Formal Aspects of Computing (1992) 4: 413–424
© 1992 BCS

Formal Aspects of Computing

Paramorphisms

Lambert Meertens

Department of Algorithmics and Architecture, CWI, Amsterdam, and Department of Computing Science, Utrecht University, The Netherlands

Keywords: Formal program construction; Initial data type

Abstract. "Catamorphisms" are functions on an initial data type (an inductively defined domain) whose inductive definitional pattern mimics that of the type. These functions have powerful calculation properties by which inductive reasoning can be replaced by equational reasoning. This paper introduces a generalisation of catamorphisms, dubbed "paramorphisms". Paramorphisms correspond to a larger class of inductive definition patterns; in fact, we show that any function defined on an initial type can be expressed as a paramorphism. In spite of this generality, it turns out that paramorphisms have calculation properties very similar to those of catamorphisms. In particular, we prove a Unique Extension Property and a Promotion Theorem for paramorphisms.

1. Introduction

This paper is a small contribution in the context of an ongoing effort directed towards the design of a calculus for constructing programs. Typically, the development of a program contains many parts that are quite standard, requiring no invention and posing no intellectual challenge of any kind. If, as is indeed the aim, this calculus is to be usable for constructing programs by completely formal manipulation, a major concern is the amount of labour currently required for such non-challenging parts.

On one level this concern can be addressed by building more or less specialised higher-level theories that can be drawn upon in a derivation, as is usual in almost all branches of mathematics, and good progress is being made here. This leaves us still with much low-level laboriousness, like administrative steps with little or no algorithmic content. Until now the efforts in reducing the overhead in low-level

Correspondence and offprint requests to: L. Meertens, CWI, Kruislaan 413, 1048 SJ Amsterdam, The Netherlands.

formal labour have concentrated on using equational reasoning together with specialised notations to avoid the introduction of dummy variables, in particular for "canned induction" in the form of promotion properties for homomorphisms – which have turned out to be ubiquitous. Recent developments and observations strongly suggest that further major gains in the proof methods are possible. One of the most promising developments is that it has become apparent that often a lengthy administrative calculation can be replaced by a single step by simply considering the types concerned. In the context of mechanical support for formal program construction, this can be mechanised in conjunction with mechanical type inference.

The present paper is concerned with another contribution to avoiding formal overhead, less dramatic, but probably still important, namely a generalisation of homomorphisms on initial data types, dubbed *paramorphisms*. While often much leverage is obtained by using homomorphisms, the occasions are also numerous where the gain of the homomorphism approach is less clear. It will be shown below that for a class of functions that are not themselves homomorphisms, but that satisfy a similar simple recursive pattern, a short-cut can be made, resulting in properties that are very similar to well-known properties of homomorphisms, such as the promotion properties. The recursive pattern involved is well known: it is essentially the same as the standard pattern used in the so-called elimination rules for a data type in constructive type theory (see, e.g., [BCM89]). The specific investigation of which these results form a part is not complete; rather, it has barely begun. There is some evidence that the approach can be generalised to other recursive patterns, possibly giving rise to a more elegant theory than expounded in this snapshot.

A few words on the notation used here are in order. As in all my other papers in this area, I have taken the liberty to conduct some further notational experiments. While deviation from "established" notation is hard on the reader, most current notations were clearly not designed with a view to the exigencies of calculation. Where notation is concerned, an attempt has been made to make this paper reasonably self-contained. However, not all non-standard notations are formally introduced – namely when their meaning can be inferred from the context.

A convention here, as well as in [Mee89], is to treat values of type A as nullary functions of type $A \leftarrow \mathbb{1}$. This makes is possible to denote function application unambigously as function composition, for which the symbol $\cdot$ is used. Within functional expressions this operator has the lowest precedence.

2. The Problem

Structural induction is the traditional technique for proving the equality of two functions that are defined on an inductively defined domain. Such functional equalities can also be proved by calculation in an equational proof style. This is based on the fact that, under a suitably chosen algebraic viewpoint, these fuctions are homomorphisms whose source algebra (an "algebraic data type") is initial. It is then possible to invoke elementary algebraic tools that replace the induction proofs [Gog80]. In particular, the *Unique Extension Property* and the *Promotion Theorem* for that source algebra provide the same proof-theoretic power as structural induction.

An examination of the proof obligations under the two approaches – traditional induction and algebra – reveals that they are ultimately identical. Thus it would seem that nothing is gained by using the homomorphic approach. However, the reduction in the labour needed to record the full proof is striking, especially when combined with a dummy-free style.

