Zoetic Data and their Generators

Paul Bailes and Colin Kemp
School of ITEE, The University of Queensland, St Lucia, QLD 4072, Australia

Keywords: Catamorphism, Church Numeral, Foldr, Functional Programming, Fusion Theorem, Haskell.

Abstract: Functional or "zoetic" representations of data embody the behaviours that we hypothesise are characteristic to all datatypes. The advantage of such representations is that they avoid the need, in order to realize these characteristic behaviours, to implement interpretations of symbolic data at each use. Zoetic data are not unheard-of in computer science, but support for them by current software technology remains limited. Even though the first-class function capability of functional languages inherently supports the essentials of zoetic data, the creation of zoetic data from symbolic data would have to be by repeated application of a characteristic interpreter. This impairs the effectiveness of the "Totally Functional" approach to programming of which zoetic data are the key enabler. Accordingly, we develop a scheme for synthesis of generator functions for zoetic data which correspond to symbolic data constructors but which entirely avoid the need for a separate interpretation stage. This avoidance allows us to achieve a clear separation of concerns between the definitions of datatypes on the one hand and their various applications on the other.

1 INTRODUCTION

The multi-faceted advantages of functional programming have long been well-documented (Hughes, 1989). However, amid the benefits of such as lazy evaluation and referential transparency, the essential defining aspect of programmer-definable higher-order functions, seems strangely to have been under-appreciated. In particular, expositions of functional programming (Bird and Wadler, 1988) (Abelson et al., 1996) typically relegate functional ("Church") representations of data as mere curiosities.

Our purpose here is to demonstrate the viability of these functional (or zoetic: "pertaining to life; living; vital", Collins English Dictionary, http://www.collinsdictionary.com) representations as comprehensive replacements for conventional symbolic data. The focus of the demonstration is on how zoetic data can be manipulated, and specifically created, (or "generated") independently of their symbolic counterparts, and thus form the basis of a "Totally Functional" programming style where symbolic data can be superseded by these zoetic representations.

In this paper overall we: provide a basic justification of zoetic data in terms of general software engineering principles; indicate how widespread and practical zoetic data actually are; provide the conceptual and semantic bases for the synthesis of generators for a wide class of zoetic data; demonstrate the applicability of our synthesis technique for a range of examples; and indicate how zoetic data provide a conceptual gateway into a comprehensive alternate view of programming based on total rather than partial recursion.

2 ZOETIC DATA EXAMPLES

We begin by showing how zoetic data play important roles in functional programming, not just theoretically but practically also, using the a range of examples of natural numbers, set data structures, and context-free grammars. The key idea in each case is that the zoetic counterpart to a conventional symbolic datatype embodies an essential characteristic behaviour.

2.1 Zoetic Naturals

Perhaps the best-known zoetic datatype in programming is the "Church numeral" (Barendregt, 1984) representation of natural numbers, whereby a natural N is represented by a counterpart function (say $\tilde{N}$) such that $\tilde{N}\ f\ x = f^N\ x$. That is, the

Bailes, P. and Kemp, C.
Zoetic Data and their Generators.
In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*, pages 260-271
ISBN: 978-989-758-189-2

characteristic behaviour Ñ of a natural number N is N-fold function composition. For example, the zoetic version of (symbolic) natural number 3 would be rendered, in Haskell concrete syntax (Haskell Programming Language, http://www.haskell.org) as the function:

```
(\f x -> f (f (f x)))
```

or equivalently

```
(\f -> f . f . f)
```

In particular, definitions for some basic zoetic naturals would be

```
zzero = (\f x -> x)
zone = (\f x -> f x)
```

2.2 Zoetic Sets

Perhaps the best generally-known zoetic datatype is the representation of sets by (or equivalently, characterising their essential behaviour in terms of) their characteristic predicates. For example, the essence of the definition of the set of even numbers as { x | x modulo 2 = 0 } is the predicate, or Boolean-valued function (in Haskell syntax)

```
evens = (\x -> x `mod` 2 == 0)
```

Membership of such a zoetic set is tested by direct function application, e.g.

```
evens 4 ->> True
evens 5 ->> False
```

Usefully, as we shall see, the characteristic predicate is the partial application of the set membership operation to the set, as exposed by the tautology

$$S = \{x \mid x \in S\} = \{x \mid (\in S)\ x\}$$

Note the adoption of Haskell operator sectioning, where "(∈ S)" denotes the partial application of ∈ to S, forming the characteristic predicate of S which is then applied to putative element x.

2.3 Zoetic Grammars

The zoetic approach to context-free grammars that is implicitly adopted by combinator parsing (Hutton, 1992) is that the relevant characteristic behaviour of the grammar is the parser for the language defined by the grammar. Accordingly, the renditions of the context-free combinations of concatenation and alternation are (higher-order) functions that apply not to grammars but to parsers, and yielding not a grammar but a parser.

In its essential form, a combinator parser for a grammar g is a nondeterministic recogniser that when applied to an input string s yields the list of suffix strings that result after occurrences of sentences of g have been found as prefixes in s:

```
type CParser = String -> [String]
```

An empty list of result strings signifies failure to parse (Wadler, 1985).

