FUSING FOLDS AND DATA STRUCTURES INTO ZOETIC DATA

Paul A. Bailes, Colin J. M. Kemp
School of ITEE
The University of Queensland QLD 4072
Australia
{paul, ck}@itee.uq.edu.au

ABSTRACT
The persistent difficulty of programming is indicative of the need for further reduction in the complexity of programming languages. Systematic simplification of functional programs is achieved through the packaging of recursion via "fold" functions, but we are led to even further simplifications by conceiving of partially-applied folds in terms of providing enlivened or "zoetic" views of otherwise inert data. Importantly, zoetic animations can indeed be found in more general structures than just pure folds, and fold-theory is critical in the synthesis of these wider applications. This zoetic style has a strong grounding in the theoretical development of functional programming and has interesting links to object-oriented programming, and subrecursive and analog computing. Its identification moreover in some of the most striking applications of higher-order functional programming leads us to suspect that this zoetic style is the ultimate simplification of functional programming.

KEY WORDS
Fold, Functional programming, Higher-order functions

1. Avoiding Interpretation in Programming

Programming persists in remaining complex, despite much research into programming methodology, tools and languages. We contend that one hitherto-overlooked factor in this complication is the need for programs (and indirectly programmers) to "enliven" the inherent underlying behaviour out of inert data through explicit interpretation. If data could be "pre-enlivened", i.e. represented "zoetically" in such a way that the inherent underlying behaviours were directly available, then at least some of the complexity of programming could be avoided.

Our ultimate goal accordingly is to discover a radical simplification of software development by exploring the functional paradigm to its limits. "Normal" functional programming [2] can be said to be characterized by the *opportunity* for unrestricted definition and use of higher-order functions. Our proposed ultimate style of functional programming however *obliges* to the greatest possible extent the replacement of non-functional components, i.e. data and data structures, by appropriate alternatives – functions – that implement the essential applicative behavior that is hypothesised to be characteristic to the processing of each data (structure) type.

The approach that we take in what follows is to begin with functional programming, in which zoetic representations and operations thereon (functions, including higher-order functions) may at least be manipulated freely. We then consider an existing and proven stylised scheme for processing symbolic data – "fold" functions – and transform it into an actual view of data as zoetic animations, as opposed to hitherto inert objects. This relationship between folds and functional representations of "pure" data types entirely characterised in terms of their type-signatures is well-known. However, we establish these representations as part of a more comprehensive regime of zoetic data, and show how direct definitions of zoetic interfaces to more general data types are relatively easily derived from specifications. We are careful not to suggest that zoetic programming would serve as an entire substitute for the interpreted style, but rather that there seem to be numerous occasions when zoetic animation is highly advantageous. Perhaps in large systems, the two styles may need to co-exist, with overall improvement being effected by the choice wherever possible of zoetic animation over interpretation.

We have coined the phrase "Totally Functional Programming" (TFP) to reflect our goal of complete, or rather as complete as practicable, dependence upon functions rather than data. (It also emerges that TFP would seem to encourage subrecursive programming, hence the suggestion of mathematically total as opposed to partial functions.)

2. Programming with Folds

Deriving, writing and verifying functions over (structured) data types are beneficially structured by "fold" operators. TFP may be seen as a bringing-to-fruition of these benefits, and "folds" play key roles in understanding and applying TFP.

2.1 Folds Simplify Programming

A key insight in the development of programming methodology was that common programming patterns should be encapsulated by "structured" control constructs [1]. A key advantage of functional programming is that this encapsulation can be achieved by programmer-definable higher-order functions [2]. Thus, the well-known list "fold" function [3] (earlier known as "reduce" in APL), definable (here, as throughout this paper, in Haskell [4]) as

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

simplifies the expression of functions on lists (and also their derivation/verification – see below). For example, the definitions:

```
1. sum = fold (+) 0
2. map f = fold (\ x xs → f x : xs) []
3. reverse = fold append []
        where append x xs = xs ++ [x]
```

respectively define the functions which:

1. sum elements of a list (of numbers)
2. apply a function f to each element of a list
3. reverse the elements of a list.

