Proceedings of the IASTED International Conference
Advances in Computer Science (ACS 2013)
April 10 - 12, 2013 Phuket, Thailand

RECURSION PATTERNS AND THEIR IMPACT ON PROGRAMMING LANGUAGE DESIGN

Paul Bailes
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072 Australia
paul@itee.uq.edu.au

ABSTRACT
Applicative/functional programming is considerably simplified through the use of specific recursion patterns as opposed to general recursion. The viability of catamorphic recursion patterns as a pragmatic and theoretical basis makes it feasible to consider a programming language based on them. Syntactic considerations focus on notation to clarify the structures involved in the use of catamorphisms in this critical role. More fundamental semantic considerations involve the recognition that a catamorphic programming style essentially involves the treatment of data exclusively as functions and the extension of this approach to the derivation of other types of data whose behaviours are not generally catamorphic. Implementation by preprocessing into Haskell can be structured to avoid the limitations of Haskell's types, but a general regime of dynamic types seems unavoidable as an alternative. Clear connections with other work on subrecursive programming illuminates other paths for further development.

KEY WORDS
Catamorphism; foldr; functional programming; recursion pattern.

1. Introduction

Recursion patterns are a well-established means of packaging iteration in an applicative/functional context [1], and offer programmers opportunities for greater convenience and indeed reliability over general recursion. For example, formal derivation and/or verification of functional programs can exploit logical laws pertaining to recursion patterns [2] that allow us to avoid the details and complexities of otherwise explicit induction just as the patterns allow us to avoid the details and complexities of general recursion.

We take the paradigm of functional programming as implicit to our investigation, in view of what will emerge as the necessity for the higher-order abstraction capabilities of functional languages to realize the potential of recursion patterns. However, just as functional programming continues more or less directly to impact upon the "mainstream" of language design (e.g. [3]), so should the impact of the results reported here.

In that context, the purpose of this paper is to expose the implications for language design of the strict exclusion of general recursion in favour of a minimal but expressive basic recursion pattern. In particular, an extended view of the capabilities of recursion patterns forms the basis for a comprehensive replacement of symbolic data (and the recursive functions that are needed to interpret them) with functional (or "zoetic") counterparts where the representation directly and inherently embodies these desired behaviours.

The resulting language (known as "Zoe") which is the basis of the examples below is reminiscent of but significantly different to Haskell [4] in ways that will be explained.

2. Programming with Recursion Patterns

A general pattern - catamorphism - emerges as *the* basis for recursion-pattern-based programming.

2.1 From Foldr to Catamorphisms

The recursion pattern with perhaps the longest pedigree is that known variously as (in the context of lists) "reduce", "foldr" or (in its generalization to any regular recursive datatype) "catamorphism" [5]. Thus for the following example types

```
TYPE bool = true | false;
TYPE nat = succ nat |zero;
TYPE list = cons t list | nil;
TYPE bintree
   = null | leaf t | branch bintree bintree;
```

we have corresponding catamorphisms

```
cataB true t f = t;
cataB false t f = f;

cataN (succ n) f x = f (cataN n f x);
cataN zero f x = x;

cataL (cons x xs) o b = o x (cataL xs o b);
cataL nil o b = b;
```

```
    cataBt null n l b = n;
    cataBt (leaf x) n l b = l x;
    cataBt (branch b1 b2) n l b =
       b (cataBt b1 n l b)(cataBt b2 n l b);
```

The formal relationship between a recursive type and its corresponding catamorphism is defined in categorical terms, however a pragmatic statement of the relationship is:

- as well as the object to which it is applied (variously the boolean, natural, list or binary tree in the above), the catamorphism for each type takes a series of handlers, one for each different shape of the type (as characterized by the different constructor functions, e.g. null vs leaf vs branch for bintree);
- thus the definition of the catamorphism involves branching on the different shapes and applying the relevant handler (e.g. respectively n vs l vs b);
- if the relevant shape has no content (e.g. null for bintree), the handler (e.g. n) is simply the value that the catamorphism returns in this case;
- if the relevant shape has simple (non-recursive) content (e.g. leaf x for bintree), the handler (e.g. l) is a function that the catamorphism applies to the content (e.g. x);
- if the relevant shape has recursive content (e.g. branch b1 b2 for bintree), the handler (e.g. b) is a function that the catamorphism applies to combine the results of the recursive application of the catamorphism to the content (e.g. cataBt b2 n l b);
- if the relevant shape has mixed content (e.g. cons x xs for list), the handler (e.g. o) is a function that the catamorphism applies to combine the non-recursive content (e.g. x) as well as the result of the recursive application of the catamorphism to the recursive content (e.g. cataL xs o b).

