Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Erik Meijer [*] Maarten Fokkinga [†] Ross Paterson [‡]

Abstract

We develop a calculus for lazy functional programming based on recursion operators associated with data type definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. We shall show that all example functions in Bird and Wadler's "Introduction to Functional Programming" can be expressed using these operators.

1 Introduction

Among the many styles and methodologies for the construction of computer programs the Squiggol style in our opinion deserves attention from the functional programming community. The overall goal of Squiggol is to *calculate* programs from their specification in the way a mathematician calculates solutions to differential equations, or uses arithmetic to solve numerical problems.

It is not hard to state, prove and use laws for well-known operations such as addition, multiplication and —at the function level— composition. It is, however, quite hard to state, prove and use laws for arbitrarily recursively defined functions, mainly because it is difficult to refer to the recursion scheme in isolation. The algorithmic structure is obscured by using unstructured recursive definitions. We crack this problem by treating various recursion schemes as separate higher order functions, giving each a notation of its own independent of the ingredients with which it constitutes a recursively defined function.

[*]University of Nijmegen, Department of Informatics, Toernooiveld 6525 ED Nijmegen, e-mail: `erik@cs.kun.nl`

[†]CWI, Amsterdam & University of Twente

[‡]Imperial College, London

This philosophy is similar in spirit to the 'structured programming' methodology for imperative programming. The use of arbitrary goto's is abandoned in favour of structured control flow primitives such as conditionals and while-loops that replace fixed patterns of goto's, so that reasoning about programs becomes feasible and sometimes even elegant. For functional programs the question is which recursion schemes are to be chosen as a basis for a calculus of programs. We shall consider several recursion operators that are naturally associated with algebraic type definitions. A number of general theorems are proven about these operators and subsequently used to transform programs and prove their correctness.

Bird and Meertens [4, 18] have identified several laws for specific data types (most notably *finite* lists) using which they calculated solutions to various programming problems. By embedding the calculus into a categorical framework, Bird and Meertens' work on lists can be extended to arbitrary, inductively defined data types [17, 12]. Recently the group of Backhouse [1] has extended the calculus to a relational framework, thus covering indeterminancy.

Independently, Paterson [21] has developed a calculus of functional programs similar in contents but very dissimilar in appearance (like many Australian animals) to the work referred to above. Actually if one pricks through the syntactic differences the laws derived by Paterson are the same and in some cases slightly more general than those developped by the Squiggolers.

This paper gives an extension of the theory to the context of lazy functional programming, i.e., for us a type is an ω-cpo and we consider only continuous functions between types (categorically, we are working in the category CPO). Working in the category SET as done by for example Malcolm [17] or Hagino [14] means that finite data types (defined as initial algebras) and infinite data types (defined as final co-algebras) constitute two different worlds. In that case it is not possible to define functions by induction (catamorphisms) that are applicable to both finite and infinite data types, and arbitrary recursive definitions are not allowed. Working in CPO has the advantage that the carriers of initial algebras and final co-algebras coincide, thus there is a single data type that comprises both finite and infinite elements. The price to be paid however is that partiality of both functions and values becomes unavoidable.

2 The data type of lists

We shall illustrate the recursion patterns of interest by means of the specific data type of cons-lists. So, the definitions given here are actually specific instances of those given in §4. Modern functional languages allow the definition of cons-lists over some type A by putting:

$$A* \;::=\; \mathsf{Nil} \mid \mathsf{Cons}\ (A \| A*)$$

The recursive structure of this definition is employed when writing functions $\in A* \to B$ that destruct a list; these have been called *catamorphisms* (from the greek preposition $\kappa\alpha\tau\alpha$ meaning “downwards” as in “catastrophe”). *Anamorphisms* are functions $\in B \to A*$ (from the greek preposition $\alpha\nu\alpha$ meaning “upwards” as in “anabolism”) that generate a list of type $A*$ from a seed from B. Functions of type $A \to B$ whose call-tree has the shape of a cons-list are called *hylomorphisms* (from the Aristotelian philosophy that form and matter are one, $\upsilon\lambda o\sigma$ meaning “dust” or “matter”).

Catamorphisms

Let $b \in B$ and $\oplus \in A \| B \to B$, then a list-catamorphism $h \in A* \to B$ is a function of the following form:

$$\begin{aligned} h\ \mathrm{Nil} &= b \\ h\ (\mathrm{Cons}\ (a, as)) &= a \oplus (h\ as) \end{aligned} \tag{1}$$

In the notation of Bird&Wadler [5] one would write $h = \mathit{foldr}\ b\ (\oplus)$. We write catamorphisms by wrapping the relevant constituents between so called banana brackets:

$$h = (\!| b, \oplus |\!) \tag{2}$$

Countless list processing functions are readily recognizable as catamorphisms, for example $\mathit{length} \in A* \to \mathrm{Num}$, or $\mathit{filter}\ p \in A* \to A*$, with $p \in A \to \mathrm{bool}$.

$$\begin{aligned} \mathit{length} &= (\!| 0, \oplus |\!)\ \mathit{where}\ a \oplus n = 1 + n \\ \mathit{filter}\ p &= (\!| \mathit{Nil}, \oplus |\!) \\ &\quad\ \mathit{where}\ a \oplus as = \mathrm{Cons}\ (a, as), \quad p\ a \\ &\qquad\qquad\qquad\quad = as, \qquad\qquad\quad \neg p\ a \end{aligned}$$

Separating the recursion pattern for catamorphisms $(\!| \_ |\!)$ from its ingredients b and $\oplus$ makes it feasible to reason about catamorphic programs in an algebraic way. For example the *Fusion Law* for catamorphisms over lists reads:

$$f \circ (\!| b, \oplus |\!) = (\!| c, \otimes |\!) \quad \Leftarrow \quad f\ b = c\ \wedge\ f\ (a \oplus as) = a \otimes (f\ as)$$

Without special notation pinpointing catas, such as $(\!| \_ |\!)$ or foldr, we would be forced to formulate the fusion law as follows.

> Let h, g be given by
>
> $$\begin{aligned} h\ \mathrm{Nil} &= b & \qquad g\ \mathrm{Nil} &= c \\ h\ (\mathrm{Cons}\ (a, as)) &= a \oplus (h\ as) & \qquad g\ (\mathrm{Cons}\ (a, as)) &= a \otimes (g\ as) \end{aligned}$$
>
> then $f \circ h = g$ if $f\ b = c$ and $f\ (a \oplus as) = a \otimes (f\ as)$.

A clumsy way of stating such a simple algebraic property.

Anamorphisms

Given a predicate $p \in B \to bool$ and a function $g \in B \to A\|B$, a list-anamorphism $h \in B \to A*$ is defined as:

$$\begin{aligned} h\ b &= Nil, & p\ b \\ &= Cons\ (a, h\ b'), & otherwise \\ & \quad where\ (a, b') = g\ b \end{aligned} \tag{3}$$

Anamorphisms are not well-known in the functional programming folklore, they are called $unfold$ by Bird&Wadler, who spend only few words on them. We denote anamorphisms by wrapping the relevant ingredients between concave lenses:

$$h = [\![g, p]\!] \tag{4}$$

Many important list-valued functions are anamorphisms; for example $zip \in A*\|B* \to (A\|B)*$ which 'zips' a pair of lists into a list of pairs.

$$\begin{aligned} zip &= [\![g, p]\!] \\ p\ (as, bs) &= (as = Nil) \lor (bs = Nil) \\ g\ (Cons\ (a, as), Cons\ (b, bs)) &= ((a, b), (as, bs)) \end{aligned}$$

Another anamorphism is $iterate$ f which given a, constructs the infinite list of iterated applications of f to a.

$$iterate\ f = [\![g, false^\bullet]\!]\ where\ g\ a = (a, f\ a)$$

We use $c^\bullet$ to denote the constant function $\lambda x.c$.

