \mathsf{I} \quad \Leftarrow \quad \phi : \mathsf{I} \dagger \mathsf{I} \stackrel{.}{\to}_{\perp} \mathsf{I}$$

$\equiv$ PROMO OVER CATA (93), NTRF FROM HOMO (33)

true .

For the remaining laws we can simply take the dual of the above proofs.

6 Monads and Reduce, Co-monads and Generate

Warning In comparison with Malcolm [25] there are no new laws here (but there are some new proofs). We simply can not resist the temptation to spend another couple of pages to the propagation of his results.

A *monad* is an algebraic structure that seems to occur often in programming (in disguised form) —and elsewhere— though most programmers do not know the concept at all. In the paper 'Comprehending Monads' (having two meanings indeed), Wadler [43] gives a thorough discussion of the concept of monad and its use in functional programming. The reader is recommended to read that article. It will then be clear that the data type $A\mathsf{L}$ of join-lists over A possesses a monad, namely

$$(\mathsf{L},\quad \mathsf{tau} : \mathsf{I} \dot{\to} \mathsf{L},\quad \mathsf{join/} : \mathsf{LL} \dot{\to} \mathsf{L})$$

where L is the map functor of join-lists, $\mathsf{tau} : \alpha \to \alpha\mathsf{L}$ is the polymorphic singleton former, and $\mathsf{join/} : \alpha\mathsf{LL} \to \alpha\mathsf{L}$ is the polymorphic flattening function that flattens a list of lists into a single list. In order that this triple $(\mathsf{L}, \mathsf{tau}, \mathsf{join/})$ is a monad indeed, the components tau and $\mathsf{join/}$ should be natural transformations, as suggested, and satisfy the following equations:

$$\begin{array}{lcll} \mathsf{tau};\mathsf{join/} & = & \mathsf{id} & : \alpha \\ \mathsf{tau\,L};\mathsf{join/} & = & \mathsf{id} & : \alpha\mathsf{L} \\ \mathsf{join/};\mathsf{join/} & = & \mathsf{join/L};\mathsf{join/} & \end{array}$$

(It is not required that $\mathsf{join/}$ is built from two components join and $/$ as happens to be the case here.) Following Malcolm [25] we show that the construction of this monad for join-lists generalises to the map functor L of any functor F_A of the form $\mathsf{F}_A = A + \mathsf{G}$. A stepping stone in this construction is the definition of $\phi/ : A\mathsf{L} \to A$ ("reduce-with-ϕ") for arbitrary $\phi : A\mathsf{G} \to A$.

Dualisation gives a co-monad for the map functor L induced by any data type functor F_A of the form $\mathsf{F}_A = A \times \mathsf{G}$; the dual of a reduce being a "generate" $\phi\backslash : A \to A\mathsf{L}$ for $\phi : A \to A\mathsf{G}$. There seems to be no law like MAP DUAL (87) here.

Specialising the functor to get a monad Let $\mathsf{F}_A = A + \mathsf{G}$ for some G. (Taking $\mathsf{G} = \mathbb{I}$ makes $[\![\mathsf{F}_A]\!]$ the data type of nonempty binary trees with values from A at the tips, i.e., "nonempty join-lists over A where the join-operation is not associative". It may be helpful to keep this particular G in mind, since the construction below closely parallels the definition of reduce for join-lists.) Let L be the map functor induced by F_A, i.e.,

$$\begin{array}{lcl} (A\mathsf{L}, in_A, out_A) & = & [\![A + \mathsf{G}]\!] \\ f\mathsf{L} & = & (\![A + \mathsf{G} \mid \ f + \mathsf{id}_{B\mathsf{LG}}; in_B]\!) \quad : \quad A\mathsf{L} \to B\mathsf{L} \end{array}$$

for $f : A \to B$. The particular form of F_A allows us to define polymorphic tau (singleton former) and join (joining operation):

(102) $\qquad \mathsf{tau}_A = \grave{\imath}; in_A \quad : A \to A\mathsf{L} \qquad$ TAU DEF

(103) $\qquad \mathsf{join}_A = \acute{\imath}; in_A \quad : A\mathsf{LG} \to A\mathsf{L} \qquad$ JOIN DEF

(104) $\qquad in = \mathsf{tau} \triangledown \mathsf{join} \quad : A + A\mathsf{LG} \to A\mathsf{L}. \qquad$ IN TAU JOIN