The relationship between the common foldr and catamorphisms as above is established by comparing the definition of cataL with the usual definition of foldr

```
    foldr o b [] = b
    foldr o b (x:xs) = o x (foldr o b xs)
```

or by the corresponding identity

```
    foldr o b xs = cataL xs o b
```

It will emerge below just how important to our overall development is this reordering of the usual arguments to foldr.

Note that strictly speaking a distinction should be made between catamorphic patterns cataB, cataL/foldr, cataBt above vs. an actual catamorphism e.g specific operation on a list resulting from instantiation of cataL/foldr with its o and b arguments. We shall however generally neglect the distinction, relying usually on context to distinguish instances from the patterns.

2.2 Catamorphisms are Useful

Catamorphisms are so useful that they must be part of any scheme of support for recursion patterns. Consider how they simplify the definition of a host of functions that would otherwise require explicit recursion definition, e.g. (using familiar notation for basic operations, i.e. '0' for zero, ':' for cons , '[ ]' for nil, [X1, …, Xn] for an n-element list, etc.):

```
    length ys = cataL ys (\x xs -> 1+xs) 0;

    sum ys = cataL ys (\x xs -> x+xs) 0;

    append ys zs = cataL ys (\x xs -> x:xs) zs;

    nodecount bt =
       cataBt bt 0 (\x -> 1) (\bl br -> bl + br);

    flatten bt =
       cataBt bt
       []
       (\x->[x])
       (\bl br -> append bl br);
```

2.3 Other Recursion Patterns

The characteristic "higher-order" capability of functional languages allows the definition of other patterns based on catamorphisms, e.g.

```
    map f ys = cataL ys (\x xs -> f x : xs) [];
```

There are however useful patterns that are not obviously compatible with catamorphisms. For example the following:

2.3.1 Left Fold

By comparison with list catamorphism/right-fold/foldr, foldl applies its binary operator argument 'o' to list elements successively from the left, i.e.

```
    foldl o b [] = b
    foldl o b (x:xs) = foldl o (o b x) xs
```

A typical application of left fold is to reverse a list, i.e.

```
    reverse ys = foldl (\ b' x -> x:b') [] ys
```

By comparison, a straightforward catamorphic definition of 'reverse' involves multiple (inefficient) appending of elements to the tail of a list:

```
    reverse ys =
       cataL ys (\x xs' -> append xs' [x])[]
```

2.3.2 List Paramorphism

By comparison with list catamorphism, paramorphism in addition makes the tail of the list available to its binary argument ‘o’ as well as the head and the result of recursive processing of the tail, i.e.

```
paraL [] o b = b;
paraL (x:xs) o b = o x (paraL xs o b) xs;
```

A typical application of list paramorphisms is to insert an element into position in an ascending-sorted list, i.e.

```
insert e ys =
   paraL ys
   (\x exs oxs ->
      IF e<x THEN e:x:oxs ELSE x:exs
   )
   [e];
```

In the absence of the ‘oxs’ operand, the case of e<x can’t be programmed.

2.3.3 Generalisation to Other Types

Just as catamorphisms exist for types other than lists, these other patterns exist for other recursive types. The conclusion is that a recursion-pattern based approach to programming needs to be able to support a multiplicity of recursion patterns.

2.4 Catamorphisms are Effective as Basic Constructs

The challenge posed to us by the foregoing conclusion is that we need to be able to define new recursion patterns as may be required, but without recourse to general recursion which would vitiate our entire approach. It’s therefore most significant that catamorphisms alone serve as an effective basis for functional programming including the definition of other patterns.

2.4.1 Adequacy of Catamorphism, Specifically

A variety of programming devices [6] allow superficially non-catamorphic functions to be expressed as catamorphisms.