Given $f \in A \to B$, the map function $f* \in A* \to B*$ applies f to every element in a given list.

$$\begin{aligned} f*Nil &= Nil \\ f*(Cons\ (a, as)) &= Cons\ (f\ a, f*as) \end{aligned}$$

Since a list appears at both sides of its type, we might suspect that map can be written both as a catamorphism and as an anamorphisms. Indeed this is the case. As catamorphism: $f* = (\!| Nil, \oplus |\!)$ where $a \oplus bs = Cons\ (f\ a, bs)$, and as anamorphism $f* = [\![g, p]\!]$ where $p\ as = (as = Nil)$ and $g\ (Cons\ (a, as)) = (f\ a, as)$.

Hylomorphisms

A recursive function $h \in A \to C$ whose call-tree is isomorphic to a cons-list, i.e., a linear recursive function, is called a hylomorphism. Let $c \in C$ and $\oplus \in B\|C \to C$ and $g \in A \to B\|A$

and $p \in A \to bool$ then these determine the hylomorphism h

$$\begin{aligned} h\ a &= c, & p\ a \\ &= b \oplus (h\ a'), & otherwise \\ & \quad where\ (b, a') = g\ a \end{aligned} \tag{5}$$

This is exactly the same structure as an anamorphism except that Nil has been replaced by c and $Cons$ by $\oplus$. We write hylomorphisms by wrapping the relevant parts into envelopes.

$$h \;=\; [\![(c, \oplus), (g, p)]\!] \tag{6}$$

A hylomorphism corresponds to the composition of an anamorphism that builds the call-tree as an explicit data structure and a catamorphism that reduces this data object into the required value.

$$[\![(c, \oplus), (g, p)]\!] \;=\; (\!|c, \oplus|\!) \circ [\!(g, p)\!]$$

A proof of this equality will be given in §15.

An archetypical hylomorphism is the factorial function:

$$\begin{aligned} fac &= [\![(1, \times), (g, p)]\!] \\ p\ n &= n = 0 \\ g\ (1+n) &= (1+n, n) \end{aligned}$$

Paramorphisms

The hylomorphism definition of the factorial maybe correct but is unsatisfactory from a theoretic point of view since it is not inductively defined on the data type $num ::= 0 \mid 1 + num$. There is however no 'simple' φ such that $fac = (\!|\varphi|\!)$. The problem with the factorial is that it "eats its argument and keeps it too" [27], the brute force catamorphic solution would therefore have fac' return a pair $(n, n!)$ to be able to compute $(n+1)!$.

Paramorphisms were investigated by Meertens [19] to cover this pattern of primitive recursion. For type num a paramorphism is a function h of the form:

$$\begin{aligned} h\ 0 &= b \\ h\ (1+n) &= n \oplus (h\ n) \end{aligned} \tag{7}$$

For lists a paramorphism is a function h of the form:

$$\begin{aligned} h\ Nil &= b \\ h\ (Cons\ (a, as)) &= a \oplus (as, h\ as) \end{aligned}$$

We write paramorphisms by wrapping the relevant constituents in barbed wire $h = \{\!|b, \oplus|\!\}$, thus we may write $fac = \{\!|1, \oplus|\!\}$ where $n \oplus m = (1 + n) \times m$. The function $tails \in A* \to A**$, which gives the list of all tail segments of a given list is defined by the paramorphism $tails = \{\!|Cons\ (Nil, Nil), \oplus|\!\}$ where $a \oplus (as, tls) = Cons\ (Cons\ (a, as), tls)$.

3 Algebraic data types

In the preceding section we have given specific notations for some recursion patterns in connection with the particular type of cons-lists. In order to define the notions of cata-, ana-, hylo- and paramorphism for arbitrary data types, we now present a generic theory of data types and functions on them. For this we consider a recursive data type (also called 'algebraic' data type in Miranda) to be defined as the least fixed point of a functor[1].

Functors

A bifunctor $\dagger$ is a binary operation taking types into types and functions into functions such that if $f \in A \to B$ and $g \in C \to D$ then $f \dagger g \in A \dagger C \to B \dagger D$, and which preserves identities and composition:

$$\begin{aligned} id \dagger id &= id \\ f \dagger g \circ h \dagger j &= (f \circ h) \dagger (g \circ j) \end{aligned}$$

Bifunctors are denoted by $\dagger, \ddagger, \S, \ldots$

A monofunctor is a unary type operation $\textsc{f}$, which is also an operation on functions, $\textsc{f} \in (A \to B) \to (A\textsc{f} \to B\textsc{f})$ that preserves the identity and composition. We use $\textsc{f}, \textsc{g}, \ldots$ to denote monofunctors. In view of the notation $A*$ we write the application of a functor as a postfix: $A\textsc{f}$. In §5 we will show that $*$ is a functor indeed.

The data types found in all current functional languages can be defined by using the following basic functors.

Product The (lazy) product $D \| D'$ of two types D and D' and its operation $\|$ on functions are defined as:

$$D \| D' = \{(d, d') \mid d \in D, d' \in D'\}$$

[1]We give the definitions of various concepts of category theory only for the special case of the category CPO. Also 'functors' are really endo-functors, and so on.

$$(f \| g)\ (x, x') \;=\; (f\ x, g\ x')$$

Closely related to the functor $\|$ are the projection and tupling combinators:

$$\begin{aligned} \grave{\pi}\ (x, y) &= x \\ \acute{\pi}\ (x, y) &= y \\ (f \vartriangle g)\ x &= (f\ x, g\ x) \end{aligned}$$

Using $\grave{\pi}, \acute{\pi}$ and $\vartriangle$ we can express $f \| g$ as $f \| g = (f \circ \grave{\pi}) \vartriangle (g \circ \acute{\pi})$. We can also define $\vartriangle$ using $\|$ and the doubling combinator $\Delta\ x = (x, x)$, since $f \vartriangle g = f \| g \circ \Delta$.

Sum The sum $D \mid D'$ of D and D' and the operation $\mid$ on functions are defined as:

$$\begin{aligned} D \mid D' &= (\{0\} \| D) \cup (\{1\} \| D') \cup \{\perp\} \\ (f \mid g)\ \perp &= \perp \\ (f \mid g)\ (0, x) &= (0, f\ x) \\ (f \mid g)\ (1, x') &= (1, g\ x') \end{aligned}$$

The arbitrarily chosen numbers 0 and 1 are used to 'tag' the values of the two summands so that they can be distinguished. Closely related to the functor $\mid$ are the injection and selection combinators:

$$\begin{aligned} \grave{\imath}\ x &= (0, x) \\ \acute{\imath}\ y &= (1, y) \\ (f \triangledown g)\ \perp &= \perp \\ (f \triangledown g)\ (0, x) &= f\ x \\ (f \triangledown g)\ (1, y) &= g\ y \end{aligned}$$

with which we can write $f \mid g = (\grave{\imath} \circ f) \triangledown (\acute{\imath} \circ g)$. Using ∇ which removes the tags from its argument, $\nabla\ \perp = \perp$ and $\nabla\ (i, x) = x$, we can define $f \triangledown g = \nabla \circ f \mid g$.

Arrow The operation $\rightarrow$ that forms the function space $D \rightarrow D'$ of continuous functions from D to D', has as action on functions the 'wrapping' function:

$$(f \rightarrow g)\ h \;=\; g \circ h \circ f$$

Often we will use the alternative notation $(g \leftarrow f)\ h = g \circ h \circ f$, where we have swapped the arrow already so that upon application the arguments need not be moved, thus localizing the changes occurring during calculations. The functional $(f \overset{F}{\leftarrow} g)\ h = f \circ h_F \circ g$ wraps its F-ed argument between f and g.

