HIGHER-ORDER CATAMORPHISMS AS BASES FOR PROGRAM STRUCTURING AND DESIGN RECOVERY

Paul Bailes, Leighton Brough, Colin Kemp
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072 Australia
paul@itee.uq.edu.au, brough@itee.uq.edu.au, ck@itee.uq.edu.au

ABSTRACT
Catamorphisms ("foldr" on lists, but generally applicable to any regular recursive datatype) are not just useful but are an effective basis for a recursion-pattern-based discipline of program design. A new presentation for catamorphisms makes it clear how they provide functional semantics for symbolic datatypes, with the capacity to expose significant variations in program design. A further development of the new presentation exploits the higher-order capabilities of functional languages. This is the key enabler for a comprehensive replacement of symbolic data and their interpreters, either implicit or explicit, with direct functional representations. These extensions, of the applicability of catamorphisms and of their presentations, make them even more attractive as bases for program structuring and design, and likewise as targets for software reverse engineering and design recovery.

KEY WORDS
Software design and development; Programming Tools and languages; Functional programming; Recursion pattern.

1 Introduction

An ongoing problem in software engineering concerns the use of appropriate structuring mechanisms or "combining forms" [1] by which program components can be assembled into more significant entities. One mechanism may be judged more appropriate than another if it is less powerful and thus less prone to misuse than another but still allows the same programs to be written. Ideally, programs will be designed and written from their origin using the optimal combining forms, but a further related problem concerns how poorly-written programs can be restructured to make explicit their at-best implicit use of the right combining forms,

In this context, the role of recursion patterns rather than the unrestricted use of general recursion as a device for the structuring of programs for better comprehensibility is well-established [1]. Of these, catamorphic patterns [2] also known (in the context of list structures) as "reduce" or "foldr" are perhaps the most prominent. While much of the literature would appear to emphasize the role of recursion patterns in program development, it should be kept in mind that transformation to more comprehensible bases of program structure is also a major goal of software reengineering and maintenance. We note that list catamorphisms have begun to be used as practical targets for source code design recovery [3, 4], with a combination of benefits in mind but at least including the goal of thereby improving source code understanding and comprehension.

In that context the purpose of this paper is broadly to add to the weight of evidence in favour of catamorphisms as bases for program structure and design, and equally as targets for program restructuring and design recovery targets, specifically:

i. to emphasise the applicability of catamorphisms to types beyond lists;
ii. to indicate that catamorphisms are effective as a basic recursion pattern, i.e. from which other patterns may be derived;
iii. to show how the applicability of catamorphisms is transformed through their application to higher-order operands.

As a result of iii above we necessarily work in a functional programming/language context [5], with Haskell [6] as the presentation vehicle. Note that any language with first-class functions qualifies as "functional" for our purposes (e.g. Scala [7]), and that much of our advocacy of catamorphisms (except iii) still applies even to non-functional contexts.

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

2 Catamorphisms are Useful

Catamorphisms represent a general, indeed generic recursion pattern on regular recursive datatypes. The design of programs is improved by expressing their recursive components as explicit instantiations of these general patterns.

2.1 List Foldr

The archetypal catamorphism is the list "foldr", rendered

in Haskell by the self-explanatory

```
foldr op b xs =
    if null xs
    then b
    else
        op (head xs) (foldr op b (tail xs))
```

We will however use pattern-matching style in the rest of this presentation, thus equivalently:

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

Many recursive definitions of list-processing functions fit the foldr pattern. For example, each of

```
sum [] = 0
sum (x:xs) = x + sum xs

append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

can be re-coded

```
sum xs = foldr (+) 0 xs
append xs ys = foldr (:) ys xs
```

2.2 General Catamorphisms

Counterparts to "foldr" can be defined for any regular recursive datatype, e.g.