Tupling: The first-class treatment of data in functional languages removes any obstacle to modifying the catamorphism result to return a tuple of values, from which the desired alternative can be selected when the control context clarifies. For example,

```
insert e ys =
   fst $ cataL ys
   (\x (exs, oxs) ->
      (IF e < x THEN e:x:oxs ELSE x:exs
      ,
      x:oxs)
   )
   ([e], []);
```

The catamorphism result is a 2-tuple combining (a) the result that would be correct if e were to be inserted at this point, with (b) the unmodified tail of the list that can be used if it emerges that e needed to be inserted earlier. The latter device is necessary because the list tail is not by default accessible to the operands of the catamorphism. Note that at the top level first element of the tuple is the one that is guaranteed to contain ‘e’ in its correct position and this must be selected as the overall result.

Functionalisation: The facility of functional languages for functions to return functions is exploited by parameterising the catamorphism result when there is an insufficiency of data in the computation, but which can be supplied from the context. For example,

```
reverse ys =
   cataL ys
   (\x xs -> (\r -> xs (x:r)))
   (\r -> r)
   [];
```

The result of the catamorphism, overall and of course at each recursive step, is a function that takes an argument ‘r’ which denotes the list to which the reverse of the current list is to prepended.

2.4.2 Adequacy of Catamorphisms, Generally

The higher-order abstraction facility of functional languages (for functions to take other functions are parameters) is exploited to transform the above specific examples into definitions of general non-catamorphic recursion patterns. For example:

Left fold: abstracts from the functionalised definition of ‘reverse’ above by replacing the specific use of ‘:’ with generic op:

```
foldl op b ys =
   cataL ys
   (\x xs -> (\c -> xs (op x c)))
   (\c -> c)
   b;
```

Paramorphism: abstracts from the tupled definition of ‘insert’ above by replacing with a generic ternary op the testing and branching involving x, xs and oxs.

```
paraXs ys op b =
   fst $ cataL ys
   (\x (xs, oxs) -> (op x xs oxs ,x:oxs))
   (b, []);
```

2.4.3 Expressiveness of Catamorphisms

Complementing the pragmatic expressiveness of catamorphisms as exposed above, is their great theoretical expressive power. It can be demonstrated [2] that catamorphisms in the context of higher-order functions (as applies here) are capable of expressiveness greater than primitive recursion e.g. can express even Ackerman's function. A functional language based purely on catamorphisms is not Turing-complete, but practically speaking any computation that doesn't involve writing a general programming language interpreter is possible.

3. Syntactic Implications: New Presentation of Catamorphisms

From the above extensive examples of catamorphisms, we can discern a number of drawbacks in their presentation.

- *Identifying catamorphism parameters:* the association of the handlers pertaining to actual catamorphism instances with the corresponding constructors involves a confusing combination of positional notation and formal parameter names that are only accidentally related to constructor names.
- *Identifying catamorphism results:* the structure of a complex result (by virtue of tupling and/or functionalization) needs to be redefined for each handler, inviting inconsistency and error.
- *Overall:* for the basic building block of a new programming style (or equivalently, of a new programming language designed to support such a style), pure applicative notation does not provide the kind of self-documentation that well-designed notation inculcates.

These issues may not perturb some experienced functional programmers, but if recursion-pattern-based programming were to have the potential for wide appeal, including for teaching, then they would need to be addressed. Accordingly, we propose concrete syntax for catamorphism applications/instances that remedies the above problems. The following examples introduce and explain.

3.1 Simple Catamorphism

Consider for example the conventional catamorphic definition of a function to sum the elements of a binary tree of numbers:

```
sumtree bt =
   cataBt bt 0 (\x -> x) (\bl br -> bl + br)
```

Despite the admirable brevity, this rendition has the drawback of the conceptual overload resulting from the unclear association of function formal with actual parameters that arises from the use of positional notation:

- it's not clear which of the handler operands of cataBt is associated with which of the alternative constructors of bintree;
- as a consequence of the above, it's not clear how the operands of the handlers relate to operands of the corresponding constructors.