Closely related to the $\rightarrow$ are the combinators:

$$\begin{array}{rcl} \mathit{curry}\ f\ x\ y & = & f\ (x,y) \\ \mathit{uncurry}\ f\ (x,y) & = & f\ x\ y \\ \mathit{eval}\ (f,x) & = & f\ x \end{array}$$

Note that $\rightarrow$ is contra-variant in its first argument, i.e. $(f \rightarrow g) \circ (h \rightarrow j) = (h \circ f) \rightarrow (g \circ j)$.

Identity, Constants The identity functor I is defined on types as $D\mathsf{I} = D$ and on functions as $f\mathsf{I} = f$. Any type D induces a functor with the same name $\underline{D}$, whose operation on objects is given by $C\underline{D} = D$, and on functions $f\underline{D} = id$.

Lifting For mono-functors F, G and bi-functor $\dagger$ we define the mono-functors FG and $\mathsf{F}\dagger\mathsf{G}$ by

$$\begin{array}{rcl} x(\mathsf{FG}) & = & (x\mathsf{F})\mathsf{G} \\ x(\mathsf{F}\dagger\mathsf{G}) & = & (x\mathsf{F}) \dagger (x\mathsf{G}) \end{array}$$

for both types and functions x.

In view of the first equation we need not write parenthesis in $x\mathsf{FG}$. Notice that in $(\mathsf{F}\dagger\mathsf{G})$ the bi-functor $\dagger$ is 'lifted' to act on functors rather than on objects; $(\mathsf{F}\dagger\mathsf{G})$ is itself a mono-functor.

Sectioning Analogous to the sectioning of binary operators, $(a\oplus)\ b = a \oplus b$ and $(\oplus b)\ a = a \oplus b$ we define sectioning of bi-functors $\dagger$;

$$\begin{array}{rcl} (A\dagger) & = & \underline{A}\dagger\mathsf{I} \\ (f\dagger) & = & f \dagger id \end{array}$$

hence $B(A\dagger) = A \dagger B$ and $f(A\dagger) = id \dagger f$. Similarly we can define sectioning of $\dagger$ in its second argument, i.e. $(\dagger B)$ and $(\dagger f)$.

It is not too difficult to verify the following two properties of sectioned functors:

$$\begin{array}{rcll} (f\dagger) \circ g(A\dagger) & = & g(B\dagger) \circ (f\dagger) & \text{for all } f \in A \rightarrow B \end{array} \tag{8}$$

$$(f\dagger) \circ (g\dagger) = ((f \circ g)\dagger) \tag{9}$$

Taking $f \dagger g = g \rightarrow f$, thus $(f\dagger) = (f\circ)$ gives some nice laws for function composition.

Laws for the basic combinators

There are various equations involving the above combinators, we state nothing but a few of these. In parsing an expression function composition has least binding power while $||$ binds stronger than $|$.

$$
\begin{aligned}
\grave{\pi} \circ f||g &= f \circ \grave{\pi} & f \mid g \circ \grave{\imath} &= \grave{\imath} \circ f \\
\grave{\pi} \circ f \vartriangle g &= f & f \triangledown g \circ \grave{\imath} &= f \\
\acute{\pi} \circ f||g &= g \circ \acute{\pi} & f \mid g \circ \acute{\imath} &= \acute{\imath} \circ g \\
\acute{\pi} \circ f \vartriangle g &= g & f \triangledown g \circ \acute{\imath} &= g \\
(\grave{\pi} \circ h) \vartriangle (\acute{\pi} \circ h) &= h & (h \circ \grave{\imath}) \triangledown (h \circ \acute{\imath}) &= h \Leftarrow h \text{ strict} \\
\grave{\pi} \vartriangle \acute{\pi} &= id & \grave{\imath} \triangledown \acute{\imath} &= id \\
f||g \circ h \vartriangle j &= (f \circ h) \vartriangle (g \circ j) & f \triangledown g \circ h \mid j &= (f \circ h) \triangledown (g \circ j) \\
f \vartriangle g \circ h &= (f \circ h) \vartriangle (g \circ h) & f \circ g \triangledown h &= (f \circ g) \triangledown (f \circ h) \Leftarrow f \text{ strict} \\
f||g = h||j &\equiv f = h \wedge g = j & f \mid g = h \mid j &\equiv f = h \wedge g = j \\
f \vartriangle g = h \vartriangle j &\equiv f = h \wedge g = j & f \triangledown g = h \triangledown j &\equiv f = h \wedge g = j
\end{aligned}
$$

A nice law relating $\vartriangle$ and $\triangledown$ is the *abides law*:

$$(f \vartriangle g) \triangledown (h \vartriangle j) \quad = \quad (f \triangledown h) \vartriangle (g \triangledown j) \tag{10}$$

Varia

The one element type is denoted **1** and can be used to model constants of type A by nullary functions of type $\mathbf{1} \to A$. The only member of **1** called *void* is denoted by $()$.

In some examples we use for a given predicate $p \in A \to bool$, the function:

$$
\begin{aligned}
p? &\in A \to A \mid A \\
p?\ a &= \bot, \quad p\ a = \bot \\
&= \grave{\imath}\ a, \quad p\ a = true \\
&= \acute{\imath}\ a, \quad p\ a = false
\end{aligned}
$$

thus $f \triangledown g \circ p?$ models the familiar conditional **if** p **then** f **else** g **fi**. The function VOID maps its argument to void: $\mathrm{VOID}\ x = ()$. Some laws that hold for these functions are:

$$
\begin{aligned}
\mathrm{VOID} \circ f &= \mathrm{VOID} \\
p? \circ x &= x \mid x \circ (p \circ x)?
\end{aligned}
$$

In order to make recursion explicit, we use the operator $\mu \in (A \to A) \to A$ defined as:

$$\mu\ f \quad = \quad x \textit{ where } x = f\ x$$

We assume that recursion (like $x = f\ x$) is well defined in the meta-language.

Let F, G be functors and $\varphi_A \in A\text{F} \to A\text{G}$ for any type A. Such a φ is called a *polymorphic* function. A *natural transformation* is a family of functions φ_A (omitting subscripts whenever possible) such that:

$$\forall f : f \in A \to B : \varphi_B \circ f\text{F} = f\text{G} \circ \varphi_A \tag{11}$$

As a convenient shorthand for (11) we use $\varphi \in \text{F} \dot{\to} \text{G}$ to denote that φ is a natural transformation. The "Theorems For Free!" theorem of Wadler, deBruin and Reynolds [28, 9, 22] states that any function definable in the polymorphic λ-calculus is a natural transformation. If φ is defined using μ, one can only conclude that (11) holds for strict f.

Recursive types

After all this stuff on functors we have finally armed ourselves sufficiently to abstract from the peculiarities of cons-lists, and formalize recursively defined data types in general.

Let F be a monofunctor whose operation of functions is continuous, i.e., all monofunctors defined using the above basic functors or any of the map-functors introduced in §5. Then there exists a type L and two strict functions $in_\text{F} \in L\text{F} \to L$ and $out_\text{F} \in L \to L\text{F}$ (omitting subscripts whenever possible) which are each others inverse and even $id = \mu(in \overset{\text{F}}{\leftarrow} out)$ [6, 23, 16, 24, 30, 12]. We let μF denote the pair (L, in) and say that it is "the least fixed point of F". Since in and out are each others inverses we have that $L\text{F}$ is isomorphic to L, and indeed L is — upto isomorphism — a fixed point of F.

For example taking $X\text{L} = \mathbf{1} \mid A \| X$, we have that $(A*, in) = \mu\text{L}$ defines the data type of cons-lists over A for any type A. If we put $Nil = in \circ \grave{\imath} \in \mathbf{1} \to A*$ and $Cons = in \circ \acute{\imath} \in A \| A* \to A*$, we get the more familiar $(A*, Nil \triangledown Cons) = \mu\text{L}$. Another example of data types, binary trees with leaves of type A results from taking the least fixed point of $X\text{T} = \mathbf{1} \mid A \mid X \| X$. Backward lists with elements of type A, or snoc lists as they are sometimes called, are the least fixed point of $X\text{L} = \mathbf{1} \mid X \| A$. Natural numbers are specified as the least fixed point of $X\text{N} = \mathbf{1} \mid X$.