The key point is that the explicit recursion that would otherwise be required in each case is encapsulated within "fold". (Indeed this approach is immediately extendable to all regular recursive datatypes, for which analogies to list "fold" exist – see below for examples.)

2.2 Folds Support Formal Methods

The simplification of program structures that results from expressing code fragments in terms of characteristic patterns such as "fold" is paralleled by a simplification of the processes of program derivation and verification that result from the availability of laws/theorems about these components. Such simplifications are applicable to patterns other than "fold", but the key role played by "fold" in the zoetic animation of data makes it beneficial to focus on "fold" as follows.

A particular advantage of functional (as opposed to mere applicative) programming is that it exploits the fact that this process (of recognising, identifying and reusing patterns) is applicable not just to first-order operations on data, but also to the higher-order constructs that combine and produce functional program components. That is, just as programmers may define what are, in effect, their own control constructs (such as "fold"), then derived laws about the behaviours of these constructs can be used in deriving and verifying programs that use them. Specifically, formal proofs of list-processing functions need not depend upon induction, but rather depend upon higher-level laws about "fold" (which, granted, are ultimately inductively-proved, but that is the limit of the need for induction).

One of the most useful such laws is the "fusion" property [3]: there holds the identity

H (fold G W Xs) = fold F V Xs

provided that there also holds the conjunction of

H W = V

H (G Y Ys) = F Y (H Ys)

Consider for example, proving that list concatenation is associative: "(A ++ B) ++ C = A ++ (B ++ C)". First, instead of the usual recursive definition (in pseudo-Haskell, with infixed "++" for concatenation):

```
[ ] ++ bs = bs
(a:as) ++ bs = a:(as ++ bs)
```

use rather a fold-based definition:

```
as ++ bs = fold (:) bs as
```

Now, the associativity requirement can be re-expressed, expanding on both sides the new definition of "++", as

fold (:) C (fold (:) B A) = fold (:) (fold (:) C B) A

This now matches the terms of the fusion law, where

H = fold (:) C *G = (:)*

W = B *Xs = A*

F = (:) *V = fold (:) C B*

In order to complete the proof we only have to show that the following hold:

1. *fold (:) C B = fold (:) C B*
   (trivially)
2. *fold (:) C ((:) Y Ys) = (:) Y (fold (:) C Ys)*
   … iff (infixing "(:) Y Ys" to "Y : Ys") …
   fold (:) C (Y : Ys) = (:) Y (fold (:) C Ys)
   (which matches the definition of "fold" with op = "(:)", b = C, x = Y and xs = Ys)

2.3 Generality of Fold

As foreshadowed above, the (possibly recursive) datatypes definable e.g. in modern functional languages all have counterparts to "fold". Simple examples are as follows.

- Booleans:

```
data Bool = True | False
foldB t f True = t
foldB t f False = f
```

- Natural numbers:

```
data Nat = Succ Nat | Zero
foldN s z Zero = z
foldN s z (Succ n) = s (foldN s z n)
```

- Binary trees:

```
data BinT t = Null | Leaf t | Branch (BinT t) (BinT t)
foldBT n l b Null = n
foldBT n l b (Leaf elt) = l elt
foldBT n l b (Branch st1 st2) =
        b (foldBT n l b st1) (foldBT n l b st2)
```

In general, as exemplified by the above, for a regular ADT T with constructors C1 … Cn where each Ci has arity mi and the jth operand is of type Tj (without loss of generality, this also applies for polymorphic T), i.e.,

data T = ... | Ci T1i ... Tmi | ...