The particular form of $F_A = A + G$ (and the above tau and join) allows us also to specialise (indicated by "SPECD") some definitions and laws that we already had for arbitrary $F_A = A \dagger I$ (in which we can take $x \dagger y = x + yG$ to get $F_A = A + G$):

(105)	$(\!\lvert f \triangledown \phi \rvert\!) : \text{tau} \xrightarrow{A} f$	CATA ON TAU
(106)	$(\!\lvert f \triangledown \phi \rvert\!) : \text{join} \xrightarrow{G} \phi$	CATA ON JOIN
(107)	$f\text{L} = (\!\lvert f + \text{id}_{B\text{LG}} ; \text{tau}_B \triangledown \text{join}_B \rvert\!)$ $= (\!\lvert (f ; \text{tau}) \triangledown \text{join} \rvert\!)$	MAP SPECD
(108)	$f\text{L}$ is strict	MAP SPECD STRICT
(109)	$f\text{L} : \text{tau} \xrightarrow{A} f ; \text{tau}$	MAP ON TAU
(110)	$f\text{L} : \text{join} \xrightarrow{G} \text{join}$	MAP ON JOIN
(111)	$\text{tau} : \text{I} \dot{\rightarrow} \text{L}$	TAU NTRF
(112)	$\text{join} : \text{LG} \dot{\rightarrow} \text{L}$	JOIN NTRF
(113)	$f = g \Leftarrow \text{tau} ; f = \text{tau} ; g \;\wedge\; f, g : \text{join} \xrightarrow{G} \phi$	CATA UEP SPECD

Law MAP SPECD (107) follows immediately from MAP DEF (86) by unfolding F_A and $\dagger$ and simplifying by combinator and functor laws. Then, by applying CATA HOMO (68) and splitting the $\triangledown$ we get MAP ON TAU (109) and MAP ON JOIN (110). These equations state that tau and join are natural transformations, which is exactly what TAU NTRF (111) and JOIN NTRF (112) denote.

Law CATA UEP (72) specialises to CATA UEP SPECD (113) by unfolding F_A and $\dagger$ and using some combinator and functor laws. Quite often it is immediate that two composite morphisms f and g are both $\text{join} \xrightarrow{G} \phi$ promotable, since composition chains the promotability so nicely, see law HOMO COMPOSE (9). Other times it is trivial to check that $\text{tau} ; f = \text{tau} ; g$. (When appropiate, the promotability condition or the equality-on-tau condition may be dealt with in the hint of a calculation step.)

So, we have already the component $\text{tau} : \alpha \rightarrow \alpha\text{L}$ of the monad. In order to find a "flatten" of type $\alpha\text{LL} \rightarrow \alpha\text{L}$, let us try and define a reduce $\phi/ : A\text{L} \rightarrow A$ for arbitrary $\phi : A\text{G} \rightarrow A$, and then take $A, \phi := \alpha\text{L}, \text{join}$ (since $\text{join} : \alpha\text{LG} \rightarrow \alpha\text{L}$).

Defining reduce and constructing the monad Let $\phi : A\text{G} \rightarrow A$ be arbitrary. We wish to define a morphism $\phi/$ of type $A\text{L} \rightarrow A$ (so that $\phi/$ reduces one L of the type, whence the name). Like we did for maps, one can *derive* the following definition of $\phi/$ from the type requirement alone.

$$\phi/ \;=\; (\!\lvert A + \text{G} \mid \; \text{id}_A \triangledown \phi \rvert\!) \;: A\text{L} \rightarrow A \qquad (114) \quad \text{REDUCE DEF}$$

or, equivalently,

(115)	$\phi/ : \text{tau} \xrightarrow{A} \text{id}_A$	REDUCE ON TAU
(116)	$\phi/ : \text{join} \xrightarrow{G} \phi$	REDUCE ON JOIN

Here are some laws that we now can prove. In each catamorphism the functor is $F_A = A + G$ ($= A \dagger I$ where $x \dagger y = x + yG$), and L is the map functor induced by this functor.