4 Recursion Schemes

Now that we have given a generic way of defining recursive data types, we can define cata-, ana-, hylo- and paramorphisms over arbitrary data types. Let $(L, in) = \mu\text{F}$, $\varphi \in A\text{F} \to A$, $\psi \in$

$A \to A_F$, $\xi \in (A||L)_F \to A$ then we define

$$(\!|\varphi|\!)_F \;=\; \mu(\varphi \overset{F}{\leftarrow} out) \tag{12}$$

$$[\![\psi]\!]_F \;=\; \mu(in \overset{F}{\leftarrow} \psi) \tag{13}$$

$$[\![\varphi, \psi]\!]_F \;=\; \mu(\varphi \overset{F}{\leftarrow} \psi) \tag{14}$$

$$\{\!|\xi|\!\}_F \;=\; \mu(\lambda f.\; \xi \circ (id \vartriangle f)_F \circ out) \tag{15}$$

When no confusion can arise we omit the F subscripts.

Definition (13) agrees with the definition given in §2; where we wrote $(\!|e, \oplus|\!)$ we now write $(\!|e^\bullet \triangledown (\oplus)|\!)$.

Definition (14) agrees with the informal one given earlier on; the notation $[\![g, p]\!]$ of §2 now becomes $[\![(VOID \mid g) \circ p?]\!]$.

Definition (15) agrees with the earlier one in the sense that taking $\varphi = c^\bullet \triangledown \oplus$ and $\psi = (VOID \mid g) \circ p?$ makes $[\![(c^\bullet, \oplus), (g, p)]\!]$ equal to $[\![\varphi, \psi]\!]$.

Definition (15) agrees with the description of paramorphisms as given in §2 in the sense that $\{\!|b, \oplus|\!\}$ equals $\{\!|b^\bullet \triangledown (\oplus)|\!\}$ here.

Program Calculation Laws

Rather than letting the programmer use explicit recursion, we encourage the use of the above fixed recursion patterns by providing a shopping list of laws that hold for these patterns. For each Ω-morphism, with $\Omega \in \{$cata, ana, para$\}$, we give an *evaluation rule*, which shows how such a morphism can be evaluated, a *Uniqueness Property*, a canned induction proof for a given function to be a Ω-morphism, and a *fusion law*, which shows when the composition of some function with an Ω-morphism is again an Ω-morphism. All these laws can be proved by mere equational reasoning using the following properties of general recursive functions. The first one is a 'free theorem' for the fixed point operator $\mu \in (A \to A) \to A$

$$f\ (\mu g) = \mu h \;\Leftarrow\; f \text{ strict} \;\wedge\; f \circ g = h \circ f \tag{16}$$

Theorem (16) appears under different names in many places[2] [20, 8, 2, 15, 7, 25, 13, 31]. In this paper it will be called *fixed point fusion*.

The strictness condition in (16) can sometimes be relaxed by using

$$f\ (\mu g) = f'\ (\mu g') \;\Leftarrow\; f \perp = f' \perp \;\wedge\; f \circ g = h \circ f \;\wedge\; f' \circ g' = h \circ f' \tag{17}$$

[2] Other references are welcome.

Fixed point induction over the predicate $P(g, g') \equiv f\ g = f'\ g'$ will prove (17).

For hylomorphisms we prove that they can be split into an ana- and a catamorphism and show how computation may be shifted within a hylomorphism. A number of derived laws show the relation between certain cata- and anamorphisms. These laws are not valid in SET. The hylomorphism laws follow from the following theorem:

$$\mu(f \overset{F}{\leftarrow} g) \circ \mu(h \overset{F}{\leftarrow} j) = \mu(f \overset{F}{\leftarrow} j) \quad \Leftarrow \quad g \circ h = id \tag{18}$$

Catamorphisms

Evaluation rule The *evaluation rule* for catamorphisms follows from the fixed point property $x = \mu f \Rightarrow x = f\ x$:

$$(\!|\varphi|\!) \circ in \quad = \quad \varphi \circ (\!|\varphi|\!)_L \tag{CataEval}$$

It states how to evaluate an application of $(\!|\varphi|\!)$ to an arbitrary element of L (returned by the constructor in); namely, apply $(\!|\varphi|\!)$ recursively to the argument of in and then φ to the result.

For cons lists $(A*, Nil \triangledown Cons) = \mu L$ where $X_L = \mathbf{1} \mid A||X$ and $f_L = id \mid id||f$ with catamorphism $(\!|c \triangledown \oplus|\!)$ the evaluation rule reads:

$$(\!|c \triangledown \oplus|\!) \circ Nil \quad = \quad c \tag{19}$$

$$(\!|c \triangledown \oplus|\!) \circ Cons \quad = \quad \oplus \circ id||(\!|c \triangledown \oplus|\!) \tag{20}$$

i.e. the variable free formulation of (1). Notice that the constructors, here $Nil \triangledown Cons$ are used for parameter pattern matching.

UP for catamorphisms The *Uniqueness Property* can be used to prove the equality of two functions without using induction explicitly.

$$f = (\!|\varphi|\!) \quad \equiv \quad f \circ \perp = (\!|\varphi|\!) \circ \perp \ \wedge \ f \circ in = \varphi \circ f_L \tag{CataUP}$$

A typical induction proof for showing $f = (\!|\varphi|\!)$ takes the following steps. Check the induction base: $f \circ \perp = (\!|\varphi|\!) \circ \perp$. Assuming the induction hypothesis $f_L = (\!|\varphi|\!)_L$ proceed by calculating:

$$\begin{array}{ll} & f \circ in = \ldots = \varphi \circ f_L \\ = & \quad \text{induction hypothesis} \\ & \varphi \circ (\!|\varphi|\!)_L \\ = & \quad \text{evaluation rule (CataEval)} \\ & (\!|\varphi|\!) \circ in \end{array}$$

to conclude that $f = (\!|\varphi|\!)$. The schematic set-up of such a proof is done once and for all, and built into law (CataUP). We are thus saved from the standard ritual steps; the last two lines in the above calculation, plus the declaration that 'by induction' the proof is complete.

The $\Rightarrow$ part of the proof for (CataUP) follows directly from the evaluation rule for catamorphisms. For the $\Leftarrow$ part we use the fixed point fusion theorem (17) with $f := (f\circ)$, $g := g' := \mathit{in} \overset{L}{\leftarrow} \mathit{out}$ and $f' := (\!|\varphi|\!)$. This gives us $f \circ \mu(\mathit{in} \overset{L}{\leftarrow} \mathit{out}) = (\!|\varphi|\!) \circ \mu(\mathit{in} \overset{L}{\leftarrow} \mathit{out})$ and since $\mu(\mathit{in} \overset{L}{\leftarrow} \mathit{out}) = \mathit{id}$ we are done.

Fusion law for catamorphisms The *Fusion Law* for catamorphisms can be used to transform the composition of a function with a catamorphism into a single catamorphism, so that intermediate values can be avoided. Sometimes the law is used the other way around, i.e. to split a function, in order to allow for subsequent optimizations.

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \quad \Leftarrow \quad f \circ \bot = (\!|\psi|\!) \circ \bot \ \wedge \ f \circ \varphi = \psi \circ f_L \qquad \text{(CataFusion)}$$

The fusion law can be proved using fixed point fusion theorem (17) with $f := (f\circ)$, $g := \varphi \overset{L}{\leftarrow} \mathit{out}$, $g' := \mathit{in} \overset{L}{\leftarrow} \mathit{out}$ and $f' := ((\!|\psi|\!)\circ)$.