then, the definition of "foldT" (i.e. fold for type T) consists of a set of equations, one for each different pattern of construction (i.e. depending on each different constructor Ci) in T:

foldT c1 … cn (Ci a1 … ami) = ci A1 … Ami

where

- each equation includes formal parameters ci corresponding to constructors Ci
- the body of the equation for Ci applies corresponding parameter ci to operands Aj, each in turn corresponding to the operands of Ci
- each operand Aj of ci is derived from operands aj of the prevailing Ci: if aj corresponds to a nested element of T, then Aj is aj "folded", i.e. of the form "foldT c1 … cn aj"; otherwise Aj is just aj

Definitions of (recursive) functions on these types accordingly may be simplified using the appropriate fold. For example, a recursive function to count the number of nodes in a binary tree

```
numnodes Null = 0
numnodes (Leaf elt) = 1
numnodes (Branch sub1 sub2) =
        numnodes sub1 + numnodes sub2
```

is more simply defined as

```
numnodes t = foldBT 0 (\elt → 1) (+) t
```

Appropriate versions of the fusion law of course apply for these fold-variants, too. Thus:

- Booleans:
  H (foldB W1 W2 B) = foldB V1 V2 B
  provided that
  H W1 = V1
  H W2 = V2
- Natural numbers:
  H (foldN G W N) = foldN F V N
  provided that
  H W = V
  H (G N) = F (H N)
- Binary trees:
  H (foldBT W G1 G2 T) = foldBT V F1 F2 T
  provided that
  H W = V
  H (G1 X) = F1 X
  H (G2 U1 U2) = F2 (H U1) (H U2)

and so on, for all regular recursive types.

3. Folds Enliven Data

"Fold" functions reveal themselves as more than just a notational simplification, but as a window on a new view of data as zoetic, i.e. enlivened as opposed to inert, objects.

3.1 Inert vs. Zoetic Data

The relationship between language design/extension and programming has been previously-canvassed (e.g. [5]). An important implication for programming (as a kind of language extension) is that just as language extension should be prosecuted (executed, evaluated and constrained) as a kind of programming (i.e. by direct definition and eschewing interpretation), so should programming be prosecuted as a kind of direct "definitional" language extension rather than by "interpretation". That is, if a particular style of language extension is to be avoided, so should the corresponding kind of programming.

Because interpretation is all about animating operations from inert symbolic representations, it follows that the opposite (and preferred) direct-definitional programming style should instead support the provision of pre-enlivened representations for what were previously inert data.

3.2 Inverting Fold

First, it's straightforward to re-order the operands to "fold". Consider how as commonly-presented (e.g. above), "fold" is an abstraction of a particular data structure from a pattern of operations on the structure. As above, the partial application, for the datatype of lists,

fold O B

(for some specific O, B) is synonymous with the abstraction of the data to which the "fold" is applied:

\ xs → fold O B xs

Alternatively however, rather than the data being abstracted from the operations, the operations may be abstracted from the data (some specific Xs), viz.

\ op b → fold op b Xs

for some specific Xs. The same effect of this alternative abstraction may be achieved more directly by defining an "inverted" variant of fold, say "ifold", which inverts operands of fold such that

ifold xs op b = fold op b xs

Thus "ifold Xs" folds Xs according to yet-to-be-supplied parameters "op" and "b".

Obviously, corresponding to the folds "foldT" that exist for all types T, there may be defined inverted folds "ifoldT".

Further, it's obvious that the fusion law applies to inverted folds, suitably inverted. For lists for example

H (ifold Xs G W) = ifold Xs F V

provided that as before

H W = V

and

H (G Y Ys) = F Y (H Ys)

3.3 Inverted Partial Applications of Fold

Partial applications of inverted folds (for example "ifold" to lists Xs giving "ifold Xs") therefore transform lists Xs into functions (on the further operands of "ifold"). Mechanically, these functions apply to operations/data O/B, replacing within Xs: (a) applications of list constructor Cons (i.e. ':') with operation O; and (b) occurrences of Nil (i.e. "[ ]") by B. Similar remarks apply to general (inverted) folds "ifoldT" over other types T.