For example, assuming definitions of context-free parsing combinators "conc" and "alt" and token recogniser "tok" (see further below for these), we can define the grammar

```
exp =    exp `conc` ((tok "+")`conc`
trm)
       `alt`
       trm

trm =    (tok "x") `alt` (tok "y")
```

We then parse according it by direct functional application, e.g.

```
(1) exp "x+y"
(2) exp "qwe"
(3) exp "x"
```

Each of these yields respective results

```
(1) ["","+y"]
(2) []
(3) [""]
```

That is, parsing with exp respectively signifies

(1) of "x+y": gives two results, one where the entirety of "x+y" is recognized with no residue, the other where only "x" is recognized leaving residue "+y"
(2) of "qwe": is unrecognized
(3) of "x": is uniquely and fully recognized.

3 CHARACTERISTIC METHODS AS BASIS OF ZOETIC DATA

The key to a systematic approach to generation of zoetic data is the recognition that they embody a uniform interpretation of an underlying symbolic datatype. We use the term "characteristic method" for this interpretation, by extension from the characteristic predicate behaviour ascribed to zoetic sets. The relative advantages of zoetic data based on characteristic method interpretations of symbolic data are exposed by example in the context of natural numbers as follows.

3.1 Pervasive Interpretation Complicates Programming

The need to adopt a uniform interpretation of symbolic data is demonstrated, albeit in microcosm, by the following definitions of arithmetic operations on natural numbers, which expose how thoroughly programming is pervaded by the need to interpret symbolic data, and how potentially harmful are the effects:

```
data Nat = Succ Nat | Zero

add (Succ a) b = Succ (add a b)
add Zero b = b

mul (Succ a) b = add b (mul a b)
mul Zero b = Zero

exp a (Succ b) = mul a (exp a b)
exp a Zero = Succ Zero
```

The drawbacks inherent in these deceptively-simple definitions are profound, as follows.

- Apart from the suggestive naming of the type (Nat) and of its two constructors (Succ and Zero), there is nothing in the definition of the type that compels treatment of members of the type as numbers of any kind, never mind natural numbers specifically.
- Granted there is an obvious isomorphism between the members of Nat and the abstract entities that behave like natural numbers, but that isomorphism needs to be implemented by each usage of Nat. This implementation takes the form of an implicit interpreter that converts symbols into actions (in this case, iterative applications of other functions).
- This implementation of the isomorphism from the symbols of Nat to the iterative behaviour of natural numbers needs to be repeated at each usage: inconsistent usage will lead to inconsistent (erroneous) behaviour.
- Defining functions through interpretation of symbols using general recursion adds the burden of proving totality i.e. termination.

3.2 Explicit Interpretations Offer Simplification

The situation may be clarified somewhat by the introduction of an explicit common interpreter for the semantics (i.e. functional behavior) of natural numbers n as n-fold iterators:

```
iter (Succ n) f x = f (iter n f x)
iter Zero f x = x
```

In the light of this, our arithmetic operation definitions can be re-expressed

```
add a b = iter a Succ b
mul a b = iter a (add b) Zero
exp a b = iter b (mul a) (Succ Zero)
```

The introduction of "iter" thus allows for the clarification of what interpretation is being given to the type Nat (here as iteration), and when that interpretation is being applied usefully and meaningfully.

Despite this clarification however, the revised interpretive arithmetic definitions are still deficient in terms of inconvenience, fragility and potential inconsistency:

- inconvenience, in that the interpreter needs to be applied explicitly;
- fragility, in that the wrong interpreter could conceivably be applied;
- potential inconsistency, in that multiple interpreters with inconsistent behaviours could be defined and applied (e.g. one might assume naturals start at 0, while another might assume they start at 1, as was once a common convention).

3.3 Separation of Concerns via Zoetic Data

All the above criticisms can be summarised as a failure to observe the key software design principle of separation of concerns (Dijkstra, 1982). In this case the separation is between application logic on the one hand and what we might call infrastructure logic on the other. In the above examples, the definitions (add, mul, exp) combine both the logic of the respective applications (addition, multiplication, exponentiation) with the logic of the semantics of natural numbers (iteration). Making the semantic interpreter ("iter") explicit ameliorates the situation somewhat but fails to consummate the separation.

In order fully to achieve separation of concerns between applications and infrastructure, our solution is to require that all members of a datatype are inherently interpreted by the type's characteristic method. Specifically, we:

(1) assume that for each (symbolic) datatype there is indeed a characteristic behaviour (such as iteration for Nat as above);

(2) treat the partial application of the characteristic interpreter (for the characteristic behaviour) to the symbolic data as a conceptual zoetic unit;

(3) reorganise programs around these zoetic data.

In the case of our running example of definitions of basic arithmetic operations, we replace naturals (a,

b, etc.) by zoetic naturals (say za, zb, etc.) where the respective identities hold:

```
za = iter a
zb = iter b
```

etc. Accordingly, we rewrite arithmetic definitions on za, zb etc.:

```
addz za zb = za succz zb
mulz za ab = za (addz zb) zzero
expz za zb = zb (mulz za) zone
```

It is at once evident that the required separation of concerns has been achieved: the only information added by these definitions is with regard to how the zoetic naturals za, zb variously combine to implement the respective arithmetic operations. In particular, the iterative behaviour of za, zb is assumed to have been provided at their creation.