The explanation of this phenomenon is simple. A proof by structural induction follows a fixed ritual, that is repeated for each next proof. In the algebraic theorems, this proof has been given once and for all; what remains as the applicability condition is the heart of the matter. Moreover, the adoption of the algebraic viewpoint makes it possible to give concise notations for inductively defined functions [Mee86, Bir87, Bir89, Mal90], reducing the formal labour further.

The straightforward algebraic approach fails, however, when the definitional pattern of a function does not mimic the structural pattern of its domain. There is a standard trick that often makes it possible to apply the algebraic methods in such cases: "tuple" the function concerned together with the identity function, thus giving another function that *is* a homomorphism. Unfortunately, this method entails much formal overhead, making it less attractive for practical use.

In this paper we develop a generic extension of the theory that caters for a slightly more general class of definitional patterns. The term "generic" here means that the theory applies to all inductively defined data types.

3. A Simple Example

A simple inductive data type is formed by the naturals, with a unary constructor succ and a nullary constructor 0. Consider the following pattern of functional equations (with a function dummy F):

(1) $\Pi(F) := (F \cdot \mathsf{succ} = s \cdot F) \wedge (F \cdot 0 = z)$

This pattern has two yet unbounded function variables, a unary $s \in A \leftarrow A$ and a nullary $z \in A \leftarrow \mathbb{1}$, where A is some type. Given bindings for s and z, a function satisfying Π is a homomorphism from the algebra of the naturals with signature $(\mathsf{succ}, 0)$ to the algebra on A with signature (s, z). Since the algebra of naturals is defined as the initial algebra in this category, there exists – by the definition of "initial" – *exactly one* such homomorphism for each choice of (s, z). Therefore this is a means for defining functions on the naturals. Moreover, given two functions $f, g \in A \leftarrow \mathbb{N}$, we have

$$f = g \Leftarrow \Pi(f) \wedge \Pi(g)$$

This is the *Unique Extension Property* for the naturals. It can be seen that the task of proving one functional equality is replaced by the obligation of proving two times (for this data type) two such equalities, which, however, tends to be simpler. To invoke this instrument, a suitable instantantiation of s and z must be chosen, but if one of the two functions is inductively defined, not only is the necessary instantiation known, but we also have for free that that function satisfies Π.

After having established $\Pi(f)$ and $\Pi(g)$, to conclude now that $f = g$ the induction approach still has to go through the following ritual steps:

Basis: $f \cdot 0 = g \cdot 0$

$\equiv$ $\{(1): \Pi(f), \Pi(g)\}$

$z = z$

$\equiv$ $\{\text{reflexivity of } =\}$

true

Step: $f{\cdot}\mathsf{succ}{\cdot}n = g{\cdot}\mathsf{succ}{\cdot}n$

$\equiv$ $\{(1):\Pi(f), \Pi(g)\}$

$s{\cdot}f{\cdot}n = s{\cdot}g{\cdot}n$

$\Leftarrow$ {Leibniz}

$f{\cdot}n = g{\cdot}n$

$\equiv$ {Induction Hypothesis}

true □

The more complicated the inductive construction of the data type, the longer these rites.

Of course, in many cases the proof of the equality of two functions can be given purely equationally without appealing to either induction or these algebraic tools – otherwise no proof would be possible at all, since the common proof obligation has the shape of a set of functional equalities. Somewhat surprisingly, it turns out that often such a proof can also be substantially shortened by appealing to the Unique Extension Property.

Not all functions on the naturals are homomorphisms. Attempts to prove a (valid) functional equality for a non-homomorphic function by appeal to the Unique Extension Property are doomed to fail, and, in fact, even for homomorphisms success is not guaranteed. An example is the factorial function *fac*: there exists no *simple* function s such that $\Pi(fac)$ holds. However, there are simple functions $\otimes$ and z such that $\Pi\Pi(fac)$ holds, where $\Pi\Pi$ is the pattern given by

$$\Pi\Pi(F) := (F{\cdot}\mathsf{succ} = F \otimes\hat{}\ \mathsf{id}) \wedge (F{\cdot}0 = z)$$

(Here $\otimes$ is a binary function; between two functions returning natureals $\otimes\hat{}$ denotes the application of $\otimes$ to the results of these functions.) The instantiation that gives the factorial function is that in which $\otimes$ is taken to be the operation such that $m \otimes n = m \times (\mathsf{succ}{\cdot}n)$, and z is $\mathbf{1}$.