By way of remedy, we propose (in terms of the above example) a new concrete syntax for simple catamorphism application:

```
sumtree bt =
   FOLD bt
   CASE null -> 0
   CASE leaf x -> x
   CASE branch bl br -> bl + br
   END;
```

In summary:

a) we identify the handlers for each shape of tree as a CASE associated with the relevant constructor;
b) the formal parameter of the handler (e.g. 'x' of leaf) is identified by pattern matching on the corresponding constructor;
c) in the case of a recursive structure, the relevant parameters naturally refer to the results of the catamorphism on the substructures (e.g. 'bl' and 'br' or branch);
d) we delimit the entire construct with FOLD … END "brackets", deliberately evoking the customary synonym for catamorphisms.

A measure of the suitability of this kind of notation is the appearance of similar forms in similar contexts, e.g. polytypic programming [7] in which an emphasis of generic catamorphisms highlights the need for improved notation, and an embedded extension of Haskell for attribute grammars [8].

Granted, our new form is more verbose than simple function application of catamorphisms, but we contend that is a fair price to pay for comprehensibility. Note that the original applicative form remains an option.

3.2 Tupled Catamorphism

Consider next the above example of inserting an element into position in a sorted list, or the expression of paramorphism in general, additional conceptual load is imposed by:

- the recursion result being a tuple in which the distinction between components is again signified only by position;
- the need to select an element of the tuple as the ultimate top-level result.

The proposed new concrete syntax extends the simple case above in response to this burden, e.g.

```
insert e ys =
   FOLD ys INTO {exs, oxs}
   CASE [] -> {exs <- [e], oxs <- []}
   CASE x:xs -> {
      exs <-
         IF e < x THEN e:x:oxs ELSE x:exs,
      oxs <- x:oxs
```

```
        }
        GIVES exs
        END;
```

In the above example we add to the simple case:

a) a local named tuple (or "record") type that identifies the components exs, oxs of the catamorphism tuple result ("INTO" which the "FOLD" results) and which are available for use inside the handler bodies as needed;
b) construction of a record by pseudo-assignment to its fields
c) a concluding projection that "GIVES" the required top-level tuple component exs.

3.3 Functionalised Catamorphism

Consider finally example of replacement of the elements of a list by the index of their positions (another example of a left fold in general):

```
    idxL ys =
        cataL ys
        (\x xs -> (\c -> c : xs(c+1)))
        (\c -> [c])
        0;
```

The conceptual burden in this case is:

- primarily the need to provide separately the same parameter for the functionalized results of the o and b arguments to cataL;
- but also a common initial value (actual parameter) separate from the parameter definitions.

The appropriate concrete syntactic development is e.g.

```
    idxL ys =
        FOLD ys
        CASE [] -> [c]
        CASE x:xs -> c:xs(c+1)
        WHERE c = 0
        END;
```

In the above example:

a) we add to the simple case a context parameter (e.g. 'c') that is common to the result of all the handlers
b) we supply an initial value to which the top-level functionalized catamorphism result is applied;

Note the use of pattern-matching using the conventional built-in notation for list constructors which is available in all the above forms.

4. Semantic Implications: Zoetic Data

Beyond the syntactic requirements presented above, there are actually profound semantic implications of a strictly catamorphism-based approach to programming and language design. They concern how symbolic data may be uniformly replaced by functional representations that capture behaviours inherent to the data as if they were in a sense alive, rather than inert symbols (hence "zoetic").

Our basic proposition remains that catamorphisms are to be *the* canonical means for processing data. It thus follows that the only access to data structures, once constructed, is via catamorphisms. That is, all methods must be able to be expressible as catamorphisms. Therefore, data structures might as well support a single method - the catamorphism pertaining to their datatype.

The significant consequence is that a data structure might as well behave as its own catamorphism/catamorphic pattern/right fold. To give a basic implementation of this consequence, it suffices to replace data by the partial application to them of the relevant catamorphism, e.g.

```
    -- zoetic counterparts of symbolic booleans B
    zb = cataB B;

    -- zoetic counterparts of symbolic naturals N
    zn = cataN N;

    -- zoetic counterparts of symbolic lists Xs
    zxs = cataL Xs;

    -- zoetic counterparts of symbolic binary
    -- trees T
    zt = cataBt T;
```