(117)	$\phi/ \quad$ is strict	REDUCE STRICT
(118)	$f\mathsf{L}; \phi/ \;=\; (\!|f \triangledown \phi|\!)$	CATA FACTORN
(119)	$f : \phi/ \overset{\mathsf{L}}{\to} \psi/ \;\Leftarrow\; f : \phi \overset{\mathsf{G}}{\to} \psi \;\wedge\; f$ strict	PROMO OVER REDUCE
(120)	$f\mathsf{L} \;:\; \mathsf{join}/ \overset{\mathsf{L}}{\to} \mathsf{join}/$	MAP PROMOTION
(121)	$\phi/ \;:\; \mathsf{join}/ \overset{\mathsf{L}}{\to} \phi/$	REDUCE PROMOTION
(122)	$(\!|f \triangledown \phi|\!) : \mathsf{join}/ \overset{\mathsf{L}}{\to} (\!|f \triangledown \phi|\!)$	CATA PROMOTION
(123)	$\phi/ : \mathsf{L} \dot{\to} \mathsf{I} \;\Leftarrow\; \phi : \mathsf{G} \dot{\to} \mathsf{I}$	REDUCE NTRF
(124)	$\mathsf{join}/ : \mathsf{LL} \dot{\to} \mathsf{L}$	JOINRED NTRF
(125)	$\mathsf{tau}\mathsf{L}; \mathsf{join}/ \;=\; \mathsf{id} \;:\alpha\mathsf{L}$	IDL FACTORN
(126)	$(\mathsf{L},\;\; \mathsf{tau} : \mathsf{I} \dot{\to} \mathsf{L},\;\; \mathsf{join}/ : \mathsf{LL} \dot{\to} \mathsf{L})$ is a monad,	MONAD

For the generality of the form $(\!|f \triangledown \phi|\!)$ in these laws, notice that for *any* $(\!|A + \mathsf{G}|\; \psi|\!)$ with a strict $\psi : C(A+\mathsf{G}) \to C$ say, we have $\psi = f \triangledown \phi$ for some f and ϕ, namely $f = \grave{\iota}; \psi : A \to C$ and $\phi = \acute{\iota}; \psi : C\mathsf{G} \to C$ (also, $f = \mathsf{tau}; (\!|\psi|\!)$). So, CATA FACTORN not only says that any map followed by a reduce is a catamorphism, but also that any catamorphism can so be factored. This theorem was first mentioned by Malcolm [24].

6.1 Proofs of the laws

For REDUCE STRICT (117) we argue

$\phi/$ strict

$\equiv$ REDUCE DEF (114), STRICTNESS JUNC (65)

true .

For CATA FACTORN (118) we argue

$f\mathsf{L}; \phi/$

$=$ MAP DEF (86), REDUCE DEF (114)

$(\!|f + \mathsf{id}; in|\!); (\!|\mathsf{id} \triangledown \phi|\!)$

$=$ CATA DECOMPOSE2 (81)

$(\!|f + \mathsf{id}; \mathsf{id} \triangledown \phi|\!)$

$=$ sum-junc law

$(\!|f \triangledown \phi|\!)$.

For PROMO OVER REDUCE (119) we argue

$f : \phi/ \overset{\mathsf{L}}{\to} \psi/$

$\equiv$ REDUCE DEF (114)

$f : (\!|\mathsf{id} \triangledown \phi|\!) \overset{\mathsf{L}}{\to} (\!|\mathsf{id} \triangledown \psi|\!)$

$\Leftarrow$ PROMO OVER CATA (93), noting that $\mathsf{I} \dagger \mathsf{I} = \mathsf{I} + \mathsf{G}$

$f : \mathsf{id} \triangledown \phi \overset{\mathsf{I+G}}{\to} \mathsf{id} \triangledown \psi \;\;\wedge\;\; f$strict

$\Leftarrow$ HOMO SUMFCTR (11)

$$f : \phi \xrightarrow{\mathsf{G}} \psi \quad \wedge \quad f \text{ strict}\,.$$

For MAP PROMOTION (120) we argue

$f\mathsf{L} : \mathsf{join}/ \xrightarrow{\mathsf{L}} \mathsf{join}/$

$\Leftarrow$ PROMO OVER REDUCE (119)$[f, \phi, \psi := f\mathsf{L}, \mathsf{join}, \mathsf{join}]$

$f\mathsf{L} : \mathsf{join} \xrightarrow{\mathsf{G}} \mathsf{join} \quad \wedge \quad f\mathsf{L}$ strict

$\equiv$ MAP ON JOIN (110), MAP SPECD STRICT (108)

true.

For REDUCE PROMOTION (121) we argue

$\phi/ : \mathsf{join}/ \xrightarrow{\mathsf{L}} \phi/$

$\Leftarrow$ PROMO OVER REDUCE (119)$[f, \phi, \psi := \phi/, \mathsf{join}, \phi]$

$\phi/ : \mathsf{join} \xrightarrow{\mathsf{G}} \phi \quad \wedge \quad \phi/$ strict

$\equiv$ REDUCE ON JOIN (116), REDUCE STRICT (117)

true.