The hitherto conventional view of a data structure, as a complex of typically nested applications of constructors, is thus revealed as a special case of an inverted fold

application – in this case, to the constructors themselves! For example

ifold [x1, x2, x3] (:) [ ]

= (expressing the list in terms of explicit application of ':' between elements)

ifold (x1:x2:x3:[ ]) (:) [ ]

= (applying "ifold" to "x1:x2:x3:[ ]")

(\op b → op x1 (op x2 (op x3 b))) (:) [ ]

= (substituting for "op", "b")

(:) x1 ((:) x2 ((:) x3 [ ]))

= (writing prefix application of sectioned ':' in infix form)

x1:x2:x3:[ ]

= (reverting to usual notation)

[x1, x2, x3]

In general, for some type T with constructors Ci, consequent (inverted) fold "ifoldT", and instances D, the following identity holds:

ifoldT D C1 ... Cn = D

or equivalently, expanding D into some construction of one of the constructors Ci: "(Ci opds …)"

ifoldT (Ci opds ...) C1 ... Cn = (Ci opds ...)

(see also "General Properties of Zoetic Data" below).

3.4 Inverted Partial Applications as Zoetic Data

The partial application "ifold Xs" therefore reveals an inherent applicative behaviour of lists: as functions, that apply to a binary operation O and a "base element" B, and apply O successively to elements of the list and the result of its application to the remainder, with B as result on the "Nil" case. Upon reflection, it is apparent that this behaviour captures the essence of a (finite) list, as an ordered sequence of elements:

- the asymmetry of O (in general) expresses the ordering of the elements of the list;
- the combination of how O is applied to the head element and the result of the tail sub-list, together with the role of B with respect to the empty list, expresses the arbitrary but finite length of the list;
- the type of the left operand of O reflects the homogeneity of the list.

More generally, partial applications to data structures of the appropriate inverted fold reveal the inherent applicative behaviours, or "animations", of data structures- as functions that inject their arguments into a relationship between the elements of the structure that is characteristic of the structure itself.

We are thus emboldened to propose that it is the generic relationship between the elements and fold operations embodied in the fold that "is" (the defining characteristic of) the data type, and equally that the partial applications of the appropriate fold to the data (structures) that "are" the data (structures). This proposition is substantiated by:

- the extent to which functions on data (structures) may be defined in terms of "fold" (or equally the inverted versions thereof);
- that if direct definitions of such functions in terms of fold are impossible or inconvenient, then the "original" data structures may be recovered by applying the zoetic structures (resulting from partial application of inverted folds to data) to the data constructors, and processing the result in the traditional manner.

In other words, we contend that of the two roles of data distinguished above- (i) as operands (ii) as representations of underlying operations – that the first is in fact an illusion, and that the second is what all data really are.

To conclude: fold-based programming is conceptually synonymous with programming with the enlivened versions of data that result from partial applications to data of inverted folds. We are thus led to consider the generality of the potential of replacing inert data by animations thereof.

3.5 General Properties of Zoetic Data

It will be useful below to note that the relationship identified above between data structures D on the one hand, and (inverted) fold over D applied to the constructors for D on the other:

ifoldT D C1 ... Cn = D

is derived from a deeper relationship between zoetic data and their operands. Specifically, it follows from the definitions of "ifoldT" that applying zoetic data derived from constructions of some Ci, to operands X1 … Xn, results in the Xi corresponding to Ci, as applied to operands as determined by the structure of the type and reflected in the definition of the (inverted) fold.

For example:

ifoldBT (Branch T1 T2) Null Leaf Branch
=
Branch
(ifoldBT T1 Null Leaf Branch)
(ifoldBT T2 Null Leaf Branch)
=
Branch T1 T2

because each of the applications

ifoldBT Ti Null Leaf Branch

(i = 1, 2) will correspondingly transform into Ti.