Like Π before, $\Pi\Pi$ has a unique solution for each choice for the unbound functions, in this case $\otimes$ and z. So the following is a valid statement:

$$f = g \Leftarrow \Pi\Pi(f) \wedge \Pi\Pi(g)$$

This can be shown to follow from the Unique Extension Property. But the proof of this is (even for a simple type like the naturals) non-obvious, lengthy, and in fact a new ritual that can be avoided by a properly designed extension of the theory.

4. Functors

Category theory provides some concepts that have proven indispensable in the formulation of generic theory, paramount among which is the notion of a functor. We give a treatment here slightly geared towards our purposes. In particular, we handle only the unary case, although the type constructors $\times$ and $+$ introduced below are also (binary) functors.

A functor is a pair of functions, one acting on types, and one on functions, with some further properties as stated below.

The application of a functor is denoted as a postfix operation. A functor $\dagger$ assigns to each type A a type $A\dagger$, and to each function $f \in A \leftarrow B$ a function $f\dagger \in A\dagger \leftarrow B\dagger$, where the latter mapping preserves function composition and identity; more precisely,

(2) $(f \cdot g)\dagger = f\dagger \cdot g\dagger$
(3) $\mathsf{id}\dagger = \mathsf{id}$

Equality (2) requires that $f \cdot g$ is well-typed; this is viewed as a wellformedness condition that applies in general to all constituents of functional expressions, and is from now on left implicit. In denoting an identity function, as in (3), its type is not stated, but in any context id is assumed to have a specific type, and so (3) stands for as many equalities as there are types.

An appeal to these equalities will be indicated in the justification of a proof step by "functor".

An important type construct is $\times$. It has a corresponding action on functions. (In [Mee89] I used different notations for $\times$ on types and on functions, which was a bad idea.) It is informally defined by

$A \times B :=$ "the type whose elements are the pairs (a, b) for $a \in A$ and $b \in B$",
$f \times g :=$ "the function that, applied to a pair (a, b), returns the pair $((f \cdot a), (g \cdot b))$"

We have the usual "projection functions" from $A \times B$ to A and B, which are denoted as

$$\ll \in A \leftarrow A \times B$$
$$\gg \in B \leftarrow A \times B$$

We also need the combinator that combines two functions $f \in A \leftarrow C$ and $g \in B \leftarrow C$ into one function

$$f \vartriangle g \in A \times B \leftarrow C$$

(The usual category-theory notation is (f, g).)

It can be characterised by

(4) $F \vartriangle G = H \equiv F = \ll \cdot H \wedge G = \gg \cdot H$

Its relationship with $\times$ is given by

$$f \times g = (f \cdot \ll) \vartriangle (g \cdot \gg)$$

which can in fact be taken as the definition of $\times$ on functions. From these equations all calculational properties of $\ll$, $\gg$, $\vartriangle$ and $\times$ are easily derived.

The relevant properties that we shall have occasion to use are

(5) $F \vartriangle G \cdot H = (F \cdot H) \vartriangle (G \cdot H)$
(6) $f \times g \cdot F \vartriangle G = (f \cdot F) \vartriangle (g \cdot G)$
(7) $\ll \cdot F \vartriangle G = F$
(8) $\ll \cdot F \vartriangle G = G$

A fact that we shall also use is that any mapping $\vartriangle F$, i.e., mapping a function f with the same domain as F to the function $f \vartriangle F$, is a bijection, so that

(9) $f = g \equiv f \vartriangle F = g \vartriangle F$

For discussing the application of the theory we need the dual type constructor $+$, which forms the "disjoint" or "tagged" union. Informally,

$A + B :=$ "the type whose elements are the union of the elements of A and B, tagged with the origin of an element (left or right)"
$f + g :=$ "the function that, for a left-tagged value a returns the left-tagged

value $f \cdot a$, and for a right-tagged value b the right-tagged value $g \cdot b$"

There are "injection functions" from each of A and B to $A+B$, which are not needed here, and a combinator that combines two functions $f \in C \leftarrow A$ and $g \in C \leftarrow B$ into one function

$$f \triangledown g \in C \leftarrow A+B$$

which amounts to applying f to left-tagged, and g to right-tagged values, thereby losing the tag information. (The usual category-theory notation is $[f, g]$.) There are similar (but dual) properties to those given for $\times$ and friends, which are not listed here since they will not be used.