```
data Nat = S Nat | Z
data Bintree a =
    Mt | Lf a | Nd (Bintree a) (Bintree a)
```

and thus

```
foldN s z Z = z
foldN s z (S n) = s (foldN s z n)

foldB m l n Mt = m
foldB m l n (Lf a) = l a
foldB m l n (Nd b1 b2) =
    n (foldB m l n b1) (foldB m l n b2)
```

Just as with lists, recursive definitions of functions on other structures often easily fit the catamorphic pattern. For example

```
flatten Mt = []
flatten (Lf a) = [a]
flatten (Nd b1 b2) =
    flatten b1 ++ flatten b2
```

is equivalently

```
flatten bt = foldB [] (:[]) (++) bt
```

(where (:[]) is Haskell for a function to make a singleton list from its operand). The foldB pattern is similarly applied to define further otherwise recursive functions on binary trees, e.g.

```
count bt = foldB 0 (const 1) (+) bt
depth bt = foldb 0 (const 1) ((+1).max) bt
```

(where max computes the maximum of two integers, (+1) adds 1 to a number and '.' is function composition).

While the formal relationship between a recursive type and its corresponding catamorphism/fold is defined in categorical terms [2], a pragmatic statement of the relationship is as follows:

- as well as the object to which the fold is applied (variously the list, Nat or Bintree 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. Mt vs Lf vs Nd for Bintree);
- thus the definition of the catamorphism involves branching on the different shapes and applying the relevant handler (e.g. respectively m vs. l vs. n);
- if the relevant shape has no content (e.g. Mt for Bintree), the handler (e.g. m) is simply the value that the catamorphism returns in this case;
- if the relevant shape has simple (non-recursive) content (e.g. Lf a for Bintree), the handler (e.g. l) is a function that the catamorphism applies to the content (e.g. a);
- if the relevant shape has recursive content (e.g. Nd b1 b2 for Bintree), the handler (e.g. n) is a function that the catamorphism applies to combine the results of the recursive application of the catamorphism to the content (e.g. "n (foldB m l n b1) (foldB m l n b2)");
- if the relevant shape has mixed content (e.g. x:xs for lists), the handler (e.g. op) 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. "foldr op b xs").

In the sense conveyed by the above, catamorphisms simply reflect the natural recursive structure of the underlying datatypes, i.e. that simply systematically replace the constructors of the given data with the functions provided as the other arguments..

2.3 Generic Catamorphism

We have space only to note in passing that modern functional languages such as Haskell are sufficiently-expressive to define generic libraries that can represent abstract catamorphisms across different types and be instantiated for each [8, 9]. For our potential design recovery purposes however, the actual catamorphisms on different types are the objects of interest.

3 Catamorphisms are Fundamental

As well as generalising "foldr" as above to catamorphisms on types other than lists, catamorphisms can be used as a basis for the expression of other useful, more complex recursion patterns. We illustrate this first with a couple of arguably the most useful key recursion patterns and subsequently we show how they can be written in terms of catamorphisms.

3.1 Left Fold

There are occasions when the right-associativity implicit in the op argument to foldr is inconvenient or inefficient. For example, to convert a list of digit values into a single value of some exponential base:

```
conv (x:xs) b = conv xs (b*base + x)
conv [] b = 0
```

The relevant, well-known left-folding recursion pattern is

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

Thus

```
conv xs base =
    foldl (\b x -> b*base+x) 0 xs
```

A further example of foldl is list reversal

```
reverse xs = foldl (\rxs x -> x:rxs) [] xs
```

which avoids the expensive repeated tail append of the corresponding foldr rendition:

```
reverse xs =
    foldr (\x rxs -> rxs ++ [x]) []
```

3.2 Paramorphism

Some conceptually simple operations don't on the surface appear to have any kind of simple foldr rendition. For example, inserting an element into position in an ascending-sorted list:

```
insert e [] = [e]
insert e (x:xs) =
    if e < x then e:x:xs else x:insert e xs
```

is not compatible with foldr because of insert's need to have access both to the result of the recursion ("insert e xs") and the original list tail xs as well.

Instead, insert as defined above is an example of a list paramorphism:

```
paraL op b [] = b
paraL op b (x:xs) = op x (paraL op b xs) xs
```

Note the key difference compared to list catamorphism in that list tail xs is included as a third argument to the op applied at each stage of the recursion. Thus:

```
insert e xs =
    paraL
    (\x rxs xs ->
        if e < x then e:x:xs
        else x : rxs
    )
    [e]
    xs
```

Observe how the op argument to paraL is basically the body of the non-empty case of insert above.

3.3 Catamorphic Design Basis

The prospect of a plenitude of recursion patterns, potentially an infinitude, would appear to vitiate the rationale implicit in the adoption of a pattern-based approach, i.e. the intellectual parsimony that results from being able to deal only with a small finite basis of concepts. Fortunately, exploitation of functional languages' defining characteristic, of first-class functions in general and function-valued functions in particular (and consequently the same for data structures), allows rendition of these other patterns in terms of catamorphisms, i.e. foldr-based definitions [10].

3.3.1 List paramorphisms as catamorphisms

Paramorphisms can be rendered in terms of catamorphisms that return the alternative possible results (as 2-tuples) back up through the list recursion to the point at which the choice between them can then be made:

```
paraL op b xs =
    fst $ foldr
    (\x (rxs, xs) -> (op x rxs xs, x:xs)
    (b, [])
    xs
```

The alternatives to be chosen between are the results (i) in which base value b has been incorporated vs. (ii) in which it hasn't. The choice point is at the return from the top-level of the recursion, i.e. the call to foldr, where it's clear that the (i) result is the desired one. Accordingly the op argument to foldr takes such a 2-tuple as its argument (rxs, xs) and returns a 2-tuple that maintains that distinction between its results.

An equivalent operational view is that in order to implement paraL, foldr reconstructs the list from the tail upwards to make it available at each stage of the recursion and then discards it at the top level.

3.3.2 Left folds as catamorphisms

Left folds can be rendered in terms of catamorphisms whose results are parameterized on further required

contextual information:

```
foldl op b xs =
    foldr
    (\x rxs -> (\c -> rxs (op c x)))
    (\c -> c)
    xs
    b
```

Observe how the top-level catamorphism result is applied to an initial context b, and how intermediate results (rxs) are applied to modified contexts

3.3.3 Generalisation to other structures

Thus, the paramorphic and fold-left patterns can be seen as developments of catamorphisms where there are respective additional details of control or data to be made explicit.

Needless to say, analogous methods are used to achieve the counterparts of left fold and paramorphism on other structures, e.g. for binary trees above

```
paraB m l n bt =
    fst $ foldB
    (m, Mt)
    (\a -> (l a, Lf a))
    (\(rb1, b1) (rb2, b2) ->
        (op rb1 b1 rb2 b2, Nd b1 b2)
    )
    bt
```

The significance of this result (the ability to render the other recursion patterns pragmatically [11] using as basis the simplest recursion pattern - foldr/catamorphisms) is that catamorphisms provide a simplest yet effective basis for the expression of (bounded) recursion/iteration.

3.3.4 Catamorphisms refine design differences

For the purposes of software design and recovery, catamorphisms thus offer a canonical representation in terms of which the differences between different instances of recursion/iteration can be isolated and highlighted, specifically as the operands of the catamorphism. Consider the above examples of how:

- different behaviours on the same lists are achieved by supplying different op and b arguments to foldr
- and different behaviours on the same binary trees are achieved by supplying different m, l and n arguments to foldB.

4 Capturing Data Semantics with Catamorphisms

A deeper level of understanding of how catamorphisms make program design explicit is accessible in terms of how they ascribe semantics to the data to which they apply, as discussed further in our related work (see 7.2 "Totally Functional" programming below.

4.1 Inverting Fold

First, it's useful to adopt a different perspective on catamorphisms, by reordering the arguments so that the data is first followed by the various handlers for the different shapes of the datatype, e.g.