4.1 Direct Generation of Zoetic Data

Before however giving examples of how zoetic data are applied, it is valuable to elaborate the supporting semantic framework of zoetic data. If data (structures) are to have functional (catamorphic) behaviours, then the functions that build data need to generate functions rather than construct symbols. Fortunately these generators can be derived by simple expansion of the partial applications that characterize zoetic data. For example, for lists (distinguishing in this part of the presentation only symbolic data and their constructors in *underlined italics* from zoetic data and their generators in **bold**):

nil o b = cataL *nil* o b = b

cons x **xs** o b
= cataL (*cons* x *xs*) o b
= o x (cataL *xs* o b)
= o x (**xs** o b)

(noting in particular that zoetic **xs** is the symbolic partial application "cataL *xs*"). It seems obvious that this kind of transformation can be automated.

The corresponding zoetic generators for the other example types above are:

```
    -- zoetic booleans
    true t f = t;
```

```
false t f = f;

-- zoetic naturals
zero s z = z;
succ n s z = s (n s z);

-- zoetic binary trees
null n l b = n;
leaf x n l b = l x;
branch b1 b2 n l b = b (b1 n l b) (b2 n l b);
```

The relationship to the definitions of the corresponding catamorphisms (cataB, cataN, cataBt) is apparent. Datatype declarations (e.g. for bool, nat, list bintree far above) can now be thought of as defining zoetic generators rather than symbolic constructors. Generator names simply take the place of constructor names in "FOLD …" syntax - see the earlier examples which continue to apply.

4.2 Sub-catamorphic Zoetic Data

The idea thus far that zoetic data can be thought of as the partial application of a canonical method (the catamorphism for the datatype) to the counterpart symbolic data can be generalised. What if the canonical method for a datatype is not the catamorphism, but of many other conceivable methods? Such a scenario integrates neatly into our framework as follows.

Consider the datatype of (zoetic) sets generated by the following:

- empty: empty set
- single x: set of single element x
- union s1 s2: union of sets s1 and s2

The next step in the specification of the zoetic datatype is the required characteristic behavior, which for these sets we choose to be as characteristic predicates. In other words, the (partial) application of a "member" method to some structure representing the sets. In our case, binary trees (bintree) serve the purpose, thus

```
empty e = member null e;
single x e = member (leaf x) e;
union s1 s2 e = member (branch s1 s2) e;
```

Derivation of implementations for the generators depends upon how we specify member, but in this case the result is straightforward:

```
empty e = FALSE;
single x e = x==e;
union s1 s2 e = s1 e OR s2 e;
```

The essential connection between "sub-catamorphic" zoetic types (such as sets, where the characteristic method is definable in terms of a catamorphism) with the basic (catamorphic) zoetic types (such as bintree) derives from the identity property of catamorphisms, which in our zoetic setting may be expressed as follows. Given an instance cval of some catamorphic zoetic type with generators cg1 … cgn, then application of the zoetic instance to the generators (it will be recalled that each of a catamorphism's operands is a handler for each different kind of structure within its associated dataype) in effect regenerates the instance (where ⬄ denotes semantic equivalence of expressions):

```
cval ⬄ cval cg1 ... cgn
```

For example

```
nil ⬄ nil cons nil
cons zx zxs ⬄ cons zx zxs cons nil
```

In brief, the identity property holds because the effect of a catamorphism is to apply its operands in place of generators. If the operands *are* the catamorphic generators, they are re-applied to reconstruct the (zoetic) catamorphic data. Application to the symbolic data constructors will yield the underlying symbolic data, a situation summarized by the following:

```
cons x1(cons x2 nil) (:) []
⬄
x1:x2:[]
⬄
[x1,x2]
```

Sub-catamorphic generators are no exception. Thus, for an instance sval of a corresponding sub-catamorphic type (such as sets corresponding to bintrees) with generators sg1 … sgn: sval ⬄ cval sg1 ... sgn

For example

```
union
(union (single x1) empty)
(union (single x2) (single x3))
⬄
branch
(branch (leaf x1) null)
(branch (leaf x2) (leaf x3))
empty single union
```

That is, instances of catamorphic types can be transformed into sub-catamorphic instances by application to the sub-catamorphic generators. This has the potential to be the basis of an interesting notion of polymorphism in this broad programming style.