From functors and $\times$ and $+$, we can form new functors. Functors can be formed by the composition of two functors, which is denoted by juxtaposition:

$$A(\dagger\ddagger) := (A\dagger)\ddagger$$
$$f(\dagger\ddagger) := (f\dagger)\ddagger$$

If B is some type, $\times B$ and $+B$ are functors, defined by

$$A(\times B) := A \times B$$
$$f(\times B) := f \times \mathsf{id}$$

and

$$A(+B) := A+B$$
$$f(+B) := f+\mathsf{id}$$

Combining this, we have, e.g., that $(\times B)(+\mathbb{1})$ is a functor, with

$$A((\times B)(+\mathbb{1})) = (A \times B)+\mathbb{1}$$

5. Types as Initial Fixed Points

The treatment in this section is mainly based on work by [Mal90]. Functors can be used to characterise a class of algebras with compatible signatures. If $\dagger$ is a functor, it characterises the class of algebras (A, ϕ), in which A is some type and the signature is

$$\phi \in A \leftarrow A\dagger$$

(For simplicity, we do not consider here the possibility of laws on the algebra. The theory developed here applies, nevertheless, equally to algebras with laws.)

For example, in the algebra of naturals $(\mathbb{N}, \mathsf{succ} \triangledown 0)$ the signature has type

$$\mathsf{succ} \triangledown 0 \in \mathbb{N} \leftarrow \mathbb{N}+\mathbb{1}$$

(which is equivalent to $(\mathsf{succ} \in \mathbb{N} \leftarrow \mathbb{N}) \wedge (0 \in \mathbb{N} \leftarrow \mathbb{1})$), so it belongs to the class characterised by the functor $+\mathbb{1}$.

If (A, ϕ) and (B, ψ) are two $\dagger$-algebras, then $h \in A \leftarrow B$ is called a *homomorphism* between these algebras when

$$\phi \cdot h\dagger = h \cdot \psi$$

We introduce a concise notation for the homomorphic property:

(10) $F \in \phi \overset{\dagger}{\leftarrow} \psi := \phi \cdot F\dagger = F \cdot \psi$

An algebra is called *initial* in the class of †-algebras if there is a unique homomorphism from it to each algebra in the class. If two algebras in the same class are initial, they are isomorphic: each can be obtained from the other by renaming. We assume that we can fix some representative, which is then called *the* initial algebra. For all functors introduced in this paper the class of algebras has an initial element. The initial algebra for † is denoted by $\mu(\dagger)$.

If we have

$$(L, \mathsf{in}) = \mu(\dagger)$$

then it can be shown (only for the lawless case!) that L and $L\dagger$ are isomorphic, which is the reason to call the type L the initial *fixed point* of †.

So the naturals can be defined by

$$(\mathbb{N}, \mathsf{succ} \triangledown 0) := \mu(+\mathbb{1})$$

The "snoc" lists over the base type A can likewise be defined by:

$$(A*, \prec\!\!\!- \triangledown \square) := \mu((\times A)(+\mathbb{1}))$$

Let (L, in) be the initial algebra $\mu(\dagger)$ for some functor †. A function $\phi \in A \leftarrow A\dagger$ determines uniquely an algebra (A, ϕ), and therefore a unique homomorphism $h \in A \leftarrow L$, that is, a function h satisfying

$$h \in \phi \overset{\dagger}{\leftarrow} \mathsf{in}$$

Denote it by $(\!|\phi|\!)$. It is useful to have a term for these homomorphisms whose domain is an initial algebra, and to this end we coin the term *catamorphism*. So we now have the following characterisation of catamorphisms:

Catamorphism characterisation

(11) $h = (\!|\phi|\!) \equiv h \in \phi \overset{\dagger}{\leftarrow} \mathsf{in}$

which we shall also invoke in the equivalent version

(12) $h = (\!|\phi|\!) \equiv \phi \cdot h\dagger = h \cdot \mathsf{in}$

obtained by unfolding definition (10), and in the weaker version

(13) $\phi \cdot (\!|\phi|\!)\dagger = (\!|\phi|\!) \cdot \mathsf{in}$

obtained by taking $h := (\!|\phi|\!)$.

The following two are now (almost) immediate;

Unique Extension Property (*UEP*)

$$f = g \Leftarrow (f \in \phi \overset{\dagger}{\leftarrow} \mathsf{in}) \wedge (g \in \phi \overset{\dagger}{\leftarrow} \mathsf{in})$$

Identity catamorphism

(14) $(\!|\mathsf{in}|\!) = \mathsf{id} \in L \leftarrow L$

Another easy consequence is

Promotion

(15) $(\!|\phi|\!) = f \cdot (\!|\psi|\!) \Leftarrow f \in \phi \overset{\dagger}{\leftarrow} \psi$

6. The Tupling Tactic

A function $f \in A \leftarrow L$ need not be a catamorphism by itself, which means that the results of the previous section cannot be applied as they stand. But, as we shall see, the result of tupling such an f with a suitably chosen catamorphism always is; in particular, tupling with the identity catamorphism achieves the effect. This tactic makes it possible to use the rules for catamorphisms on the tupled function.