For CATA PROMOTION (122) we argue

$(\!|f \triangledown \phi|\!) : \mathsf{join}/ \xrightarrow{\mathsf{L}} (\!|f \triangledown \phi|\!)$

$\equiv$ CATA FACTORN (118)

$f\mathsf{L}; \phi/ : \mathsf{join}/ \xrightarrow{\mathsf{L}} f\mathsf{L}; \phi/$

$\Leftarrow$ HOMO COMPOSE (9)

$f\mathsf{L} : \mathsf{join}/ \xrightarrow{\mathsf{L}} \mathsf{join}/ \quad \wedge \quad \phi/ : \mathsf{join}/ \xrightarrow{\mathsf{L}} \phi/$

$\equiv$ MAP PROMOTION (120), REDUCE PROMOTION (121)

true.

For REDUCE NTRF (123) we argue

$\phi/ : \mathsf{L} \dot{\rightarrow}_{\perp} \mathsf{I} \quad \Leftarrow \quad \phi : \mathsf{G} \dot{\rightarrow}_{\perp} \mathsf{I}$

$\equiv$ PROMO OVER REDUCE (119)$[\psi := \phi]$, NTRF FROM HOMO (33)

true.

For JOINRED NTRF (124) we argue

$\mathsf{join}/ : \mathsf{LL} \dot{\rightarrow} \mathsf{L}$

$\equiv$ MAP PROMOTION (120), NTRF FROM HOMO (33)

true.

For IDL FACTORN (125) we argue

$\mathsf{tau}\mathsf{L}; \mathsf{join}/ = \mathsf{id}_{AL}$

$\equiv$ in rhs: CATA ID (69) and IN TAU JOIN (104)

$\mathsf{tau}\mathsf{L}; \mathsf{join}/ = (\!|\mathsf{tau} \triangledown \mathsf{join}|\!)$

$\equiv$ CATA FACTORN (118)

true.

For MONAD (126) we argue

> Apart from 'L is a functor', '$\mathsf{tau} : \mathsf{I} \dot{\to} \mathsf{L}$', and '$\mathsf{join}/ : \mathsf{LL} \dot{\to} \mathsf{L}$', which have been shown above, there are just three equations to be satisfied. These are REDUCE ON TAU (115)[$\phi := \mathsf{join}$], IDL FACTORN (125), and REDUCE PROMOTION (121)[$\phi := \mathsf{join}$].

Defining generate and constructing the co-monad Let $\mathsf{F}_A = A \times \mathsf{G}$ for some G. We have

$$\begin{array}{lcl} (A\mathsf{L}, in_A, out_A) & = & [\![A \times \mathsf{G}]\!] \\ f\mathsf{L} & = & [\![B \times \mathsf{G} |\ out_A ; f \times \mathsf{id}_{A\mathsf{LG}}]\!] \quad : A\mathsf{L} \to B\mathsf{L} \end{array}$$

for $f : A \to B$, and further we define, or have,

$$car = out_A ; \grave{\pi} \quad : A\mathsf{L} \to A \tag{127} \qquad \text{CAR DEF}$$

$$cdr = out_A ; \acute{\pi} \quad : A\mathsf{L} \to A\mathsf{LG} \tag{128} \qquad \text{CDR DEF}$$

$$out = car \vartriangle cdr \tag{129} \qquad \text{OUT CAR CDR}$$

Let $\phi : A \to A\mathsf{G}$ be arbitrary. The *generate with* ϕ is defined

$$\phi\backslash = [\![A \times \mathsf{G} |\ \ \mathsf{id}_A \vartriangle \phi]\!] \quad : A \to A\mathsf{L} \tag{130} \qquad \text{GENERATE DEF}$$

The formulation and proofs of the laws are left to the industrious reader.

Acknowledgements We acknowledge the suggestion by Ross Paterson that has lead to our notion of anamorphism, and the many suggestions by Jaap van der Woude, amongst others to stress the role of least fixed points of envelopes (hylomorphisms). We have profited a lot from the stimulating environment provided by (all current and former participants of) the weekly STOP meetings on Constructive Algorithmics at Utrecht University. In particular we wish to thank Lambert Meertens, Johan Jeuring and Jaap van der Woude for discussions and comments on this paper.