The remainder of this paper this focusses upon how such inherent behaviours are necessarily inbuilt when creating zoetic data, and thus achieving the further required properties of robustness (no chance of applying the wrong characteristic method) and consistency (that there indeed exists a unique characteristic method).

4 GENERATING ZOETIC DATA

The approach we shall follow is simply-stated: instead of creating zoetic data from partial applications of characteristic interpreters to symbolic data, generate the zoetic data directly with zoetic analogues of the symbolic constructors. When programming, calls to symbolic constructors are replaced by calls to the zoetic generators. In other words, we effect an isomorphism between the symbolic and zoetic type. As zoetic generators produce a member of the zoetic type, no final application of the interpreter (iter etc.) is required.

In this section, we preview the definitions of some interesting generators of zoetic data, which we later show how to derive by calculation from their specifications.

4.1 Zoetic Natural Generators

For example, the zoetic versions of natural numbers would be specified in terms of partial application of the above iterative interpreter “iter” to their usual symbolic renditions as follows:

```
zzero = iter Zero
zone = iter (Succ Zero)
ztwo = iter (Succ (Succ Zero))
-- etc.
```

Accordingly, we instead require systematic generation of zoetic naturals such as the above by application of zoetic counterparts of the symbolic constructors Zero and Succ, i.e.

```
zzero = (\f x -> x)
succz n = (\f x -> f (n f x))
```

or equivalently

```
zzero f x = x
succz n f x = f (n f x)
```

and thus

```
zone = succz zzero
ztwo = succz zone
zthree = succz ztwo
-- etc.
```

In particular, it is easy to show (by simple term rewriting from the definitions of zzero and succz) that these correctly yield the expected Church numerals, e.g.

```
zthree
=
succz (succz (succz zzero))
=
(\f x -> f (f (f x)))
```

4.2 Zoetic Set Generators

For zoetic sets it’s easy to intuit generators that apply to appropriate elements or (sub-)sets yielding characteristic predicates that test the membership or otherwise of a putative element x:

```
empty x = False
singleton e x = x==e
union zs1 zs2 x = zs1 x || zs2 x
complement zs x = not (zs x)
-- etc…
```

4.3 Zoetic Grammar Generators

Following our example above, combinator parsers are generated, as are context-free grammars, from *alt*ernation of grammars/parsers, or *conc*atenation of grammars/parsers, or *tok*ens. Alternation accordingly builds a combinator parser from two components p1 and p2, by appending the results of parsing s with each of p1 and p2:

```
alt :: CParser -> CParser -> CParser
alt p1 p2 s = p1 s ++ p2 s
```

Concatenation accordingly builds a combinator parser from two components p1 and p2, by parsing s with p1 and then parsing each of the results with p2:

```
conc :: CParser -> CParser -> CParser
conc p1 p2 s = concat (map p2 (p1 s))
```

A token t parses string s by removing prefix t from s:

```
tok :: String -> CParser
tok t s =
  if prefix t s then [chop t s] else []
```

where

- "prefix t s" tests if string t is a prefix of s;
- "chop t s" removes prefix t from s.

5 GENERATOR SYNTHESIS FOR ALGEBRAIC ZOETIC TYPES

In the context of the above, our (related) problem is:

- to discover the zoetic generators that correspond to symbolic constructors in a systematic way, as opposed to the mere intuitions that have led to the examples above.
- which then enables us to replace the partial applications to symbolic data of characteristic methods/interpreters with direct applications of these generators to zoetic data;

In this section, we deal with the simplest kind of zoetic datatype which corresponds to regular algebraic types (Backhouse et al., 1999). We recognise the generic catamorphic pattern (Meijer et al., 1991) on the regular algebraic type (more widely-known as the foldr operation in the list context, and represented by iter above for naturals) as the characteristic behaviour of its zoetic counterparts.

This choice is made on the basis of:

- category-theoretic justifications of catamorphisms as capturing the essence (categorically-speaking, "initiality") of a regular algebraic type;
- the practical capability of catamorphic patterns to express a wide range of subrecursive operations (Hutton, 1999);
- the above capability including the ability to express other more apparently-sophisticated recursion patterns (Bailes and Brough, 2012).