So assume $\psi \in B \leftarrow B\dagger$, so that $f \vartriangle \llparenthesis\psi\rrparenthesis \in A \times B \leftarrow L$. We first derive a condition under which $f \vartriangle \llparenthesis\psi\rrparenthesis$ is a catamorphism:

(16) $f \vartriangle \llparenthesis\psi\rrparenthesis = \llparenthesis\phi \vartriangle (\psi \cdot \gg\dagger)\rrparenthesis \equiv \phi \cdot (f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger = f \cdot \mathsf{in}$

Proof.

$$
\begin{array}{ll}
 & f \vartriangle \llparenthesis\psi\rrparenthesis = \llparenthesis\phi \vartriangle (\psi \cdot \gg\dagger)\rrparenthesis \\
\equiv & \{(12)\text{: Catamorphism characterisation}\} \\
 & \phi \vartriangle (\psi \cdot \gg\dagger) \cdot (f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger = f \vartriangle \llparenthesis\psi\rrparenthesis \cdot \mathsf{in} \\
\equiv & \{(5)\text{: } F \vartriangle G \cdot H = (F \cdot H) \vartriangle (G \cdot H) \text{ (both sides)}\} \\
 & (\phi \cdot (f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger) \vartriangle (\psi \cdot \gg\dagger \cdot (f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger) = (f \cdot \mathsf{in}) \vartriangle (\llparenthesis\psi\rrparenthesis \cdot \mathsf{in}) \\
\equiv & \{(2)^{\smile}\text{: functor, } (13)^{\smile}\text{: Catamorphism (weak version)}\} \\
 & (\phi \cdot (f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger) \vartriangle (\psi \cdot (\gg \cdot f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger) = (f \cdot \mathsf{in}) \vartriangle (\psi \cdot \llparenthesis\psi\rrparenthesis\dagger) \\
\equiv & \{(8)\text{: } \gg \cdot F \vartriangle G = G;\ (9)\text{: } \vartriangle F \text{ is a bijection}\} \\
 & \phi \cdot (f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger = f \cdot \mathsf{in}
\end{array}
$$

□

For brevity, put $\chi := \phi \vartriangle (\psi \cdot \gg\dagger)$. The first equality in (16), which is then $f \vartriangle \llparenthesis\psi\rrparenthesis = \llparenthesis\chi\rrparenthesis$, can, by virtue of (4), be equivalently expressed in the form

(17) $f = \ll \cdot \llparenthesis\chi\rrparenthesis \wedge \llparenthesis\psi\rrparenthesis = \gg \cdot \llparenthesis\chi\rrparenthesis$

The validity of the first conjunct depends on f, ϕ and ψ, but the second conjunct is valid independent of the instantiation. This can be seen as follows:

$$
\begin{array}{ll}
 & \llparenthesis\psi\rrparenthesis = \gg \cdot \llparenthesis\chi\rrparenthesis \\
\Leftarrow & \{(15)\text{: Promotion}\} \\
 & \gg \in \psi \overset{\dagger}{\leftarrow} \chi \\
\equiv & \{(10)\text{: homomorphic property}\} \\
 & \psi \cdot \gg\dagger = \gg \cdot \chi \\
\equiv & \{\text{definition of } \chi\} \\
 & \psi \cdot \gg\dagger = \gg \cdot \phi \vartriangle (\psi \cdot \gg\dagger) \\
\equiv & \{(8)\text{: } \gg \cdot F \vartriangle G = G\} \\
 & \psi \cdot \gg\dagger = \psi \cdot \gg\dagger \\
\equiv & \{\text{reflexivity of } =\} \\
 & \mathsf{true}
\end{array}
$$

Hence, $f \vartriangle \llparenthesis\psi\rrparenthesis = \llparenthesis\chi\rrparenthesis \equiv f = \ll \cdot \llparenthesis\chi\rrparenthesis$, and therefore we obtain from (16), writing out χ in full again:

(18) $f = \ll \cdot \llparenthesis\phi \vartriangle (\psi \cdot \gg\dagger)\rrparenthesis \equiv \phi \cdot f \vartriangle \llparenthesis\psi\rrparenthesis)\dagger = f \cdot \mathsf{in}$

By instantiating $\psi := \mathsf{in}$ in 18, using that by (14) $\llparenthesis\mathsf{in}\rrparenthesis = \mathsf{id}$, we obtain