A slight variation of the fusion law is to replace the condition $f \circ \bot = (\!|\psi|\!) \circ \bot$ by $f \circ \bot = \bot$, i.e. f is strict.

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \quad \Leftarrow \quad f \text{ strict} \ \wedge \ f \circ \varphi = \psi \circ f_L \qquad \text{(CataFusion')}$$

This law follows from (16). In actual calculations this latter law is more valuable as its applicability conditions are on the whole easier to check.

Injective functions are catamorphisms Let $f \in A \to B$ be a strict function with left-inverse g, then for any $\varphi \in A_F \to A$ we have

$$f \circ (\!|\varphi|\!) = (\!|f \circ \varphi \circ g_F|\!) \quad \Leftarrow \quad f \text{ strict} \ \wedge \ g \circ f = \mathit{id} \qquad (21)$$

Taking $\varphi = \mathit{in}$ we immediatly get that any strict injective function can be written as a catamorphism.

$$f = (\!|f \circ \mathit{in} \circ g_F|\!)_F \quad \Leftarrow \quad f \text{ strict} \ \wedge \ g \circ f = \mathit{id} \qquad (22)$$

Using this latter result we can write out in terms of in since $\mathit{out} = (\!|\mathit{out} \circ \mathit{in} \circ \mathit{in}_L|\!) = (\!|\mathit{in}_L|\!)$.

Catamorphisms preserve strictness The given laws for catamorphisms all demonstrate the importance of strictness, or generally of the behaviour of a function with respect to $\perp$. The following "poor man's strictness analyser" for that reason can often be put into good use.

$$\mu F \circ \perp = \perp \quad \Leftarrow \quad \forall f :: F\ f \circ \perp = \perp \tag{23}$$

The proof of (23) is by fixed point induction over $P(F) \equiv F \circ \perp = \perp$.

Specifically for catamorphisms we have

$$(\!|\varphi|\!)_L \circ \perp = \perp \quad \equiv \quad \varphi \circ \perp = \perp$$

if L is strictness preserving. The $\Leftarrow$ part of the proof directly follows from (23) and the definition of catamorphisms. The other way around is shown as follows

$$\begin{array}{ll}
& \perp \\
= & \quad \text{premise} \\
& (\!|\varphi|\!) \circ \perp \\
= & \quad in \circ \perp = \perp \\
& (\!|\varphi|\!) \circ in \circ \perp \\
= & \quad \text{evaluation rule} \\
& \varphi \circ (\!|\varphi|\!)L \circ \perp \\
= & \quad L \text{ preserves strictness} \\
& \varphi \circ \perp
\end{array}$$

Examples

Unfold-Fold Many transformations usually accomplished by the unfold-simplify-fold technique can be restated using fusion. Let $(Num*, Nil \triangledown Cons) = \mu L$, where $X_L = \mathbf{1} \mid Num \| X$ and $f_L = id \mid id \| f$ be the type of lists of natural numbers. Using fusion we derive an efficient version of $sum \circ squares$ where $sum = (\!|0^\bullet \triangledown +|\!)$ and $squares = (\!|Nil \triangledown (Cons \circ SQ \| id)|\!)$. Since sum is strict we just start calculating aiming at the discovery of a ψ that satisfies the condition of (CataFusion').

$$\begin{array}{ll}
& sum \circ Nil \triangledown (Cons \circ S \| id) \\
= & \\
& (sum \circ Nil) \triangledown (sum \circ Cons \circ SQ \| id) \\
= & \\
& Nil \triangledown ((+) \circ id \| sum \circ SQ \| id) \\
= & \\
& Nil \triangledown ((+) \circ SQ \| id \circ id \| sum) \\
= & \\
& Nil \triangledown ((+) \circ SQ \| id) \circ sumL
\end{array}$$

and conclude that $sum \circ squares = (\!|Nil \triangledown ((+) \circ SQ \| id)|\!)$.

A slightly more complicated problem is to derive a one-pass solution for

$$average \quad = \quad DIV \circ sum \vartriangle length$$

Using the tupling lemma of Fokkinga [10]

$$(\!|\varphi|\!)_L \vartriangle (\!|\psi|\!)_L \quad = \quad (\!|(\varphi \circ \grave{\pi}_L) \vartriangle (\psi \circ \acute{\pi}_L)|\!)$$

a simple calculation shows that $average = DIV \circ (\!|(0^\bullet \triangledown (+) \circ id \| \grave{\pi}) \vartriangle (0^\bullet \triangledown (+1) \circ \acute{\pi}|\!)$.

Accumulating Arguments An important item in the functional programmer's bag of tricks is the technique of *accumulating arguments* where an extra parameter is added to a function to accumulate the result of the computation. Though stated here in terms of catamorphisms over cons-lists, the same technique is applicable to other data types and other kind of morphisms as well.

$$\begin{array}{l} (\!|c^\bullet \triangledown \oplus|\!)\ l = (\!|(c\otimes)^\bullet \triangledown \ominus|\!)\ l\ \nu_\oplus\ where\ (a \ominus f)\ b = f\ (a \odot b) \\ \Leftarrow \\ a \otimes \nu_\oplus = a\ \wedge\ \bot \otimes a = \bot\ \wedge\ (a \oplus b) \otimes c = b \otimes (a \odot c) \end{array} \tag{24}$$

Theorem (24) follows from the fusion law by taking $Accu \circ (\!|c^\bullet \triangledown \oplus|\!) = (\!|(c\oplus)^\bullet \triangledown \ominus|\!)$ with $Accu\ a\ b = a \otimes b$.

Given the naive quadratic definition of $reverse \in A* \to A*$ as a catamorphism $(\!|Nil^\bullet \triangledown \oplus|\!)$ where $a \oplus as = as +\!\!+ (Cons\ (a, Nil))$, we can derive a linear time algorithm by instantiating (24) with $\oplus := +\!\!+$ and $\odot := Cons$ to get a function which accumulates the list being reversed as an additional argument: $(\!|id \triangledown \ominus|\!)$ where $(a \ominus as)\ bs = as\ (Cons\ (a, bs))$. Here $+\!\!+$ is the function that appends two lists, defined as $as +\!\!+ bs = (\!|id^\bullet \triangledown \oplus|\!)\ as\ bs$ where $a \oplus f\ bs = Cons\ (a, f\ bs)$.

In general catamorphisms of higher type $L \to (I \to S)$ form an interesting class by themselves as they correspond to *attribute grammars* [11].

Anamorphisms

Evaluation rule The evaluation rule for anamorphisms is given by:

$$out \circ [\!(\psi)\!] \quad = \quad [\!(\psi)\!]_L \circ \psi \tag{AnaEval}$$

It says what the result of an arbitrary application of $[\!(\psi)\!]$ looks like: the constituents produced by applying out can equivalently be obtained by first applying ψ and then applying $[\!(\psi)\!]_L$ recursively to the result.

Anamorphisms are real old fusspots to explain. To instantiate (AnaEval) for cons list we define:

$$\begin{aligned} \mathrm{hd} &= \perp \triangledown \grave{\pi} \circ \mathrm{out} \\ \mathrm{tl} &= \perp \triangledown \acute{\pi} \circ \mathrm{out} \\ \mathrm{is\_nil} &= \mathrm{true}^{\bullet} \triangledown \mathrm{false}^{\bullet} \circ \mathrm{out} \end{aligned}$$

Assuming that $f = [\!(\mathrm{VOID} \mid (h \vartriangle t) \circ p?)\!]$ we find after a little calculation that:

$$\begin{aligned} \mathrm{is\_nil} \circ f &= p \\ \mathrm{hd} \circ f &= h \Leftarrow \neg p \\ \mathrm{tl} \circ f &= t \Leftarrow \neg p \end{aligned}$$

which corresponds to the characterization of unfold given by Bird and Wadler [5] on page 173.