4. Direct Generation of Zoetic Representation

The zoetic animation of structures, by inverted folds as above, turns out to be a special case of a more general approach to zoetic data. These zoetic data structures (and the inverted folds) considered above may be considered "pure", in that no comparisons to other data are involved. However, many data structures are "impure" in that comparisons with other data are inherent. This is an important distinction, because comparison of symbolic data is the essence of interpretation which we are striving to avoid, as far as possible. Nevertheless, it seems practically impossible to avoid all reference to symbolic data, and a minimum necessary degree of interpretation in the TFP approach. How then can the zoetic view as above be extended to such impure structures, and how do folds assist with construction of such extended zoetic views?

4.1 Zoetic Sets

Consider for example sets of elements, where the key operation is membership testing of putative elements. This necessitates comparing the putative elements with the elements in the set, and hence is "impure" by our definition.

A specification for simple but effective polymorphic sets of elements of type 't' is as follows.

```
data Set t = Empty | Single t | Union (Set t) (Set t)
member Empty elt = False
member (Single x) elt = x==elt
member (Union s1 s2) elt = (member s1 elt) or (member s2 elt)
```

It will be immediately observed that the type-signature of such sets is isomorphic to that for binary trees, and consequently an isomorphic (inverted) fold operator can be used, as well as the fusion law that follows. In particular:

```
ifoldS Empty e s u = e
ifoldS (Single x) e s u = s x
ifoldS (Union s1 s2) e s u = u (ifoldS s1 e s u) (ifoldS s2 e s u)
```

The "inverted fusion" law for sets is

H (ifoldS R G1 G2) = ifoldS S V F1 F2

provided that

H R = S
H (G1 X) = F1 X
H (G2 R1 R2) = F2 (H S1) (H S2)

The zoetic behaviour we seek, for sets with membership testing as the characteristic impure operation, is as functions that apply the membership test to the putative members as operand, which is better known as the "characteristic predicate" representation for sets. Trivially, this can be achieved by routine partial application of the distinguished selector ("member" here) to constructions (sets), and as such admirably parallels the routine application of the appropriate "ifold" to constructions in order to create pure zoetic animations. Thus, instead of constructing sets S and subsequently applying "member S X" for putative elements X, instead: construct S; partially apply "member S" (call the result SA); and subsequently apply the result directly to X: "SA X".

4.2 Composing Impure Animations with Generators

4.2.1 Composing Animations. However, the above zoetic animation of sets is deficient in that it is not *compositional* – having thus formed zoetic animations SA, to form unions etc. it would be necessary to refer to the original construction S.

For example, given

```
SA1 = member S1
SA2 = member S2
```

then the zoetic animation of the union would have to be created from the union of the original unenlivened constructions S1, S2:

member (Union S1 S2)

rather than in terms of some direct "union" of the zoetic SAi, i.e.

union SA1 SA2

The problem is that aside from the inconvenience of having to retain inert data whilst programming in "zoetic mode", it's not guaranteed that in a non-trivial program, constructions Si would be visible in all contexts in which zoetic animations SAi are required.

Note in passing that this inaccessibility of constructions is not the case with pure zoetic animations: the identity

ifoldT D C1 … Cn = D

means that as long as all the constructors are available, the original construction can be recovered from the zoetic animation, e.g. for zoetic binary trees "TAi = ifoldBT Ti", a new binary node can be formed "ifoldBT (Branch (TA1 Null Leaf Branch) (TA2 Null Leaf Branch))" – inconvenient and contingent (upon availability of all the constructors), but possible nonetheless.

4.2.2 Generators. Thus, in order to be able to deal with impure zoetic animations, we seek "generators", e.g. for sets say *empty, single* and *union* – counterparts to constructors but which will build zoetic animations (from nested zoetic animations, where appropriate), rather than building constructions.

The required behaviours of these generators may be captured by an equation schema which serves as their specification:

```
member (Ci opdsC ...) = (Gi opdsG ...)
```

where

- Ci are the constructor functions for the "Set" type, and "(Ci opdsC …)" are constructions of sets (following conventions as above);
- Gi denote the generators corresponding to Ci
- "(Gi opdsG …)" denote application of Gi to operands (possibly other zoetic animations, e.g. in the case of recursive types) to construct zoetic sets.