(19) $f = \ll \cdot \llparenthesis\phi \vartriangle (\mathsf{in} \cdot \gg\dagger)\rrparenthesis \equiv \phi \cdot (f \vartriangle \mathsf{id})\dagger = f \cdot \mathsf{in}$

We show, finally, that the second functional equality in (19) is satisfied by taking $\phi := f \cdot \mathsf{in} \cdot \gg\dagger$:

$$
\begin{aligned}
& f\cdot \mathsf{in}\cdot {\gg}\dagger\cdot(f\vartriangle \mathsf{id})\dagger \\
= \quad & \{(2)^{\smile}\text{: functor; } (8)\text{: } {\gg}\cdot F\vartriangle G = G\} \\
& f\cdot \mathsf{in}\cdot \mathsf{id}\dagger \\
= \quad & \{(3)\text{: functor; } \mathsf{id} \text{ is identity}\} \\
& f\cdot \mathsf{in}
\end{aligned}
$$

It follows that all functions $f \in A \leftarrow L$ can be expressed in the form $\ll\cdot(\!|\chi|\!)$.

7. Paramorphisms

Throughout this and the next section (L, in) denotes the initial algebra $\mu(\dagger)$ for some functor $\dagger$.

We have seen in the previous section that any function $f \in A \leftarrow L$ can be expressed in the form $\ll\cdot(\!|\chi|\!)$, in which $\chi = \phi\vartriangle(\mathsf{in}\cdot{\gg}\dagger)$ for a suitably chosen function $\phi \in A \leftarrow (A\times L)\dagger$. Note that $\ll\cdot(\!|\chi|\!)$ depends only on the choice for ϕ. In this section we introduce a notation for expressions of this form, and examine its properties.

So define, for $\phi \in A \leftarrow (A\times L)\dagger$,

(20) $[\![\phi]\!] := \ll\cdot(\!|\phi\vartriangle(\mathsf{in}\cdot{\gg}\dagger)|\!) \in A \leftarrow L$

Functions expressed in this form will be called *paramorphisms*. The actual notation used here is provisional, but is chosen to be reminiscent of the notation $(\!|\phi|\!)$ used for catamorphisms.

The calculational properties of catamorphisms all follow from the characterisation rule (11). Therefore we formulate now, likewise, a unique characterisation for paramorphisms. From it the other, calculationally possibly more important, properties follow easily. The characterisation has, in fact, already been given by (19); all we have to do is to "fold" this with the definition (20) of $[\![\phi]\!]$, thus obtaining

Paramorphism characterisation

(21) $f = [\![\phi]\!] \equiv \phi\cdot f\vartriangle \mathsf{id})\dagger = f\cdot\mathsf{in}$

The substitution $f := [\![\phi]\!]$ gives the weaker version

(22) $\phi\cdot([\![\phi]\!]\vartriangle \mathsf{id})\dagger = [\![\phi]\!]\cdot\mathsf{in}$

The uniqueness gives us

UEP for Paramorphisms

$$f = g \Leftarrow (\phi\cdot(f\vartriangle\mathsf{id})\dagger = f\cdot\mathsf{in}) \wedge (\phi\cdot(g\vartriangle\mathsf{id})\dagger = g\cdot\mathsf{in})$$

Whereas for catamorphisms the unique characterisation involves a condition of the same form as for the promotion law, here we find a divergence. The analogon of the promotion law for paramorphisms is

Parapromotion

(23) $[\![\phi]\!] = f\cdot[\![\psi]\!] \Leftarrow \phi\cdot(f\times\mathsf{id})\dagger = f\cdot\psi$

Proof.

$$
\begin{aligned}
& [\![\phi]\!] = f\cdot[\![\psi]\!] \\
\equiv \quad & \{(21)\text{: Paramorphism characterisation}\} \\
& \phi\cdot((f\cdot[\![\psi]\!])\vartriangle\mathsf{id})\dagger = f\cdot[\![\psi]\!]\cdot\mathsf{in} \\
\equiv \quad & \{\mathsf{id} \text{ is identity of } \cdot\}
\end{aligned}
$$