UP for anamorphisms The UP for anamorphisms is slightly simpler than the one for catamorphisms, since the base case does not have to be checked.

$$f = [\!(\varphi)\!] \quad \equiv \quad \mathrm{out} \circ f = f_L \circ \varphi \qquad \text{(AnaUP)}$$

To prove it we can use fixed point fusion theorem (16) with $f := (\circ f)$, $g := \mathrm{in} \overset{L}{\leftarrow} \mathrm{out}$ and $h := \mathrm{in} \overset{L}{\leftarrow} \psi$. This gives us $\mu(\mathrm{in} \overset{L}{\leftarrow} \mathrm{out}) \circ f = \mu(\mathrm{in} \overset{L}{\leftarrow} \psi)$ and again since $\mu(\mathrm{in} \overset{L}{\leftarrow} \mathrm{out}) = \mathrm{id}$ we are done.

Fusion law for anamorphisms The strictness requirement that was needed for catamorphisms can be dropped in the anamorphism case. The dual condition of $f \circ \perp = \perp$ for strictness is $\perp \circ f = \perp$ which is vacuously true.

$$[\!(\varphi)\!] \circ f = [\!(\psi)\!] \quad \Leftarrow \quad \varphi \circ f = f_L \circ \psi \qquad \text{(AnaFusion)}$$

This law can be proved by fixed point fusion theorem (16) with $f := (\circ f)$, $g := \mathrm{in} \overset{L}{\leftarrow} \varphi$ and $h := \mathrm{in} \overset{L}{\leftarrow} \psi$.

Any surjective function is an anamorphism The results (21) and (22) can be dualized for anamorphisms. Let $f \in B \to A$ a surjective function with right-inverse g, then for any $\psi \in A \to A_L$ we have

$$[\!(\psi)\!] \circ f = [\!(g_L \circ \psi \circ f)\!] \quad \Leftarrow \quad f \circ g = id \tag{25}$$

since $\psi \circ f = f_L \circ (g_L \circ \psi \circ f)$. The special case where ψ equals out yields that any surjective function can be written as an anamorphism.

$$f = [\!(g_L \circ out \circ f)\!]_L \quad \Leftarrow \quad f \circ g = id \tag{26}$$

As in has right-inverse out, we can express in using out by $in = [\!(out_L \circ out \circ in)\!] = [\!(out_L)\!]$.

Examples

Reformulated in the lense notation, the function $iterate\ f$ becomes:

$$iterate\ f \quad = \quad [\!(\acute{\imath} \circ id \vartriangle f)\!]$$

We have $[\!(\acute{\imath} \circ id \vartriangle f)\!] = [\!(VOID \mid id \vartriangle f \circ false^{\bullet}?)\!]\,(= [\!(id \vartriangle f, false^{\bullet})\!]$ in the notation of section 2).

Another useful list-processing function is $takewhile\ p$ which selects the longest initial segment of a list all whose elements satisfy p. In conventional notation:

$$\begin{array}{rcll} takewhile\ p\ Nil & = & Nil & \\ takewhile\ p\ (Cons\ a\ as) & = & Nil, & \neg p\ a \\ & = & Cons\ a\ (takewhile\ p\ as), & otherwise \end{array}$$

The anamorphism definition may look a little daunting at first:

$$takewhile\ p \quad = \quad [\!(\grave{\imath} \triangledown (VOID \mid id \circ (\neg p \circ \grave{\pi})?) \circ out)\!]$$

The function $f\ while\ p$ contains all repeated applications of f as long as predicate p holds:

$$f\ while\ p \quad = \quad takewhile\ p \circ iterate\ f$$

Using the fusion law (after a rather long calculation) we can show that $f\ while\ p = [\!(VOID \mid (id \vartriangle f) \circ \neg p?)\!]$.

Hylomorphisms

Splitting Hylomorphisms In order to prove that a hylomorphism can be split into an anamorphism followed by a catamorphism

$$[\![\varphi, \psi]\!] \quad = \quad (\!|\varphi|\!) \circ [\!(\psi)\!] \qquad \text{(HyloSplit)}$$

we can use the total fusion theorem (18).

Shifting law Hylomorphisms are nice since their decomposability into a cata- and an anamorphism allows us to use the respective fusion laws to shift computation in or out of a hylomorphism. The following *shifting law* shows how computations can be shifted within a hylomorphism.

$$[\![\varphi \circ \xi, \psi]\!]_L = [\![\varphi, \xi \circ \psi]\!]_M \quad \Leftarrow \quad \xi \in L \stackrel{\cdot}{\to} M \qquad \text{(HyloShift)}$$

The proof of this theorem is straightforward.

$$\begin{array}{ll} & [\![\varphi \circ \xi, \psi]\!]_L \\ = & \quad \text{definition hylo} \\ & \mu(\lambda f.\varphi \circ \xi \circ f_L \circ \psi) \\ = & \quad \xi \in L \stackrel{\cdot}{\to} M \\ & \mu(\lambda f.\varphi \circ f_M \circ \xi \circ \psi) \\ = & \quad \text{definition hylo} \\ & [\![\varphi, \xi \circ \psi]\!]_M \end{array}$$

An admittedly humbug example of (HyloShift) shows how left linear recursive functions can be transformed into right linear recursive functions. Let $f_L = id \mid f \| id$ and $f_R = id \mid id \| f$ define the functors which express left respectively right linear recursion, then if $x \oplus y = y \oplus x$ we have

$$\begin{array}{ll} & [\![c \triangledown \oplus, f \mid (h \vartriangle t) \circ p?]\!]_L \\ = & [\![c \triangledown \oplus \circ SWAP, f \mid (h \vartriangle t) \circ p?]\!]_L \\ = & \quad SWAP \in L \stackrel{\cdot}{\to} R \\ & [\![c \triangledown \oplus, SWAP \circ f \mid (h \vartriangle t) \circ p?]\!]_R \\ = & [\![c \triangledown \oplus, f \mid (t \vartriangle h) \circ p?]\!]_R \end{array}$$

where $SWAP = id \mid (\acute{\pi} \vartriangle \grave{\pi})$.

Relating cata- and anamorphisms

From the splitting and shifting law (HyloShift), (HyloSplit) and the fact that $(\!|\varphi|\!) = [\![\varphi, out]\!]$ and $[\!(\psi)\!] = [\![in, \psi]\!]$ we can derive a number of interesting laws which relate cata- and anamorphisms with each other.

$$(\!|in_M \circ \varphi|\!)_L = [\!(\varphi \circ out_L)\!]_M \quad \Leftarrow \quad \varphi \in L \dot{\to} M \tag{27}$$

Using this law we can easily show that

$$\begin{aligned} (\!|\varphi \circ \psi|\!)_L &= (\!|\varphi|\!)_M \circ [\!(\psi \circ out_L)\!]_M \Leftarrow \psi \in L \dot{\to} M & (28)\\ &= (\!|\varphi|\!)_M \circ (\!|in_M \circ \psi|\!)_L \Leftarrow \psi \in L \dot{\to} M & (29)\end{aligned}$$

$$\begin{aligned} [\!(\varphi \circ \psi)\!]_M &= (\!|in_M \circ \varphi|\!)_L \circ [\!(\psi)\!]_L \Leftarrow \varphi \in L \dot{\to} M & (30)\\ &= [\!(\varphi \circ out_L)\!]_M \circ [\!(\psi)\!]_L \Leftarrow \varphi \in L \dot{\to} M & (31)\end{aligned}$$

This set of laws will be used in §5.