Thus, the specification for sets is:

```
member Empty = empty
member (Single X) = single X
member (Union S1 S2) = union SA1 SA2
```

where as before, SAi are the zoetic counterparts of structures Si.

4.2.3 Derivation of Generators. Now, using the specific identity that for any type T

ifoldT D C1 … Cn = D

i.e.

ifoldT (Ci opdsC ...) C1 … Cn = (Ci opdsC...)

then the left-hand-sides of the specifying equations, which have the form "*member (Ci opdsC ...)*", may be transformed as a result into:

member (ifoldS (Ci opdsC ...) C1 … Cn)

Similarly, using the more general identity that for any type T

ifoldT (Ci opdsC ...) X1 ... Xn = (Xi opdsX ...)

the right-hand-sides which have the form *"(Gi opdsG ...)*" may be transformed into into "*ifoldS (Ci opdsC ...) G1 ... Gn*". (Note carefully that the precise detail of the "opdsG" is not a matter of specific concern at this point of the development, and will practically be obvious in each case, as exemplified below.)

The instances of the specifying equation schema (one for each different set constructor "Empty", "Single" and "Union") are thus transformed into

```
member ((ifoldS Empty) Empty Single Union)
        = ifoldS Empty empty single union
member ((ifoldS (Single X)) Empty Single Union)
        = ifoldS (Single X) empty single union
member ((ifoldS (Union S1 S2)) Empty Single Union)
        = ifoldS (Union S1 S2) empty single union
```

Because these now together cover all cases of the set construction "(Ci …)" for constructors "Empty", "Single" and "Union", they can be consolidated (where S is further shorthand for the set construction) into

```
member ((ifoldS S) Empty Single Union) =
        (ifoldS S) empty single union
```

At this point, "inverted fusion" for sets is directly applicable- in order to make the above equation hold, all that is necessary is

```
member Empty = empty
member (Single X) = single X
member (Union S1 S2) =
        union (member S1) (member S2)
```

From these requirements and the equations for "member" we may obviously derive definitions for the impure generators:

```
empty elt = False
single x elt = x==elt
```

The derivation of the third generator

```
union s1 s2 elt = s1 elt or s2 elt
```

requires however further explication. Noting that the forms "member Si" denote the desired impure zoetic sets, proceed

union s1 s2
= (as required by fusion)
member (Union S1 S2)
= (following "member")
\ elt → member S1 elt or member S2 elt
= (re-expressing "member Si" as the corresponding zoetic set)
\ elt → s1 elt or s2 elt

as required. Note how the operands of generator "union" are, correctly, the zoetic versions of sets.

To summarise, because both original inert sets and zoetic sets can be expressed in terms of (inverted) folds, the generators of zoetic sets can be synthesised by fusion from the defining equations of the underlying operation ("member") on the inert sets.

The generality of the above treatment of sets should convince that the technique for deriving generators of impure zoetic data structures is generally-applicable. In summary, proceed:

- specify the structure of the type ("signature");
- select and specify the selector operation O on the type that provides the characteristic behaviour of the zoetic data;
- to the identity "O (ifoldT D C1 … Cn)" = "ifoldT D G1 … Gn", apply fusion to relate Gi to defining equations for O on constructions from corresponding Ci;
- solve these relations to yield defining equations for Gi (provided that solutions exist – i.e. when the selector is expressible, as frequently is, in terms of a single "fold").

4.3 Generators for Pure Animations

Now that derivation of generators has been achieved for impure zoetic animations, we similarly show how they can be derived for pure zoetic animations. These generator definitions are straightforward, noting that account must be made taken for the fact that nested structures will be the zoetic version. For example, for lists, we seek generators "nil" and "cons" corresponding to the obvious counterparts. From the specifications

nil
=ifold Nil
= \ op b → b

and

cons x (ifold xs)
= ifold (x : xs)
= \ op b → op x (fold xs op b)

we can thus derive defining equations

- Lists:

```
nil op b = b
cons x xs op b = op x (xs op b)
```

Generators for the other "pure" types considered above follow similarly.