```
-- foldr op b xs = cataL xs op b
cataL [] c n = n
cataL (x:xs) c n = c x (cataL xs c n)

-- foldN s z n = cataN n s z
cataN Z s z = z
cataN (S n) s z = s (cataN n s z)

-- foldB m l n bt = cataB bt m l n
cataB Mt m l n = m
cataB (Lf a) m l n = l a
cataB (Nd b1 b2) m l n =
    n (cataB b1 m l n) (cataB b2 m l n)
```

4.2 Catamorphisms as Interpreters

From this perspective, it's easy to see how catamorphisms can be thought of as interpreters that apply to (recursive) data structures. The results of these (partial) applications are the functions that take the remaining parameters of the catamorphism. In other words, catamorphisms are semantic interpreters that transform data into their meanings as functions.

For example, the partial application "cataN M" for some M::Nat assigns to M an interpretation as M-fold function composition or iteration. We can then use this interpretation to restructure what would be the conventional recursive definitions of the basic arithmetic operations on Nats, from

```
add Z b = b
add (S a) b = S (add a b)

mul Z b = Z
mul (S a) b = add b (mul a b)

pow a Z = S Z
pow a (S b) = mul a (pow a b)
```

to

```
add' a b = cataN a S b
mul' a b = cataN a (add' b) Z
pow' a b = cataN b (mul' a) (S Z)
```

Thus:

- addition of a to b is evidently the simple a-fold application of S(uccessor) to b;
- multiplication of b by a is the a-fold addition of b to Z(ero);
- the b-th power of a is the b-fold multiplication of a to unity.

4.3 Catamorphic Interpreters Expose Design Decisions

A consequent advantage of how catamorphic interpreters reveal the functional semantics of data is that possibly different semantics for data are exposed through their different interpreters.

More generally, there is actually nothing inherent in the Nat type's symbols - Z, S Z, S (S Z), etc. - that ascribes to them the properties we customarily associate with isomorphic representations such as 0, 1, 2, etc. Rather, it's the interpretation assigned to the Nat symbols in the operations that use them - implicitly in add, mul, pow and explicitly in add', mul' and pow' - that makes them behave like natural numbers. These interpetations, of the symbols, can be simply 'read off' from the catamorphism; a unique advantage to writing interpreters as catamorphisms.

5 Zoetic Data

The arrangement reached at the end of the previous section is still not ideal. Granted, we have been able to make explicit how programs are structured by the realization of the applicative semantics of data through catamorphic interpreters. However, the need for repeated construction of the same interpretation (e.g. the several applications of cataN to Nats a, b in the arithmetic examples above) represents a conceptual inefficiency. From a program development point of view, the duplication represents an opportunity for error. From a program comprehension/design recovery point of view, it represents a failure to give common identification of multiple occurrences of the same concept.

We now show how data and their catamorphic interpreters (cataN etc) can be systematically replaced in programs, removing the repeated applications and conceptual inefficiency

5.1 Catamorphic Functional Representations of Data

The key concept in rectifying this imperfect state of affairs is to recognize the significance of partial applications of catamorphic patterns cataN, cataL, cataB etc. to their relevant symbolic data. These enlivened or "zoetic" data are refinements of concrete symbolic data into an abstract domain of functional behaviours as catamorphisms. Thus:

- instead of symbolic Z(ero) we deal with its zoetic counterpart cataN Z;
- for unity, instead of S(uccessor) Z(ero) we deal with cataN (S Z), etc. for all the naturals
- similarly replace lists L by cataL L
- replace binary trees T by cataB T
- etc.

As a result, programming becomes further simplified. For example, the definitions of the arithmetic operators evolve now to definitions on zoetic naturals za, zb:

```
zadd za zb = za zsuc b
zmul za zb = za (zadd zb) (cataL Z)
zpow za zb = zb (zmul za) (cataL (S Z))
```

Observe how the functional (catamorphic) semantics of natural numbers (n-fold composition) are now inherent to the zoetic naturals, requiring no need for separate multiple applications of the interpreting catamorphism.

5.2 Zoetic Generators

We are yet to match the simplicity of programming with zoetic data on the one hand with their definition on the other. Accordingly, rather than building symbolic data then interpreting into functions by partial application of catamorphic patterns (e.g. cataN Z, cataN (S Z) in the above), we want to build directly the functional representations with the required catamorphic behaviours. For zoetic data, the conventional constructors for symbolic datatypes will be replaced by generators which build the partial applications of the relevant catamorphism.

Thus e.g. for lists, we specify:

```
zcons x (cataL xs) = cataL (x:xs)
znil = cataL []
```

noting how the list operand to zcons is made zoetic by partial application of its catamorphism cataL.

Implementations for zcons and znil follow by simple calculation:

```
znil op b = cataL [] op b = b

zcons x (cataL xs) op b
    = cataL (x xs) op b
    = op x (cataL xs op b)
    = op x (zxs op b) where zxs = cataL xs
```

Thus leading to Haskell declarations:

```
znil op b = b
zcons x zxs op b = op x (zxs op b)
```

Zoetic constructors for other types follow similarly, e.g.:

```
zero s z = z          -- zoetic naturals
zsuc n s z = s (n s z)

zmt m l n = m         -- zoetic binary trees
zlf a m l n = l a
znd zb1 zb2 m l n =
    n (zb1 m l n) (zb2 m l n)
```

5.3 Program Design with Zoetic data

From a program design and design recovery point of

view, information about the catamorphic behaviour of zoetic data is now consolidated into single occurrences of definitions of their generators (i.e. zoetic counterparts of symbolic data constructors).

For example whereas before in the definitions of add', mul' and pow'

```
add' a b = cataN a S b
-- etc…
```

we needed explicitly to signify the application of catamorphic behavior to naturals a and b, now the behavior is implicit in zoetic naturals za and zb and derives from a single source: the definition of generators zero and zsuc. Of course, grouping of these related generator definitions would be essential to ensure the desired level of ease of comprehension.

Accordingly, the terms in which we express data make direct use of generators instead of partial applications of catamorphisms, e.g.

```
zmul za zb = za (zadd zb) zero
zpow za zb = zb (zmul za) (szuc zero)
```

6 Sub-Catamorphic Behaviours

While partial application of cataN etc provide the default catamorphic behaviours for zoetic refinements of datatypes, other characteristic behaviours are conceivable, but still definable as catamorphisms, that is under the umbrella of same hence our term "sub-catamorphic".

6.1 Sub-Catamorphic Interpreters

Consider for example using binary trees to represent sets in terms of membership as follows:

```
memb Mt e = False
memb (Lf a) e = a==e
memb (Nd b1 b2) e = memb b1 e || memb b2 e
```

This notion of membership is equivalently expressed catamorphically

```
memb bt e =
    cataB bt
    False
    (\a -> a==e)
    (\b1 b2 -> b1 || b2)
```

Now just as cataB above interprets binary trees as catamorphisms, so does memb interpret binary trees as sets (with Mt as {}, Lf a as {a}, Nd b1 b2 as b1 U b2) in terms of their characteristic predicates..

6.2 Sub-Catamorphic Zoetic Data

So, just as we formed catamorphic zoetic data by the partial application of the relevant catamorphism, so sub-catamorphic zoetic data are formed by partial application of the relevant sub-catamorphic method. For example the partial application memb bt defines a sub-catamorphic zoetic binary tree with the behaviour of the characteristic predicate of the set represented by bt.

In general, partial application of a method creates zoetic data with that method as its characteristic zoetic behavior, exemplified by the partial application of memb(er) above. In that light, partial application of the relevant catamorphism as a method can be thought of as supplying a default characteristic zoetic behaviour for any datatype.

6.3 Sub-Catamorphic Zoetic Generators

Just as with catamorphic zoetic data, it's conceptually efficient to incorporate the desired behavior of sub-catamorphic zoetic data into a set of generator functions that replace the symbolic data constructors.

Derivation of sub-catamorphic generators is a little more complex than the catamorphic case, and exploits the well-known identity property for catamorphisms. That is, when the operands to which a catamorphism is applied to are the corresponding symbolic data constructors, the evaluation result (relation denoted by →) is the original structure, e.g

```
cataL Xs (:) [] → Xs
```

For catamorphic zoetic data and generators as above the identity property maintains their zoetic character, e.g.:

```
cataL Xs zcons znil → cataL Xs
zcons X Xs zcons znil → zcons X Xs
znil zcons znil → znil
cataN X zsuc zero → cataN X
zsuc X zsuc zero → zsuc X
zero zsuc zero → zero
```

That is, zoetic data generators are exactly the operands to the catamorphism that gives the desired catamorphic behavior to the original symbolic data!

Prima facie then, the generators for sub-catamorphic zoetic sets would appear to be the operands of the catamorphic definition of memb(er) above, i.e.