From the total fusion theorem (18) we can derive:

$$[\!(\psi)\!]_L \circ (\!|\varphi|\!)_L = id \quad \Leftarrow \quad \psi \circ \varphi = id \tag{32}$$

Example: Reflecting binary trees

The type of binary trees with leaves of type A is given by $(tree\ A, in) = \mu L$ where $X_L = \mathbf{1} \mid A \mid X \| X$ and $f_L = id \mid id \mid g \| g$. Reflecting a binary tree can be defined by: $reflect = (\!|in \circ SWAP|\!)$ where $SWAP = id \mid id \mid (\acute{\pi} \vartriangle \grave{\pi})$. A simple calculation proves that $reflect \circ reflect = id$.

$$\begin{aligned} & reflect \circ reflect \\ = & \qquad SWAP \circ f_L = f_L \circ SWAP \\ & [\![SWAP \circ out]\!] \circ (\!|in \circ SWAP|\!) \\ = & \qquad SWAP \circ out \circ in \circ SWAP = id \\ & id \end{aligned}$$

Paramorphisms

The *evaluation rule* for paramorphisms is

$$\langle\!|\varphi|\!\rangle \circ in \quad = \quad \varphi \circ (id \vartriangle \langle\!|\varphi|\!\rangle)_L \tag{ParaEval}$$

The *UP* for paramorphisms is similar to that of catamorphisms:

$$f = (\!|\varphi|\!) \quad\equiv\quad f \circ \bot = (\!|\varphi|\!) \circ \bot \ \wedge\ f \circ \mathit{in} = \varphi \circ (\mathit{id} \vartriangle f)_L \qquad \text{(ParaUP)}$$

The *fusion law* for paramorphisms reads

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \quad\Leftarrow\quad f \text{ strict } \wedge\ f \circ \varphi = \psi \circ (\mathit{id} \| f)_L \qquad \text{(ParaFusion)}$$

Any function f (of the right type of course!) is a paramorphism.

$$f \quad=\quad (\!|f \circ \mathit{in} \circ \grave{\pi}_L|\!)$$

The usefulness of this theorem can be read from its proof.

$$
\begin{aligned}
& (\!|f \circ \mathit{in} \circ \grave{\pi}_L|\!) \\
= & \quad \text{definition (15)} \\
& \mu(\lambda g.f \circ \mathit{in} \circ \grave{\pi}_L \circ (\mathit{id} \vartriangle g)_L \circ \mathit{out}) \\
= & \quad \text{functor calculus} \\
& \mu(\lambda g.f \circ \mathit{in} \circ \mathit{out}) \\
= & \\
& f
\end{aligned}
$$

Example: composing paramorphisms from ana- and catamorphisms

A nice result is that any paramorphism can be written as the composition of a cata- and an anamorphism. Let $(L, \mathit{in}) = \mu_L$ be given, then define

$$
\begin{aligned}
X_M &= (L \| X)_L \\
h_M &= (\mathit{id} \| h)_L \\
(M, \mathit{IN}) &= \mu_M
\end{aligned}
$$

For natural numbers we get $X_M = (\mathit{Num} \| X)_L = \mathbf{1} \mid \mathit{Num} \| X$, i.e. $(\mathit{Num}*, \mathit{in}) = \mu_M$, which is the type of lists of natural numbers.

Now define $\mathit{preds} \in L \to M$ as follows:

$$\mathit{preds} \quad=\quad [\![\Delta_L \circ \mathit{out}_L]\!]_M$$

For the naturals we get $\mathit{preds} = [\![\mathit{id} \mid \Delta \circ \mathit{out}]\!]$, that is given a natural number $N = n$, the expression $\mathit{preds}\ N$ yields the list $[n-1, \ldots, 0]$.

Using preds we start calculating:

$$
\begin{array}{cl}
 & (\!|\varphi|\!)_M \circ preds \\
= & \\
 & (\!|\varphi|\!)_M \circ [\![\Delta_L \circ out_L]\!]_M \\
= & \\
 & \mu(\lambda f.\varphi \circ f_M \circ \Delta_L \circ out_L) \\
= & \\
 & \mu(\lambda f.\varphi \circ (id||f)_L \circ (id \vartriangle id)_L \circ out_L) \\
= & \\
 & \mu(\lambda f.\varphi \circ (id \vartriangle f)_L \circ out_L) \\
= & \\
 & \langle\!|\varphi|\!\rangle_L
\end{array}
$$

Thus $\langle\!|\varphi|\!\rangle_L = (\!|\varphi|\!)_M \circ preds$. Since $(\!|IN|\!)_M = id$ we immediately get $preds = \langle\!|IN|\!\rangle_L$.

5 Parametrized Types

In §2 we have defined for $f \in A \to B$, the map function $f* \in A* \to B*$. Two laws for $*$ are $id* = id$ and $(f \circ g)* = f* \circ g*$. These two laws precisely state that $*$ is a functor. Another characteristic property of map is that it leaves the 'shape' of its argument unchanged. It turns out that any *parametrized* data type comes equipped with such a map functor. A parametrized type is a type defined as the least fixed point of a sectioned bifunctor. Contrary to Malcolms approach [17] map can be defined both as a catamorphism and as an anamorphism.

Maps

Let $\dagger$ be a bi-functor, then we define the functor $*$ on objects A as the parametrized type $A*$ $where$ $(A*, in) = \mu(A\dagger)$, and on functions $f \in A \to B$ as:

$$f* \;\; = \;\; (\!| in \circ (f\dagger) |\!)_{(A\dagger)} \tag{33}$$

Since $(f\dagger) \in (A\dagger) \dot{\to} (B\dagger)$, from (27) we immediately get an alternative version of $f*$ as an anamorphism:

$$f* \;\; = \;\; [\![(f\dagger) \circ out]\!]_{(B\dagger)}$$

Functoriality of $f*$ is calculated as follows:

$$
\begin{array}{cl}
 & f* \circ g* \\
= & \quad \text{definition } * \\
 & (\!| in \circ (f\dagger) |\!) \circ (\!| in \circ (g\dagger) |\!) \\
= & \quad (29)
\end{array}
$$

$$
\begin{array}{ll}
 & (\!|in \circ (f\dagger) \circ (g\dagger)|\!) \\
= & \quad (9) \\
 & (\!|in \circ ((f \circ g)\dagger)|\!) \\
= & \quad \text{definition } * \\
 & (f \circ g)*
\end{array}
$$

Maps are shape preserving. Define $SHAPE = VOID*$ then $SHAPE \circ f* = VOID \circ f* = SHAPE$.

For cons-list $(A*, Nil \triangledown Cons) = \mu(A\dagger)$ with $A \dagger X = \mathbf{1} \mid A \| X$ and $f \dagger g = id \mid f \| g$ we get $f* = [\![f \dagger id \circ out]\!]$. From the UP for catas we find that this conforms to the usual definition of map.

$$
\begin{array}{rcl}
f* \circ Nil & = & Nil \\
f* \circ Cons & = & Cons \circ f \| f*
\end{array}
$$

Other important laws for maps are *factorization* [26] and *promotion* [4].

$$(\!|\varphi|\!) \circ f* = (\!|\varphi \circ (f\dagger)|\!) \tag{34}$$
$$f* \circ [\![\psi]\!] = [\![(f\dagger) \circ \psi]\!] \tag{35}$$

$$(\!|\varphi|\!) \circ f* = g \circ (\!|\chi|\!) \Leftarrow g \circ \chi = \varphi \circ f \dagger g \ \wedge \ g \text{ strict} \tag{36}$$
$$f* \circ [\![\psi]\!] = [\![\xi]\!] \circ g \Leftarrow \xi \circ g = f \dagger g \circ \psi \tag{37}$$

Now we know that $*$ is a functor, we can recognize that $in \in \imath\dagger* \dot{\to} *$ and $out \in * \dot{\to} \imath\dagger*$ are natural transformations.

$$
\begin{array}{rcl}
f* \circ in & = & in \circ f \dagger f* \\
out \circ f* & = & f \dagger f* \circ out
\end{array}
$$

Iterate promotion

Recall the function $iterate\ f = [\![\iota \circ id \vartriangle f]\!]$, the following law turns an $\mathcal{O}(\backslash^{\in})$ algorithm into an $\mathcal{O}(\backslash)$ algorithm, under the assumption that evaluating $g \circ f^n$ takes n steps.

$$g* \circ iterate\ f = iterate\ h \circ g \ \Leftarrow \ g \circ f = h \circ g \tag{38}$$

Law (38) is an immediate consequence of the promotion law for anamorphisms (37).