$$
\begin{array}{ll}
& \phi\cdot((f\cdot[\![\psi]\!])\vartriangle(\mathsf{id}\cdot\mathsf{id}))\dagger = f\cdot[\![\psi]\!]\cdot\mathsf{in} \\
\equiv & \{(6)^{\smile}\colon (f\cdot F)\vartriangle(g\cdot G) = f\times g\cdot F\vartriangle G\} \\
& \phi\cdot(f\times\mathsf{id}\cdot[\![\psi]\!]\vartriangle\mathsf{id})\dagger = f\cdot[\![\psi]\!]\cdot\mathsf{in} \\
\equiv & \{(2)\colon \text{functor}\} \\
& \phi\cdot(f\times\mathsf{id})\dagger\cdot([\![\psi]\!]\vartriangle\mathsf{id})\dagger = f\cdot[\![\psi]\!]\cdot\mathsf{in} \\
\equiv & \{(22)^{\smile}\colon \text{Paramorphism (weak version)}\} \\
& \phi\cdot(f\times\mathsf{id})\dagger\cdot([\![\psi]\!]\vartriangle\mathsf{id})\dagger = f\cdot\psi\cdot([\![\psi]\!]\vartriangle\mathsf{id})\dagger \\
\Leftarrow & \{\text{Leibniz}\} \\
& \phi\cdot(f\times\mathsf{id})\dagger = f\cdot\psi
\end{array}
\qquad\square
$$

8. Relationship with Catamorphisms

We shall see now two ways in which paramorphisms and catamorphisms are related. Firstly, paramorphisms can be viewed as a generalisation of catamorphisms, in the sense that the characterisation for catamorphisms, (12), follows formally from that for paramorphisms, (21). To show this we have to express a catamorphism as a paramorphism. The crucial result is

(24) $h = [\![\phi\cdot\ll\dagger]\!] \equiv \phi\cdot h\dagger = h\cdot\mathsf{in}$

Proof.

$$
\begin{array}{ll}
& h = [\![\phi\cdot\ll\dagger]\!] \\
\equiv & \{(21)\colon \text{Paramorphism characterisation}\} \\
& \phi\cdot\ll\dagger\cdot(h\vartriangle\mathsf{id})\dagger = h\cdot\mathsf{in} \\
\equiv & \{(2)^{\smile}\colon \text{functor}\} \\
& \phi\cdot(\ll\cdot h\vartriangle\mathsf{id})\dagger = h\cdot\mathsf{in} \\
\equiv & \{(7)\colon \ll\cdot F\vartriangle G = F\} \\
& \phi\cdot h\dagger = h\cdot\mathsf{in}
\end{array}
\qquad\square
$$

The right-hand side of (24) is precisely the equivalent of $h = (\!|\phi|\!)$ figuring in (12); in other words, considering paramorphisms as primitive, $[\![\phi\cdot\ll\dagger]\!]$ can be viewed as a new definition of the catamorphism $(\!|\phi|\!)$. With this definition, then, (24) states the same as (12).

Secondly, note that the condition in the rule for paramorphism promotion, (23), can be expressed as a homomorphic property, namely as follows. Let $\ddagger$ denote the functor $(\times L)\dagger$, that is,

(25) $A\ddagger = (\mathrm{A}\times\mathrm{L})\dagger$
(26) $f\ddagger = (f\times\mathsf{id})\dagger$

Then

$$
\begin{array}{ll}
& f\in\phi\overset{\dagger}{\leftarrow}\psi \\
\equiv & \{(10)\colon \textit{homomorphic property}\} \\
& \phi\cdot f\ddagger = f\cdot\psi \\
\equiv & \{(26)\} \\
& \phi\cdot f\times\mathsf{id})\dagger = f\cdot\psi
\end{array}
$$

which is precisely the condition of (23). So a "parapromotable" function with respect to $\dagger$ is a true homomorphism in the category of $\ddagger$-algebras.

Put $(M, \mathsf{jn}) := \mu(\ddagger)$, in which we use jn as notation for the constructor to avoid confusion with the constructor in of L. We have $\mathsf{jn} \in M \leftarrow M\ddagger$, or, equivalently, expanding $\ddagger$ by means of (25),

$$\mathsf{jn} \in M \leftarrow (M \times L)\dagger$$

Therefore jn has a type that makes the form $[\![\mathsf{jn}]\!]$ meaningful. We give a name to this paramorphism:

(27) $\mathit{preds} := [\![\mathsf{jn}]\!] \in M \leftarrow L$

Now it turns out that all paramorphisms can be formed from this particular one by the composition with a catamorphism on the type M. To make explicit that these catamorphisms are defined on the initial $\ddagger$-algebra, rather than the $\dagger$-algebra as until now, we write them as $(\!|\phi|\!)_\ddagger$. The result is then

$$[\![\phi]\!] = (\!|\phi|\!)_\ddagger \cdot \mathit{preds}$$

Proof.