```
False
(\a -> a==e)
(\b1 b2 -> b1 || b2)
```

To serve however as generators, the catamorphism operands must be "closed" i.e. self-contained functions with key inputs as parameters not as global variables, e.g. those used to express memb(er) above are not effective. The effective and simple solution (but NB only in a functional context) is to make the unbound quantity (here the putative element e) into a further parameter, and explicitly apply the resulting higher-order catamorphism to it:

```
memb bt e =
    cataB bt
    (\e -> False)
    (\a -> (\e -> a==e))
    (\b1 b2 -> (\e -> b1 e || b2 e))
    e
```

From the operands to cataB we can read off the zoetic set generators (suitably renamed and presented for the purpose):

```
zempty e = False
zsingle a e = a==e
zunion zs1 zs2 e = zs1 e || zs2 e
```

A similar abstraction-transformation can be applied to the earlier expression of foldl as a catamorphism:

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

This leads to the generators for another kind of zoetic lists i.e. a sub-catamorphism with foldl-style behaviours:

```
zlcons x zxs op b = zxs op (op b x)
zlnil op b = b
```

6.4 Sub-catamorphic zoetic program design

From a design recovery point of view, we thus see that information about zoetic data with sub-catamorphic behaviours can also be consolidated into conceptually-economical generator definitions, just as those with straightforward catamorphic behaviours.

So, just as we define and use catamorphic zoetic generators and data zsuc, zero, zsuc zero etc. above, we define and use sub-catamorphic generators and data. For example, the set-theoretic expression {} U {2, 3} would be rendered as

```
zunion
    zempty
    (zunion (zsingle 2) (zsingle 3))
```

7 Related Work

As cited in our Introduction, some progress has already been made by others in targeting catamorphisms as the result of design recovery. There has also been much work that directly or indirectly promotes the theoretical and practical significance of catamorphisms as manifested in Zoetic data. Zoetic data have made largely-anonymous but impressively-diverse appearances since the origins of programming. Our "zoetic naturals" are exactly the Church numeral representation of natural numbers in the lambda-calculus [12], and our zoetic lists and binary trees are equally recognized and the standard "Church" versions of these types also.

7.1 Combinator parsing

Sub-catamorphic zoetic data are arguably even more impactful. The zoetic representation of sets as characteristic predicates (sub-catamorphic binary trees) are an important concept in mathematics. But maybe even more apparently practically, from a software engineering point of view, is how combinator parsers [13] can be viewed as sub-catamorphic zoetic versions of context-free grammars, with parsing as their characteristic method. This view unfolds as follows.

Begin with a double binary tree, i.e. with two kinds of branch:

```
data Dbt a =
   Tok a
   | Cat (Dbt a) (Dbt a)
   | Alt (Dbt a) (Dbt a)
```

The idea is that the type "Dbt t" corresponds to a context-free grammar with: terminal symbols of S given by leaves "Tok s", concatenation of sub-grammars G1 and G2 given by sub-trees "Cat G1 G2"; and alternation of sub-grammars G1 and G2 given by sub-trees "Alt G1 G2.

In order to discover the sub-catamorphic generators "tok", "cat" and "alt" corresponding respectively to "Tok", "Cat" and "Alt", it's merely necessary to discover the closed-form operands to the catamorphic "parse" operation. Thus, following