Interestingly we may also define $iterate$ as a cyclic list:

$$iterate\ f\ x = \mu(\lambda xs.Cons\ (x, f*xs))$$

and use fixed point fusion to prove (38).

Map-Reduce factorization

A data type $(A*, in) = \mu(A\dagger)$ with $A \dagger X = A \mid X\text{F}$ is called a *free* F-type over A. For a free type we can always write *strict* catas $(\!|\psi|\!)$ as $(\!|f \triangledown \varphi|\!)$ by taking $f = \psi \circ \grave{\imath}$ and $\varphi = \psi \circ \acute{\imath}$. For $f*$ we get

$$\begin{aligned} f* &= (\!|in \circ f \mid id|\!) \\ &= (\!|tau \mid join \circ f \mid id|\!) \\ &= (\!|tau \circ f \triangledown join|\!) \end{aligned}$$

where $tau = in \circ \grave{\imath}$ and $join = in \circ \acute{\imath}$.

If we define the *reduction* with φ as

$$\varphi/ \ = \ (\!|id \triangledown \varphi|\!) \tag{39}$$

the factorization law (34) shows that catamorphisms on a free type can be factored into a map followed by a reduce.

$$\begin{aligned} & (\!|f \triangledown \varphi|\!) \\ = \ & (\!|id \triangledown \varphi \circ f \mid id|\!) \\ = \ & (\!|id \triangledown \varphi|\!) \circ f* \\ = \ & \varphi/ \circ f* \end{aligned}$$

The fact that tau and $join$ are natural transformations give evaluation rules for $f*$ and $\varphi/$ on free types.

$$\begin{aligned} f* \circ tau &= tau \circ f & \varphi/ \circ tau &= id \\ f* \circ join &= join \circ f*\text{F} & \varphi/ \circ join &= \varphi \circ (\varphi/)\text{F} \end{aligned}$$

Early Squiggol was based completely on map-reduce factorization. Some of these laws from the good old days; *reduce promotion* and *map promotion*.

$$\begin{aligned} \varphi/ \circ join/ &= \varphi/ \circ (\varphi/)* \\ f* \circ join/ &= join/ \circ f** \end{aligned}$$

Monads

Any free type gives rise to a monad [17], in the above notation, $(*, tau \in \text{I} \stackrel{\cdot}{\rightarrow} *, join/ \in ** \stackrel{\cdot}{\rightarrow} *)$ since:

$$\begin{aligned} join/ \circ tau &= id \\ join/ \circ tau* &= id \\ join/ \circ join/ &= join/ \circ join/* \end{aligned}$$

Wadler [29] gives a thorough discussion on the concepts of monads and their use in functional programming.

6 Conclusion

We have considered various patterns of recursive definitions, and have presented a lot of laws that hold for the functions so defined. Although we have illustrated the laws and the recursion operators with examples, the usefulness for practical program calculation might not be evident to every reader. Unfortunately we have not enough space here to give more elaborate examples.

There are more aspects to program calculation than just a series of combining forms (like $(\!|\_|\!)$, $[\!(\_)\!]$, $\{\!|\_|\!\}$, $[\![\_,\_]\!]$) and laws about them. For calculating large programs one certainly needs high level algorithmic theorems. The work reported here provides the necessary tools to develop such theorems. For the theory of lists Bird [3] has started to do so, and with success.

Another aspect of program calculation is machine assistance. Our experience —including that of our colleagues— shows that the size of formal manipulations is much greater than in most textbooks of mathematics; it may well be comparable in size to "computer algebra" as done in systems like MACSYMA, Maple, Mathematica etc. Fortunately, it also appears that most manipulations are easily automated and, moreover, that quite a few equalities depend on natural transformations. Thus in several cases type checking alone suffices. Clearly machine assistance is fruitful and does not seem to be too difficult.

Finally we observe that category theory has provided several notions and concepts that were indispensable to get a clean and smooth theory; for example, the notions of functor and natural transformation. (While reading this paper, a category theorist may recognize several other notions that we silently used). Without doubt there is much more categorical knowledge that can be useful for program calculation; we are just at the beginning of an exciting development.

Acknowledgements Many of the results presented here have for the case SET already appeared in numerous notes of the STOP Algorithmics Club featuring among others Roland Backhouse, Johan Jeuring, Doaitse Swierstra, Lambert Meertens, Nico Verwer and Jaap van der Woude. Graham Hutton provided many useful remarks on draft versions of this paper.

References

[1] Roland Backhouse, Jaap van der Woude, Ed Voermans, and Grant Malcolm. A relational theory of types. Technical Report ??, TUE, 1991.

[2] Rudolf Berghammer. On the use of composition in transformational programming. Technical Report TUM-I8512, TU München, 1985.

[3] R. 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, October 1986.

[4] Richard Bird. Constructive functional programming. In M. Broy, editor, *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, NATO Advanced Science Institute Series. Springer Verlag, 1989.

[5] Richard Bird and Phil Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

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

[7] Manfred Broy. *Transformation parallel ablaufender Programme*. PhD thesis, TU München, München, 1980.

[8] A. de Bruin and E.P. de Vink. Retractions in comparing Prolog semantics. In *Computer Science in the Netherlands 1989*, pages 71–90. SION, 1989.

[9] Peter de Bruin. Naturalness of polymorphism. Technical Report CS 8916, RUG, 1989.

[10] Maarten Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4), 1989.

[11] Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. Translating attribute grammars into catamorphisms. *The Squiggolist*, 2(1), 1991.

[12] Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report 91-4, CWI, 1991.

[13] C. Gunter, P. Mosses, and D. Scott. Semantic domains and denotational semantics. In *Marktoberdorf International Summer school on Logic, Algebra and Computation*, 1989. to appear in: Handbook of Theoretical Computer Science, North Holland.

[14] Tasuya Hagino. Codatatypes in ML. *Journal of Symbolic Computation*, 8:629–650, 1989.

[15] J.Arsac and Y Kodratoff. Some techniques for recursion removal. *ACM Toplas*, 4(2):295–322, 1982.

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

[17] Grant Malcolm. *Algebraic Types and Program Transformation*. PhD thesis, University of Groningen, The Netherlands, 1990.

[18] Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[19] Lambert Meertens. Paramorphisms. To appear in Formal Aspects of Computing, 1990.

[20] John-Jules Ch. Meyer. *Programming calculi based on fixed point transformations: semantics and applications*. PhD thesis, Vrije Universiteit, Amsterdam, 1985.

[21] Ross Paterson. *Reasoning about Functional Programs*. PhD thesis, University of Queensland, Brisbane, 1988.

[22] John C. Reynolds. Types abstraction and parametric polymorphism. In *Information Processing '83*. North Holland, 1983.

[23] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[24] 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.

[25] Joseph E. Stoy. *Denotational Semantics, The Scott-Strachey Approach to Programming Language Theory*. The MIT press, 1977.

[26] Nico Verwer. Homomorphisms, factorisation and promotion. *The Squiggolist*, 1(3), 1990. Also technical report RUU-CS-90-5, Utrecht University, 1990.

[27] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. Technical Report 34, Programming Methodology Group, University of Göteborg and Chalmers University of Technology, March 1987.

[28] Philip Wadler. Theorems for free ! In *Proc. 1989 ACM Conference on Lisp and Functional Programming*, pages 347–359, 1989.

[29] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990.

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

[31] Hans Zierer. Programmierung mit funktionsobjecten: Konstruktive erzeugung semantische bereiche und anwendung auf die partielle auswertung. Technical Report TUM-I8803, TU München, 1988.