A little more prosaically, the definition of sub-catamorphic types and their generators can usefully derive from the underlying catamorphic type using a variant of our new syntax for catamorphisms. E.g. for sets from trees:

```
TYPE sets FROM bintree
CASE null -> empty = FALSE
CASE leaf x -> single = x==e
CASE branch b1 b2 -> union = b1 e OR b2 e
```

```
WITH e
END;
```

A bintree bt can then be refined to the corresponding set by

```
bt TO sets
```

which applies bt to the appropriate generators for sets as determined by the TYPE … FROM declaration.

On the subject of methods other than catamorphisms, a further similar notation can be used to define a distinguished "SHOW" method for catamorphic types

```
SHOW bintree
CASE null -> ""
CASE leaf x -> "(" ++ SHOW x ++ ")"
CASE branch b1 b2 ->
   "(" ++ SHOW b1 ++ "," ++ SHOW b2 ++ ")"
END;
```

4.3 Totally-Functional Programming

Taken to the limit, zoetic data lead to a programming style where explicit recursion and symbolic data can be entirely foregone, hence the style is "totally functional". Consider for example, Church numerals [9] which arise from the zoetic generators succ, zero above:

```
one = succ zero = (\f x -> f x)
two = succ one = (\f x -> f (f x))
-- etc ...
```

As an example of how the TFP approach has the potential greatly to simplify programming, consider how the zoetic representation for naturals permits the following direct definitions of basic arithmetic operators

```
add a b = a succ b
mul a b = a (add b) zero
exp a b = b (mul a) one
```

Observe the lack of recursion and branching that is the source of much potential effort and error in programming.
Our hypothesis is that corresponding simplifications exist for more complex types. A signal example of a sub-catamorphic datatype is given by combinator parsers [10]. Space limitations preclude a detailed treatment here, but suffice it to say that the key insight is to replace a symbolic grammar that needs to be interpreted by some kind of parsing engine, by its zoetic counterpart: a parser itself. The essential zoetic generators are as usual higher-order counterparts of the context-free compositions of alternation and concatenation, plus token-matching:

```
-- parser for the alternation of languages
-- recognized by parsers p1, p2
alt p1 p2 s = p1 s ++ p2 s

-- parser for the concatenation of languages
-- recognized by parsers p1, p2
cat p1 p2 s = concat (map p2 (p1 s))

-- parser for token string t
tok t s = residue in s after t, else []
```

(where concat and map are as in Haskell). Note that a parser so defined is inherently nondeterministic, returning the list of possible residues after matching a given string s, and thus [ ] in the case of failure.

Combinator parsers can be thought of as a sub-catamorphic version of

```
TYPE dbtree =
   nl string              -- tok
   | b1 dbtree dbtree     -- cat
   | b2 dbtree dbtree;    -- alt
```

4.4 Built-in Zoetic Data

The principle that leads us to data structures with functional representations and behaviours (catamorphic or sub-catamorphic) leads to the conclusions that inbuilt data type also need to be zoetic, i.e. have functional behaviours for example as follows.

4.4.1 Booleans

Conditional expressions

```
IF exp1 THEN exp2 ELSE exp2
```

are basically syntactic sugar for the application

```
exp1 exp2 exp3
```

where built-in truth values are defined as if zoetic generators:

```
TRUE x y = x;
FALSE x y = y;
```

Note that the same effect could be achieved using our catamorphism notation:

```
FOLD exp1
CASE TRUE -> exp2
CASE FALSE -> exp3
END
```

but the traditional notation in this case is advantageous.

4.4.2 Natural Numbers

As well as being subject to arithmetic operators, natural numbers are iterators. Thus e.g. the evaluation (where ➔ denotes a transformation or rewrite rule)

```
4 func exp ➔ func (func (func (func exp)))
```

Under varying circumstances this might be more or less acceptable as an idiom for n-fold iteration than using our catamorphism notation (extended for typical pattern matching):

```
FOLD n
CASE 0 -> exp
CASE n+1 -> func n
END
```

4.4.3 Lists

The symbolic constructor behavior of [ ] and : is overridden by zoetic generator behavior, thus

```
[] func exp0 ➔ exp0

(x:xs) func exp0 ➔ func x (xs func exp0)
```

Also

```
[exp1, ..., expn] func exp0
➔
func exp1 (func ... (func expn exp0)...)
```

See earlier for list processing examples using our "FOLD" notation.