```
cataDbt (Tok ys) t c a = t ys
cataDbt (Cat g1 g2) t c a =
   c (cataDbt g1 t c a) (cataDbt g2 t c a)
cataDbt (Alt g1 g2) t c a =
   a (cataDbt g1 t c a) (cataDbt g2 t c a)
```

we can then define the appropriate parser, i.e. that when applied to a string returns the list of strings resulting from stripping from the original string all prefixes matching the grammar being parsed, as follows:

```
parse dbt xs =
   cataDbt dbt
   (\ys xs -> strip ys xs)
   (\g1 g2 xs -> concat (map g2 (g1 xs)))
   (\g1 g2 xs ->  g1 xs ++  g2 xs)
   xs
```

where

```
strip aas bs =
   if found aas bs then [chop aas bs] else
[]
found (x1:x1s) (x2:x2s) =
   if x1==x2 then found x1s x2s else False
found [] xs = True
found (x:xs) []= False
chop (x1:x1s) (x2:x2s) = chop x1s x2s
chop [] xs = xs
```

From the operands to "parse" above we read off

generator definitions

```
tok ys xs = strip ys xs
cat g1 g2 xs = concat (map g2 (g1 xs))
alt g1 g2 xs = g1 xs ++ g2 xs
```

The evident clarity of this approach to parsing, in which language specification and implementation are unified as opposed to the typical approach of requiring a separate interpreter or "parsing engine" corroborates our proposition that program design is usefully expressed in terms of zoetic data.

7.2 "Totally Functional" programming

Another catamorphism-based approach to programming is Turner's TFP (Total Functional Programming) [14, 15] which emphasizes the opportunities and advantages of subrecursion, i.e. avoiding general recursion as enabled by basic recursion patterns.

The work presented in this paper derives from a development of TFP (Total*ly* Functional Programming) [16,17] which refines Turner's by taking catamorphisms to their logical conclusion i.e. refinement of symbolic into zoetic data as per the various developments motivated and elaborated above.

8 Conclusions

We have demonstrated how catamorphisms are useful bases for program structure and design. In order to reach their potential in this role they moreover need to exploit higher-order functions: in order to express other recursion patterns; and in order to enable their comprehensive presentation of program designs in terms of zoetic data.

Consequently, just as catamorphisms have an important role to play in the development phase of software engineering, they offer great promise in software maintenance and re-engineering development at least potential targets for program restructuring/design recovery. This view is encouraged by earlier work in the reverse engineering of recursive programs into catamorphic form [3, 4]. It follows that future research would profitably extend these existing techniques from the established presentations of catamorphisms to the new forms presented above.

Acknowledgments

Earlier work by Ian Peake and Sean Seefried on our TFP project is gratefully acknowledged.

References

[1] J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *CACM, vol. 9*, 1978.

[2] 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.

[3] J. Launchbury and T. Sheard, Warm Fusion: Deriving Build-Catas from Recursive Definitions, *Proc. FPCA 1995*, ACM, New York, 1995, 314-323.

[4] S. Mak and T. van Noort, *Recursion Pattern Analysis and Feedback* (Center for Software Technology, Universiteit Utrecht, 1986).

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

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

[7] Scala, http://www.scala-lang.org/ , accessed 15 July 2012.

[8] J. Gibbons, Datatype-generic programming, in: R. Backhouse, J. Gibbons, R. Hinze and J. Jeuring (eds.), *Spring School on Datatype-Generic Programming 2006, LNCS, vol. 4719*, Springer, Heidelberg, 2007, 1–71

[9] T. Uustalu, V. Vene and A. Pardo, Recursion Schemes from Comonads, *Nordic J. of Comput. vol. 8*(3), 2001, 366-390.

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

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

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

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

[14] D.A. Turner, Elementary Strong Functional Programming, in: R. Plasmeijer and P. Hartel (eds.), *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 and C. Kemp, Fusing Folds and Data Structures into Zoetic Data, *Proc. 23rd IASTED International Multi-Conference on Applied Informatics (AI 2005)*, Acta Press, Calgary, 2005, 299-306.

[17] C. Kemp, *Theoretical Foundations for Practical "Totally-Functional Programming"* (PhD Thesis, The University of Queensland, 2009).