- Booleans:

```
true t f = f
false t f = f
```

- Natural numbers:

```
zero s z = z
succ n s z = s (n s z)
```

- Binary trees:

```
null n l b = n
leaf elt n l b = l elt
branch sub1 sub2 n l b = b (sub1 n l b) (sub2 n l b)
```

Note how synthesis of these "pure" generators proceeds by simple expansion of the relevant inverted folds, compared to fusion in the case of impure generators.

The earlier identities in the applications of "ifold" to data also have counterparts with respect to generators. In particular, for a type T with generators Gi,

```
ifoldT D G1 ... Gn = ifoldT D
```

5. Totally Functional Programming in Context

The validity and potential of Totally Functional Programming is supported by identifying its implicit appearance and connections to other themes in computer science. The simplification of programming by obviating the need to interpret inert, symbolic data through the use of zoetic representations already appears in the following contexts.

5.1 History of Programming with Zoetic Data

Some simple applications exemplify the style of programming with zoetic data that we contend is revealed by the (inverted) fold-based approach.

The original "Church numeral" representations of natural numbers in the lambda-calculus [6] were in terms of the generators "zero" and "succ" above. Thus, a natural N would be represented as N-fold composition; in other words, the computation inherent in a number N, as reflected in the zoetic representation, is iteration. Examples of simplified zoetic programming in this representation include definitions of basic arithmetic operations:

- Natural numbers:

```
add n1 n2 = n1 succ n2
mult n1 n2 = n1 (add n2) zero
```

These definitions apply the generators (not constructors) "succ" and "zero" and yield zoetic animations rather than data.

The same is true for other types similarly, yielding zoetic data by using zoetic animations and applying generators rather than constructors throughout, e.g.:

- Lists:

```
sum l = l (+) zero
l1 ++ l2 =l1 cons l2
```

- Booleans:

```
not b = b false true
and b1 b2 = b1 b2 false
or b1 b2 = b1 true b2
```

- Binary trees:

```
numnodes t = t zero (\elt → succ zero) add
concatnodes t = t nil (\ elt → cons elt nil) (++)
```

In general, the pure zoetic animations of data and their definition in terms of generators have been known as long as Reynolds' second-order polymorphic typed-lambda-calculus [7], which incidentally provides a type system at least theoretically adequate to the demands of zoetic programming. Likewise, impure zoetic animations such as characteristic predicates for sets are well-known (see also "Wider Applications of Zoetic Data" below). This history does not however extend to our presentation of the connections between pure and impure zoetic animations in terms of folds, and in particular of their significance in the context of a comprehensive animation-based programming style and its motivation from a programming-as-language-extension perspective.

5.2 Wider Applications of Zoetic Data

As well as the extensive use of pure zoetic animations in functional programming (albeit disguised as folds), there are some significant examples of impure zoetic animations. In "combinator parsers" [8] the usual combination of a grammar (or equivalent representation such as a parse table) and its interpeting "parsing engine" are replaced by a collection of parsers, one for each nonterminal symbol in the grammar. Context-free operations of grammar concatenation and alternation are implemented by appropriate higher-order functions on parsers. Compare this with the conventional situation where a grammar is a static data structure awaiting interpretation by a parsing engine. Similarly, exact real arithmetic [9] may be achieved when a real number is represented by a function which computes a real to any required rational precision. Also, programmed graph reduction [10] of functional languages represents functions not as graphs into which substitutions are to be performed, but as programs which construct the substituted graphs.

5.3 Relationship with Object-Orientation

There is an apparent positive connection between zoetic programming and some of the origins of OOP, in that one conception of methods on objects is as portals to which messages are sent and acted upon by the object, rather than as operations that manipulate the object. For example, in the zoetic definition of "add" above, a number m takes two messages – an operation "succ" and another number n, and replies with m+n. For a whimsical yet extensive development of the analogy between functional programming and message-passing, see Smullyan [11].