4.4.4 Strings

In order to provide a basic handle for processing input data, character strings are not lists of characters, but need to have functional behaviours akin to token acceptors in combinator parsing [10].

5. Ongoing/Further Work

A number of challenges have been at least alluded to en passant in this presentation. With respect to the development of a platform for the significant exploration of TFP, the most pressing are as follows.

5.1 Type-Checking

We have been deliberately un-specific about the nature of the type system behind our base language and its extension to support catamorphisms. The reasons for this are that zoetic data are not accepted by the Hindley-Milner type inference system [11] commonly found in functional languages, and that alternatives are problematic. For example, non-trivial applications of the above recursion-free definitions of arithmetic operations are rejected. In particular, the more powerful type-checking systems (first-class/impredicative polymorphic) that seem to be required, while remaining decidable, do not support convenient type inference. Indeed, the burden of explicit type annotation may be too great for the typical programmer [12].

Instead, a regime of dynamic types involving the selective attachment of assertions to declarations and expressions seems unavoidable. The essentials of such a system (much as earlier proposed [13]) are:

- a basic notion of datatype as the set of all applications of a distinctive constructor;
- type-checking by retracting expressions into sets, thus

```
exp :: typ
➔
IF exp IN typ THEN exp ELSE ABORT
```

- composition of types typi into mapping types typ1 → typ2 that are checked lazily, thus

```
func :: typ1 -> typ2
➔
(\x -> func (x :: typ1) :: typ2)
```

- correspondingly lazy application of type checks to data structure element, thus e.g. for

```
TYPE tlist = tcons typ tlist | tnil;
```

  the following transformation applies

```
cons x xs ➔ cons (x :: typ) (xs :: tlist)
```

- lazy type combinations for built-in types, thus

```
[exp1, ..., expn]::LIST typ
➔
[exp1::typ, ..., expn::typ]
```

and

```
(exp1, ..., expn) :: TUPLE (typ1, ..., typn)
➔
(exp1 :: typ1, ..., expn :: typn)
```

- other combinations and construction of types using set-theoretic operations.

Because types as above are first-class values, polymorphic types are in principle available as a result of the underlying unrestricted functional abstraction capability, but the most elegant rendition of same remains under investigation

5.2 Corecursion Patterns

In order to process infinite data structures, it will be necessary to accommodate the generative "corecursive" dual of catamorphisms, i.e. anamorphisms [4]. Care however needs to be taken if the total (i.e. terminating) property that results from eschewing general recursion is not violated by the unrestricted combination of cata- and anamorphisms. It seems as if the type system may need to distinguish between finite structures (data) consumed by catamorphisms and the potentially infinite structures ("codata") generated by anamorphisms.

5.3 Implementation

A prototype implementation for a subset of the language embodying these proposals and as exposed above, by preprocessing into Haskell has been developed as a proof-of-concept. The source of most complexity arises from the need to evade the Haskell type regime as motivated above, by tagging functions as data upon definition and untagging before application. What thus amounts to an interpreter is relatively trivial, as if

```
apply (tag F) X → F X
```

The need however for Haskell-side uniformity to apply this device to every source construct is tedious and means that all the features of Haskell are not automatically available.
The next stage of development is self-implementation of the preprocessor.

6. Related Work

In addition to the various items of related work acknowledged throughout the above, the following are of interest.

6.1 Turner's "Total Functional Programming"

A vision of the benefits of a comprehensive subrecursive programming style (hence total functions) has been articulated by David Turner [14, 15] in particular the complementary role of cata- and anamorphisms and the challenges posed with a satisfactory treatment of the latter. Our work presented here can be viewed as an extension of Turner's in the matters of (a) notation for catamorphisms (b) comprehensive replacement of symbolic by zoetic data.