$$\begin{array}{ll}
 & [\![\phi]\!] = (\!|\phi|\!)_\ddagger \cdot \mathit{preds} \\
\equiv & \{(27){:}\,\mathit{preds}\} \\
 & [\![\phi]\!] = (\!|\phi|\!)_\ddagger \cdot [\![\mathsf{jn}]\!] \\
\Leftarrow & \{(23){:}\,\text{Parapromotion}\} \\
 & \phi \cdot ((\!|\phi|\!)_\ddagger \times \mathsf{id})\dagger = (\!|\phi|\!)_\ddagger \cdot \mathsf{jn} \\
\equiv & \{(26)^{\smile}{:}\,\ddagger\} \\
 & \phi \cdot (\!|\phi|\!)_\ddagger \ddagger = (\!|\phi|\!)_\ddagger \cdot \mathsf{jn} \\
\equiv & \{(13){:}\ \text{Catamorphism (weak version)}\} \\
 & \mathsf{true}
\end{array}$$

□

This result subsumes the parapromotion rule (23) in the sense that (23) can easily be obtained from it.

To conclude, we examine what this means for the initial example, the factorial function *fac*. Here $L := \mathbb{N}$, which is obtained by taking the initial fixed point of $\dagger := +\mathbb{1}$. Putting

$$\otimes := \times \cdot (\mathsf{id} \times \mathsf{succ})$$
$$1 := \mathsf{succ} \cdot 0$$

the recursive definition pattern of *fac* can be expressed as

$$\otimes \triangledown 1 \cdot (\mathit{fac} \vartriangle \mathsf{id}) + \mathsf{id} = \mathit{fac} \cdot \mathsf{succ} \triangledown 0$$

which equivales, by (21),

$$\mathit{fac} = [\![\otimes \triangledown 1]\!]$$

We have, further, $\ddagger = (\times \mathbb{N})(+\mathbb{1})$. Then (M, jn) is $(\mathbb{N}{*}, \dashv\!\!<\, \triangledown \square)$, the algebra of the finite lists of naturals, and thus $\mathit{preds} \in \mathbb{N}{*} \leftarrow \mathbb{N}$. It satisfies, by (21) with the proper instantiations, the pattern

$$\dashv\!\!<\, \triangledown \square \cdot (\mathit{preds} \vartriangle \mathsf{id}) + \mathsf{id} = \mathit{preds} \cdot \mathsf{succ} \triangledown 0$$

which in a more traditional style can be expressed as

$$\mathit{preds} \cdot \mathsf{succ} \cdot n = (\mathit{preds} \cdot n) \dashv\!\!<\, n$$
$$\mathit{preds} \bullet 0 = \square$$

or, informally, $\mathit{preds} \cdot n = [0, 1, \ldots, n-1]$. Catamorphisms on snoc-lists are also known as left-reduces, and another way of writing $(\!|\otimes \triangledown 1|\!)$ is $\otimes \not\rightarrow_1$ [Bir87, Bir89]. Thus,

$$\mathit{fac} = \otimes \not\rightarrow_1 \cdot \mathit{preds}$$

References

[BCM89]	Backhouse, R., Chisholm, P., Malcolm, G. and Saaman, E.: Do-it-Yourself Type Theory. *Formal Aspects of Computing*, **1**, 19–84 (1989).
[Bir87]	Bird, R. S.: An introduction to the Theory of Lists. In: *Logic of Programming and Calculi of Discrete Design*, M. Broy, (ed.), *NATO ASI Series*, vol. F36, pp. 5–42, Springer-Verlag, 1987.
[Bir89]	Bird, R. S.: Lectures on Constructive Functional Programming. In: *Constructive Methods in Computing Science*, M. Broy, (ed.), *NATO ASI Series*, vol. F55, pp. 151–216, Springer-Verlag, 1989.
[Gog80]	Goguen, J. A.: How to Prove Inductive Hypotheses Without Induction. In: *Proc. 5th Conf. on Automated Deduction*, W. Bibel and R. Kowalski (eds), LNCS 87, pp. 356–373, Springer-Verlag, 1980.
[Mal90]	Malcolm, G.: Data Structures and Program Transformation. *Science of Computer Programming*, **14**, 255–279 (1990).
[Mee86]	Meertens, L.: Algorithmics – towards Programming as a Mathematical Activity. In: *Proc. CWI Symp. on Mathematics and Computer Science*, J. W. de Bakker, M. Hazewinkel and J. K. Lenstra (eds), *CWI Monographs*, vol. 1, pp. 289–334, North-Holland, 1986.
[Mee89]	Meertens, L.: Constructing a Calculus of Programs. In: *Mathematics of Program Construction*, J. L. A. van de Snepscheut (ed.), LNCS 375, pp. 66–90, Springer-Verlag, 1989.

Received January 1990
Accepted in revised form June 1991 by R. C. Backhouse