References

[1] L. Aiello, G. Attardi, and G. Prini. Towards a more declarative programming style. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*. North-Holland, 1978. Proceedings IFIP TC-2 Conference, St. Andrews, Canada, August 1977.

[2] R.C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS8810, Dept of Math and Comp Sc, University of Groningen, 1988.

[3] R.C. Backhouse. Naturality of homomorphisms. In *International Summerschool on Constructive Algorithmics*, 1989. Held on Ameland, Netherlands, September 1989.

[4] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[5] R.S. Bird. *Programs and Machines*. Wiley, New York, 1976.

[6] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984. Addendum: Ibid. 7(3):490–492, 1985.

[7] R.S. Bird. A calculus of functions for program derivation. Technical Report PRG–64, Oxford University, Computing Laboratory, Programming Research Group, December 1987.

[8] 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.

[9] 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), ISBN 3-540-51369-8, 0-387-51369-8.

[10] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[11] R. Bos and C. Hemerik. An introduction to the category-theoretic solution of recursive domain equations. Technical Report Computing Science Notes 88/15, Eindhoven University of Technology, Eindhoven, Netherlands, October 1988.

[12] P.J. de Bruin. Naturalness of polymorphism. Technical Report CS 8916, Department of Computing Science, University of Groningen, Netherlands, 1989.

[13] R.M. Burstall and P.J. Landin. Programs and their proofs: An algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 17–43. Edinburgh University Press, 1969.

[14] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11(1):1–30, 1978.

[15] E.W. Dijkstra. Invariance and nondeterminacy. Note EWD871, February 1984.

[16] H. Ehrig and B. Mahr. *Fundamentals of Equational Specification 1*. Springer Verlag, 1985.

[17] J.A. Goguen. How to prove inductive hypotheses without induction. In W. Bibel and R. Kowalski, editors, *Proc. 5th Conference on Automated Deduction*, number 87 in Lect. Notes in Comp. Sc., pages 356–373, 1980.

[18] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[19] C. Gunter, P. Mosses, and D. Scott. Semantic domains and denotational semantics. In *Lecture Notes International Summerschool on Logic, Algebra and Computation*, 1989. Marktoberdorf. Also to appear in *Handbook of Theoretical Computer Science*, North-Holland.

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

[21] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 283 in Lect. Notes in Comp. Sc., pages 140–157. Springer Verlag, 1988.

[22] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.

[23] M. MacKeag and J. Welsh. *Structured System Programming*. Prentice Hall, 1980.

[24] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lect. Notes in Comp. Sc., pages 335–347. Springer Verlag, 1989.

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

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

[27] G. Malcolm. Squiggoling in context. *The Squiggolist*, 1(3):14th – 19th, 1990.

[28] E.G Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Text and Monographs in Computer Science. Springer Verlag, 1986.

[29] L. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker and J.C. van Vliet, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[30] L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, Amsterdam, February 1990. To appear in Formal Aspects of Computing.

[31] E. Meijer. *???* PhD thesis, University of Nijmegen, Netherlands, 199?

[32] H. Partsch. Transformational program development in a particular problem domain. *Science of Computer Programming*, 7:99–241, 1986.

[33] R. Paterson. *Reasoning about Functional Programs.* PhD thesis, University of Queensland, Department of Computer Science, September 1987.

[34] R. Paterson. Operators. In *Lecture Notes International Summerschool on Constructive Algorithmics*, September 1990. Organized by CWI Amsterdam, Utrecht University, University of Nijmegen, and held at Hollum (Ameland), Netherlands.

[35] B.C. Pierce. A taste of category for computer scientist. *ACM Computing Surveys.* To appear. Also Tech Report CMU-CS-90-113, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 125213.

[36] J.C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, July 1977.

[37] D.A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon, 1986. Also: Wm. C. Brown Publishers, Dubuque, Iowa, USA, 1988.

[38] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–785, November 1982.

[39] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[40] Lecture notes International Summer School on Constructive Algorithmics, September 1989. Organized by CWI Amsterdam, Utrecht University, University of Nijmegen, and held at Hollum (Ameland), Netherlands.

[41] N. Verwer. Homomorphisms, factorisation and promotion. *The Squiggolist*, 1(3):45–54, 1990. Also tecnical report RUU-CS-90-5, Utrecht Univerity, 1990.

[42] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. FPCA '89, Imperial College, London.

[43] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, June 1990.

[44] M. Wand. Fixed point constructions in order enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.