Just as with Naturals above, zoetic versions of these types in general are specified by the partial application of the relevant catamorphism pattern for that type to the symbolic data. For example, for the types of lists and rose trees:

```
data List a = Cons t (List a) | Nil
data Rose a =
  Tip a | Branch (List (Rose a))
```

we have catamorphism patterns:

```
catL Nil c n = n
catL (Cons x xs) c n =
  c x (catL xs c n)

catR (Tip x) t b = t x
catR (Branch rs) t b =
  b (mapL (\r -> catR r t b) rs)
mapL f xs =
-- corresponds to Haskell prelude
-- "map", but on List t instead of [t]
  catL xs
  (\x xs' -> Cons (f x) xs')
  Nil
```

Note that the usual order of operands in changed to facilitate partial application of the catamorphic pattern to symbolic data. In particular

```
catL xs op b = foldr op b xs
```

Note also how in the case of rose trees, where the recursion is not a simple polynomial, that some additional complexity is entailed in that the structure of the n-ary recursion has to be processed by the relevant map function (in this case over Lists). The resulting list is processed by some combining function b which could well be a (List) catamorphism also. For these Algebraic (catamorphic-pattern-based) Zoetic Types (AZTs), synthesis of the generators proceeds by straightforward equational reasoning, as exemplified by the following.

5.1 Synthesis of Generators for Zoetic Naturals

For example, for zoetic Naturals as defined above, from the specifications of the isomorphism between Nat and our zoetic Naturals, we specify the generators as follows, i.e. as partial applications of the interpreter for the required characteristic behaviour - the relevant catamorphism pattern "iter". Observe how the zoetic operand to generator succz is consistently specified as the partial application of "iter" to symbolic natural n:

```
zzero = iter Zero
succz (iter n) = iter (Succ n)
```

To calculate their implementations, we proceed respectively, in each case adding sufficient relevant parameters to the specifications in order to allow the expansion of the application of iter and then simplification according to the definition of iter in the course of which the interpreter (iter) is eliminated, i.e.:

```
zzero f x
```

= (supplying additional parameters to the RHS also)

```
iter Zero f x
```

= (by definition of iter)

```
x
```

and

```
succz (iter n) f x
```

= (supplying additional parameters to the RHS also)

```
iter (Succ n) f x
```

= (by definition of iter)

```
f (iter n f x)
```

Thus, recognizing the partial application "iter n" as the zoetic natural zn, we have the Haskell function declarations for the zoetic natural generators:

```
zzero f x = x
succz zn f x = f (zn f x)
```

5.2 Synthesis of Generators for Zoetic Lists

Lists are treated similarly, from the specification of generators in terms of partial application of the application of the characteristic symbolic interpreter - in this case the catamorphism pattern for lists "catL":

```
znil = catL Nil
zcons x (catL xs) = catL (Cons x xs)
```

We proceed respectively by adding parameters and simplifying according to the defining equations of catL:

```
znil c n = catL Nil c n = n
```

and (in detail)

```
zcons x (catL xs) c n
```

= (supplying additional parameters to the RHS also)

```
catL (Cons x xs) c n
```

= (by definition of catL)

```
c x (catL xs c n)
```

Thus, recognizing the partial application "catL xs" as the zoetic list zxs, we have the Haskell function declarations to implement the zoetic list generators:

```
znil c n = n
zcons x zxs c n = c x (zxs c n)
```

5.3 Synthesis of Generators for Zoetic Rose Trees

Again, we follow the principle that the specification of generators is in terms of partial application of the application of the characteristic symbolic interpreter - in this case the catamorphism pattern for rose trees "catR". Note especially in this case how the zoetic operand to zbranch is specified as the list of the partial applications of "catR" to each of the symbolic rose trees in the branch, as effected by "mapL".

```
ztip x = catR (Ztip x)

zbranch (mapL catR rs)) =
  catR (Branch rs)
```

To calculate the implementation we as usual proceed respectively

```
ztip x t b
```

= (supplying additional parameters to the RHS also)

```
catR (Ztip x) t b
```

= (by definition of catR)

```
t x
```

and

```
zbranch (mapL catR rs)) t b
```

= (supplying additional parameters to the RHS also)

```
catR (Branch rs) t b
```

= (by definition of catR)

```
b (mapL (\r -> catR r t b) rs)
```

= (abstracting "catR r")

```
b(mapL(\r->(\zr->(zr t b))(catR r))rs)
```

= (identifying function composition)

```
b(mapL(\r->((\zr->(zr t b)).catR) r)rs)
```

= (removing r by eta-reduction)

```
b (mapL ((\zr -> (zr t b)).catR) rs)
```

= (distributing mapL over composition)

```
b (mapL (\zr -> zr t b)(mapL catR rs))
```

Thus, recognizing the list of partial applications "mapL catR rs" as the list of zoetic rose trees zrs, we have the Haskell function declarations for the zoetic rose tree generators:

```
ztip x t b = t x

zbranch zrs t b =
  b (mapL (\zr -> zr t b) zrs)
```

If the n-ary recursive structure of rose trees is represented not by a symbolic list zrs but rather is zoetic, then the effect of mapL on zrs is simply achieved by its direct application as a list catamorphism:

```
zbranch zrs t b =
  b (
    zrs
    (\zr zrs' -> zcons (zr t b) zrs')
```

```
    znil
  )
```

6 SPECIFIC ZOETIC TYPES

The above systemic approach to AZT generator synthesis is however only part of the story. A wider class of zoetic types than AZTs is formed by the partial application of specific characteristic methods rather than the generic catamorphic patterns on the relevant symbolic algebraic types. In other words, while catamorphic patterns represent the most general behaviours, specialisations may be required in specific circumstances. Examples of these as seen so far in this presentation are combinator parsers and characteristic predicates.