6.2 Programming = Language Design

It should be noted at least in passing that there exists a parallel justification for eschewing general recursion and the adoption of zoetic data [16]. In brief summary:

- programming is essentially an exercise in language design by language extension;
- writing interpreters for language extensions is a poor alternative to direct definition;
- programming languages should be sufficiently-powerful to avoid the need for interpretation as opposed to direct definition, i.e. functional languages are necessary;
- in the context of the foregoing, programmers should avoid the unnecessary recourse to interpretation for the extensions they create, i.e. expressive power needed only for writing interpreters in dispensable;
- symbolic datatypes exemplify unnecessary interpretive extensions;
- instead, the expressiveness of functional languages should be exploited for zoetic data and higher-order functions for the operations thereon.

7. Conclusion

From the foregoing we contend that it's clear that catamorphism-based language design is both well-motivated and at least plausible. The resultant language needs the following distinctive characteristics at least:

- optimally if not necessarily based on the functional paradigm;
- support for catamorphisms by judicious syntactic sugaring;
- extension of the basic catamorphism-oriented syntax to support the idioms (of tupling and functionalization) that allow catamorphisms feasibly to serve as a basis for other recursion patterns;
- uniformly zoetic representations (i.e. as functions not symbols) for all inbuilt and programmer-defined datatypes;
- extension of semantic support and appropriate syntax to zoetic data (and their types) whose behaviours derive from catamorphisms but are not simply so.

More problematic areas that provide us with our future research challenges include the following as detailed above:

- type-checking;
- support for codata (e.g. infinite lists or "streams");
- robust implementation.

Acknowledgements

We are grateful to several colleagues who have contributed to the development of the above ideas over the years: Leighton Brough, Colin Kemp [17], Ian Peake and Sean Seefried; and to anonymous referees of earlier versions.

References

[1] J. Hughes, Why Functional Programming Matters, The Computer Journal, vol. 32, no. 2, 1989, 98-107.

[2] G. Hutton, A Tutorial on the Universality and Expressiveness of Fold, *J. Functional Programming, vol. 9*, 1999, 355-372.

[3] Scala, http://www.scala-lang.org/ , accessed 8 June 2012.

[4] The Haskell Programming Language, http://www.haskell.org, accessed 2 June 2012

[5] E. Meijer, M. Fokkinga and R. Paterson, Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire, *Proc. FPCA 1991, LNCS vol. 523*, 1991, 124-144.

[6] P. Bailes and L. Brough, Making Sense of Recursion Patterns, Proc. 1st International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), IEEE, 2012, 16-22.

[7] P. Jansson, J. Jeuring (1997), Polyp - A Polytypic Programming Language, *Proc. POPL, 1997*, 470-482.

[8] S.D. Swierstra, A. Alcocer and J.A.B.V Saraiva, Designing and Implementing Combinator Languages. In S.D. Swiestra, P.R. Henriques and J.N. Oliveira (eds.), *Advanced Functional Programming, Third International School, LNCS vol. 1608, 1999*, 150-206.

[9] H. Barendregt, The Lambda Calculus - Its Syntax and Semantics 2nd ed., North-Holland, Amsterdam, 1984.

[10] G. Hutton, Higher-order functions for parsing, *J. Functional Programming, vol. 2*, 1992, 323-343.

[11] R. Milner, A Theory of Type Polymorphism in Programming, *J. Comp. Syst. Scs.*, vol. 17, 1977, 348-375.

[12] D. Vytiniotis, S. Weirich and S.L.P. Jones, Boxy types: inference for higher-rank types and impredicativity, *Proc. ICFP 2006*, 2006, 251-262.

[13] P.A. Bailes and C.J.M Kemp, Integrating Runtime Assertions with Dynamic Types: Structuring Derivation From an Incomputable Specification, *Proc. COMPSAC,* 2003.

[14] D.A. Turner, Elementary Strong Functional Programming, in: R. Plasmeijer and P. Hartel (eds.), *Proc. First International Symposium on Functional Programming Languages in Education, LNCS, vol. 1022*, 1995, 1-13.

[15] D.A. Turner, Total Functional Programming, *Journal of Universal Computer Science, vol. 10*, no. 7, 2004, 751-768.

[16] P. Bailes, The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design), *Proc. ASWEC 1986.*, IEAust, Canberra, 1986, 14-18.

[17] C. Kemp, Theoretical Foundations for Practical 'Totally-Functional Programming', PhD Thesis, The University of Queensland, St Lucia, 2009.