Regarding the latter, consider for example the following type of trees with a mixture of binary and unary subtrees:

```
Bt a =
Nul | Lf a | Brn(Bt a)(Bt a)| One(Bt a)
```

This algebraic type has a default interpretation in terms of its catamorphic pattern:

```
catBt Nul n l b o = n
catBt (Lf x) n l b o = l x
catBt (Brn t1 t2) n l b o =
  b(catBt t1 n l b o)(catBt t2 n l b o)
catBt (One t) n l b o = o t
```

Generators for the consequent AZT, derived using the above methods are:

```
nul n l b o = n
lf x n l b o = l x
brn t1 t2 n l b o =
  b (t1 n l b o) (t2 n l b o)
one t n l b o = o t
```

A different interpretation however of these trees as sets is given by the characteristic method "member":

```
member Nul e = False
member (Lf x) e = x==e
member (Brn t1 t2) e =
  member t1 e || member t2 e
member (One t) e = not (member t e)
```

Obviously, zoetic sets that behave as characteristic predicates are given by partial applications

```
member bt  -- NB bt :: Bt a
```

The challenge now facing us, in order to widen the practical range of zoetic data beyond pure catamorphic behaviours, is synthesis of the generators for such specific zoetic types (SZTs). With respect ot the above example, this means empty, singleton, union, complement as further above corresponding respectively to Bt constructors Nul, Lf, Brn and One. This is more complex than that of generic catamorphism-based AZTs, and hence first requires the conceptual infrastructure of the following section.

7 PRINCIPLES OF GENERATOR DERIVATION FOR SZTs

Derivation of generators for SZTs depends upon some further properties common to zoetic data and catamorphisms.

7.1 Catamorphic Expressibility

The continuing central role played by catamorphisms in zoetic data is reflected in the critical assumption that the characteristic functions of specific zoetic data are all expressible as catamorphisms, i.e. as the generic catamorphic patterns themselves (for AZTs) or, as well shall see, specialisations by applying these patterns to appropriate operands to the generic catamorphic patterns on the underlying types (for SZTs).

The basis for this assumption relates to one of the basic premises for zoetic data, i.e. the liberation of programming from the burden of interpretation. Thus, if interpreters don't need to be written, then programming languages don't need to be so complex as to express interpreters. Rather, the expressiveness of catamorphisms a.k.a. "fold" (Hutton, 1999) seems to provide a sufficient basis for all practically-imaginable applications (i.e. other than a Universal Turing Machine or equivalent programming language interpreter). Formally-speaking, the iterative aspect of any function provably terminating in second-order arithmetic (Reynolds, 1985) is expressible as a catamorphism.

Accordingly, our derivations of SZT generators are limited to those for which holds what we call the "Catamorphic-Expressible property" (CE) - that the characteristic behaviour B on some symbolic data D of type T can be expressed as a catamorphism:

(CE) `B D = catT D G1 … Gn`

where

- cataT is the catamorphism on type T
- Gi are the operands to cataT that implement B (which as we see below, are actually the zoetic generators we seek).

7.2 Schematic Catamorphism

Demonstration of key properties common to catamorphisms is facilitated by a scheme of catamorphisms captured in Haskell source code by definitions as follows (Uustalu et al., 2001).

First, identify some general algebraic abstractions:

```
type Algebra f a = f a -> a

-- as per Haskell prelude
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The key to a generic definition of the catamorphic pattern is the pattern functor that defines the shape of the data for a (recursive) type. In order to isolate the (non-recursive) pattern functor, direct recursion in type definitions is not appropriate. Recursive types are instead the explicit fixed-point of their pattern functor, thus we need a fixpoint operator for data types:

```
newtype Mu f = InF { outF :: f (Mu f) }
```

(The respective constructor and extractor functions for Mu - Inf and outF - are artefacts of the Haskell type system.)

For example, the pattern functor type of the polymorphic list type (with elements of type 'a') is:

```
data Lf a lf = Nl | (Cns a lf)
instance Functor (Lf a) where
   fmap g Nl = Nl
   fmap g (Cns x xs) = Cns x (g xs)
```

Thus, the actual recursive polymorphic List type is the least fixed point of "Lf a":

```
type List a = Mu (Lf a)
```

The catamorphic recursion pattern for any type is the most general homomorphism from the algebra given by the pattern functor of the type to any other result algebra (i.e. the polymorphic target type of the recursion pattern). A generic rendition in Haskell is

```
cata ::     Functor f =>
           Algebra f a -> Mu f -> a
cata f = f . fmap (cata f) . outF
```

where 'f' embodies the embedded operation that characterises the catamorphism in terms of the pattern algebra of the type. Observe how the recursive application of "cata f" by fmap ensures the desired recursive operation of the catamorphism.

So in order to define specific catamorphic operations, all that is required is to define the embedded operation (the 'f' parameter of cata). For example:

- the catamorphic definition of the length of a list is

  now

  ```
  length xs = cata phi where
     phi Nl = 0
     phi (Cns _ xs) = 1+xs
  ```

- the list catamorphism pattern (catL) as above can be written simply by passing its parameters to the catamorphism's characteristic operation:

  ```
  catL xs o b = cata phi where
     phi Nl = b
     phi (Cns x xs) = o x xs
  ```

7.3 Fusion Theorem

Just as catamorphisms exemplify how the programming of recursion can be packaged and simplified, so fusion (Hutton, 1999) is an example of how reasoning about recursive programs can be packaged and simplified.

In terms of the schematic catamorphism above, fusion is the implication:

h (phi x) = chi (fmap h x)
→
h (cata phi x) = cata chi x

where phi and chi are the embedded operations of type-compatible catamorphisms

Fusion for individual types can be derived from the above schema, typically with the antecedent of the implication as: the conjunction of the instantiation of the schema with the particulars of each of the constructors for the relevant pattern functor.

For example, for list catamorphisms the two characteristic operations are typified by

```
phi Nl = B1
phi (Cns x xs) = O1 x xs

chi Nl = B2
chi (Cns x xs) = O2 x xs
```

Thus for some base values Bi and binary operations Oi, we instantiate the fusion theorem for lists as follows.

- The antecedent condition, case Nl, is:

  ```
  h (phi Nl)= chi (fmap h Nl)
  ⇔
  h (phi Nl)= chi Nl
  ⇔
  h B1 = B2
  ```

- The antecedent condition, case Cns x xs, is

  ```
  h (phi (Cns x xs))=
                chi (fmap h (Cns x xs))
  ⇔
  h (phi (Cns x xs)) =
                chi (Cns x (h xs))
  ```

```
⇔
h (O1 x xs) = O2 x (h xs)
```

- There is a single consequent:
```
h (cata phi xs) = cata chi xs
⇔
h (catL xs O1 B1) = catL xs O2 B2
```

Thus, conjunction of the consequents gives the single implication:

```
h B1 = B2 ∧ h (O1 x xs) = O2 x (h xs)
➔
h (catL xs O1 B1) = catL xs O2 B2
```

7.4 Identity Property and Constructor Replacement

The Identity property for Catamorphisms (Backhouse et al., 1999) (IC) is crucial in our development. Its simplest typical form is:

(IC) `catT D C1 … Cn = D`

where the Ci are the (symbolic) constructors for (symbolic type) T and (of course) D :: T. That is, applying a catamorphism to a structure with the structure's own constructors yields the same structure.

IC follows from how a catamorphism can be thought of as implementing constructor replacement with the catamorphism operands. In terms of the schematic catamorphism, observe how the embedded operation 'f' is applied by cata recursively to and across each level of substructure. At each level the embedded operation is applied to an instance of the pattern algebra where the constructors are replaced according to the programming of the embedded operation.

7.5 Identifying and Deriving Generators

The values Gi used in CE above as operands to the relevant catamorphic pattern cataT to express SZT behaviours B can be demonstrated to have a critical use as follows. For each distinct case of CE, typically

```
B D = catT D G1…Gn
```

we first use IC to expand the LHS systematically, in what we identify as a Derivative of Catamorphic-Expressibility (DCE):

(DCE) `B (catT D C1…Cn)= catT D G1…Gn`

or, in terms of the schematic catamorphism

B (cata phi D) = cata chi D

where embedded phi and chi as usual replace constructors Ci, in this case for phi by themselves and for chi by the Gi.

At this point, fusion is applicable, i.e. to establish the above identity, we need schematically

chi (fmap B x) = B (phi x)

or typically

```
Gi (B args′ ...) = B (Ci args ...)
```

where "args …" are the operands to which Ci applies to produce some D :: T, and "args′ …" are the args … but with Ci uniformly replaced throughout by Gi.

Thus, the operands Gi (that are used to express the behaviours of SZTs) are not only calculable by fusion, but they are also the zoetic generators corresponding to the symbolic constructors. From this point equational reasoning yields implementations of Gi as for generic catamorphic types as above. Illustrative examples now follow.

8 GENERATOR DERIVATIONS FOR EXEMPLARY SZTs

8.1 Derivation of Zoetic Set Generators

Recall the type of trees with a mixture of binary and unary subtrees:

```
Bt a =
Nul | Lf a | Brn(Bt a)(Bt a)| One(Bt a)
```

for which the the relevant catamorphic pattern is catBt (as defined above).

The relevant fusion law (derivable from the earlier fusion schema) is

$h\ n_a = n_b$
$\wedge$
$h\ (l_a\ x) = l_b\ x$
$\wedge$
$h\ (b_a\ t_1\ t_2) = b_b\ (h\ t_1)\ (h\ t_2)$
$\wedge$
$h\ (o_a\ t) = o_b\ (h\ t)$
$\rightarrow$
$h\ (catBt\ t\ n_a\ l_a\ b_a\ o_a) = catBt\ t\ n_b\ l_b\ b_b\ o_b$

Now, if these trees are to be interpreted as zoetic sets by the characteristic method "member" above, the relevant expression of DCE in this case is:

```
member (catBt bt Nl Lf Brn One)
=
catBt bt m s u c
```

Application of fusion gives:

```
(1) member Nul = m
```

```
(2) member (Lf x) = s x
(3) member (Brn t1 t2) =
      u (member t1) (member t2)
(4) member (One t) = c (member t)
```

From this point, equational reasoning respectively proceeds in each case:

```
(1) m e = member Nul e = False
(2) s x e = member (Lf x) e = x==e
(3) u (member t1) (member t2) e
    = member (Brn bt1 bt2) e
    = member t1 e || member t2 e
(4) c (member t) e
    = member (One t) e
    = not (member t e)
```

Thus, recognising in particular the partial applications "member ti" as zoetic sets zsi, we have derived implementations for zoetic generators of empty, singleton, union and complement of sets respectively m, s, u and c:

```
m e = False
s x e = x==e
u zs1 zs2 e = zs1 e || zs2 e
c zs e = not (zs e)
```

which are identical (modulo names) to the intuitive definitions offered originally far above.

8.2 Derivation of Zoetic Grammar Generators

Based on an interpretation of two-flavoured Rose trees (where the different "flavours" are distinguished by respective constructors B1 and B2), we can specify and derive implementations for n-ary versions of the parsing combinators conc and alt from far above. The basic infrastructure is as follows. (For simplicity of presentation, native Haskell lists are used to implement the n-ary subtree structure.)

```
data Rose2 =
  Tip String | B1 [Rose2] | B2 [Rose2]

catR2 (Tip s) t b1 b2 = t s
catR2 (B1 r2s) t b1 b2 =
  b1 (
    map (\r2 -> catR2 r2 t b1 b2) r2s
  )
catR2 (B2 r2s) t b1 b2 =
  b2 (
    map (\r2 -> catR2 r2 t b1 b2) r2s
  )
```

The relevant fusion law (again derivable from the earlier fusion schema) is

$h\ (t_a\ x) = t_b\ x$
$\wedge$
$h\ (b1_a\ rs) = b1_b\ (map\ h\ rs)$
$\wedge$
$h\ (b2_a\ rs) = b2_b\ (map\ h\ rs)$
$\rightarrow$
$h\ (catR2\ rs\ t_a\ b1_a\ b2_a) = catR2\ rs\ t_b\ b1_b\ b2_b$

Then the following interpretation (by "parse") ascribes behaviours

- to Tip: the behaviour of a token
- to B1: the behaviour of n-ary alternation
- to B2: the behaviour of n-ary concatenation:

```
parse (Tip tk) str =
  if prefix tk str
  then [chop tk str]
  else []
-- prefix, chop as before

parse (B1 r2s) str =
  concat (
    map (\p -> p str)(map parse r2s)
  )

parse (B2 r2s) str =
  foldr
  (\p ss -> concat (map p ss))    -- op
  (head (map parse r2s) str)      -- b
  (reverse (tail (map parse r2s)))-- xs
```

The required n-ary generators tok, nalt and nconc are specified by the relevant expression of DCE:

```
parse (catR2 rs Tip B1 B2)
=
catR2 rs tok nalt nconc
```

From the above, fusion yields:

```
(1) parse (Tip tk) = tok tk
(2) parse (B1 r2s) =
      nalt (map parse r2s)
(3) parse (B2 r2s) =
      nconc (map parse r2s)
```

Equational reasoning respectively proceeds

```
(1) tok tk str
    = parse (Tip tk) str
    = if prefix tk str
      then [chop tk str]
      else []

(2) nalt (map parse r2s) str
    = parse (B1 r2s) str
    = concat (
        map(\p-> p str)(map parse r2s)
      )

(3) nconc (map parse r2s) str
    = parse (B2 r2s) str
    = foldr
      (\p ss -> concat (map p ss))
     (head (map parse r2s) str)
     (reverse (tail (map parse r2s)))
```

Finally, recognising "parse (Tip ts)" as "tok ts" and "map parse r2s" as the list of zoetic grammars gs, and p(arser) as zoetic g(rammar) yields implementations as follows. The implementation of tok repeats the intuitive definition above:

```
tok tk str =
  if prefix tk str
  then [chop tk str]
  else []
```

The respective implementations of nalt and ncat are evident generalisations of the intuitive definitions of binary alt and conc of traditional combinator parsers above:

```
nalt gs str =
  concat (map (\g -> g str) gs)

nconc gs str =
  foldr
  (\g ss -> concat (map g ss))
  (head gs str)
  (reverse (tail gs))
```

9 RELATED WORK

We have already emphasised how zoetic data, while not recognised or identified distinctively as such, are not uncommon to functional programming in general (e.g. characteristic predicates, combinator parsers).

Our grounding of zoetic data in catamorphic recursion patterns establishes a further link, to Turner's "Total Functional Programming" (Turner, 2004) which emphasises the use of subrecursive program structuring mechanisms to ensure that its programs avoid unproductive non-termination i.e. are total functions). The basis for the link is that our grounding of zoetic data in catamorphic recursion patterns also ensures functional totality.

Thus, our "Totally Functional" programming develops Turner's, as follows.

(1) For every datatype there is posited a characteristic method, so that symbolic data are completely ("totally") replaced by functional representations.

(2) However, these characteristic methods are all ultimately definable totally as catamorphisms: in the case of AZTs, directly in terms of the catamorphic pattern that can be thought of as characterising the type; in the case of SZTs, by application of an underlying catamorphic pattern to operands (that turn out to be the generators for the SZT).

Note that not every function of interest to us is expressible as a (single) catamorphism. For example the catamorphic pattern catR for Rose trees requires mapL on the list of subtrees, itself definable in terms of the list catamorphism pattern catL. Also, other operations (as basic as inserting an element into a sorted list) may involve non-iterative post-processing of the results of catamorphisms - see Bailes and Brough (2012) for a summary. However, the essential iterative behaviour of non-trivial zoetic data seems to remain amenable to our methods as above.

We acknowledge that programming based on the catamorphisms implicit in regular recursive type definitions is not original e.g. Coq (The Coq Proof Assistant, https://coq.inria.fr/); however we take the further step of attempting to treat all data as behaviour, i.e. "totally functional".

10 FUTURE DIRECTIONS

We recognize the need for further work in some key areas as follows.

First, in view of the evident usefulness of the categorical dual of list catamorphisms - for lists the "unfold" (Gibbons et al., 2001), or "anamorphisms" more generally - zoetic versions of these as embraced by Turner in his Total Functional Programming (above), need exploration. We anticipate that for every AZT there would be a dual algebraic zoetic co-datatype, and that from these specific zoetic co-datatypes are derivable.

Second, automatic type inference is not available for zoetic data, because they require functional types that transcend the expectations implicit in Milner (1977) and its derivatives. For example, the simple application

```
expz ztwo ztwo
```

fails (spectacularly) to type-check, with 28 lines of error message from WinGHCi (Haskell Platform, http://www.haskell.org/platform). A cleverer definition of expz solves the problem in this case:

```
expz za zb = zb za
```

However, replacement of straightforward definitions by such subtleties does not seem to be the basis of a sustainable solution. Higher type systems (Vytiniotis et al., 2006) offer apparent remedies, but the cost of the loss of the convenience of inference remains to be understood.

11 CONCLUSIONS

The concept of zoetic data arises from the observation that much of the complexity of programming arises from the need to interpret characteristic behaviours for symbolic data each time they are used. Strict adherence to the principle of separation of concerns however indicates that this problem would be addressed by decoupling the definition of datatypes from their various applications.

In our "totally functional" approach to programming this separation is achieved by replacing constructors of symbolic data with zoetic data generators that produce functional representations of data that embody the characteristic behaviours inherent to each datatype. Specific characteristic behaviours arise from application of generic catamorphic patterns to operands that have the effect of defining a more specialised catamorphism. If however such specific behaviour isn't articulated, the catamorphic pattern for the underlying regular algebraic type is itself the characteristic behaviour. From these bases the zoetic generators arise as formally-derived counterparts to symbolic data constructors, thus completely bypassing symbolic data and their interpreters.

ACKNOWLEDGEMENTS

We gratefully acknowledge our various colleagues' contributions over the years to our ongoing work reflected here, especially those of Leighton Brough.

REFERENCES

Abelson, H., Sussman G.J. and Sussman, J., 1996. *Structure and Interpretation of Computer Programs* 2nd ed. MIT Press.

Backhouse, R., Jansson, P., Jeuring, J. and L. Meertens, 1999. Generic Programming - An Introduction. In S. Swierstra, S., Henriques, P. and Oliveira, J. (eds.), *Advanced Functional Programming*, LNCS, vol. 1608, pp. 28-115.

Bailes, P. and Brough, L., 2012. Making Sense of Recursion Patterns. In *Proc. 1st FormSERA: Rigorous and Agile Approaches*, IEEE, pp. 16-22.

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

Bird R. and Wadler, P., 1988. *Introduction to Functional Programming,* Prentice-Hall International.

Coq Proof Assistant, https://coq.inria.fr/, accessed 22 February 2016.

Collins English Dictionary, http://www.collinsdictionary.com, accessed 4 July 2014.

Dijkstra, E., 1982. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*, pp. 60-66. Springer-Verlag, New York.

Gibbons, J., Hutton G. and Altenkirch, T., 2001. When is a function a fold or an unfold?. In *Electronic Notes in Theoretical Computer Science,* vol. 44 (1).

Haskell Platform, http://www.haskell.org/platform/, accessed 4 July 2014.

Haskell Programming Language, http://www.haskell.org, accessed 4 July 2014.

Hughes, J., Why Functional Programming Matters, 1989. In *The Computer Journal*, vol. 32 (2), pp. 98-107.

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

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

Meijer, E., Fokkinga, M. and Paterson, R., 1991. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA 1991*, LNCS vol. 523, pp. 142-144.

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

Reynolds, J., 1985. Three approaches to type structure". In *Mathematical Foundations of Software Development*, LNCS, vol. 185, pp. 97-138.

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

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

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

Wadler, P., 1985. How to Replace Failure by a List of Successes. In *Proc. FPCA 1985*, LNCS, vol. 201, pp. 113-128.