On the other hand, zoetic objects as we have characterised throughout this paper admit only a single extractor method besides generators. Zoetic animations correspond to objects upon which a single method is defined – the applicative behavior of the object, i.e., the behavior inherent to the data that the zoetic animation embodies, *is* the method. Consider for example how the single "member" method on sets was subsumed into the characteristic predicate representation. For natural numbers, the single method is iteration, etc. for the other types considered. In contrast, OOP permits numerous methods to be defined on objects. Our current aspiration for the reconciliation of TFP with OOP depends the hope of being able to demonstrate that for a genuinely-cohesive class, the various apparently-disparate methods can all be viewed as instances of some higher-order characteristic behavior, of course in turn defined in terms of folds as above.

6. Summary and conclusion

While the relationships between "folds" and datatypes have been known to some degree since the origins of the lambda-calculus, our approach goes further in that:

- generally, the use of folds in located within a comprehensive context that demands the replacement of symbolic data by functions;
- specifically, we've shown how to synthesise functional representations for types that require characterisations beyond those representable by pure folds.

Our zoetic approach develops conventional functional programming in two senses of the word "total": (i) it tends to eschew data, so that programming tends to be "totally" concerned with functions; (ii) it tends also to avoid recursion, i.e. such programs (or components) tend to be total rather than partial functions. Ultimately, a convergence with analog computing (where computations are likewise "enlivened", i.e. structured around the behaviours of their components) may be discovered and exploited.

A further priority for development of TFP would appear to be discovery of a suitable practical type (checking) system. Relatively simple functions seem to require complex regimes that elude the facility of type inference [12]. While some further work [13] does not seem directly applicable to TFP, it suggests by example that some sort of compromise may be achieved.

We are grateful to our co-workers for their influence on the above, notably Ian Peake and Sean Seefried. Some of our earlier work on this project was reported in [14].

7. References

[1] E.W. Dijkstra, Goto Statement Considered Harmful, *Comm. ACM, vol. 11 no. 3*, 1968, 147-148.

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

[3] G. Hutton, A Tutorial on the Universality and Expressiveness of Fold, *Journal of Functional Programming, vol. 9, no. 4*, 1999, 355-372.

[4] http://www.haskell.org

[5] P.A. Bailes, The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design), *Proc. 1986 Austrln. Software Eng. Conf.*, Canberra, 1986, 14-18.

[6] H.P. Barendregt, *The Lambda Calculus - Its Syntax and Semantics* (North-Holland, Amsterdam, 1984).

[7] J.C. Reynolds, Three approaches to type structure, *Mathematical Foundations of Software Development, LNCS Vol 185* (Springer-Verlag, 1985).

[8] G. Hutton, Parsing Using Combinators, *Proc. Glasgow Workshop on Functional Programming*, Springer, 1989.

[9] H. Boehm & R. Cartwright, Exact Real Arithmetic: Formulating Real Numbers as Functions, in D.A. Turner, (ed.), *Research Topics in Functional Programming* (Addison-Wesley, 1990).

[10] S. Peyton Jones, *The Implementation of Functional Programming Languages* (Hemel Hempstead: Prentice-Hall International, 1987).

[11] R. Smullyan, *To Mock a Mockingbird* (Alfred A. Knopf, New York, 1985).

[12] J.B. Wells, Typability and Type Checking in the Second-Order Lambda-Calculus are Equivalent and Undecideable, *Logic in Comp. Sci.*, 1994, 176-185.

[13] M.P. Jones, First-class Polymorphism with Type Inference, *Proc. Symp. on Princ. of Prog. Langs*, Paris, 1997.

[14] P.A. Bailes, C.J.M. Kemp, I.D. Peake, & S. Seefried, Why Functional Programming *Really* Matters, *Proc. 21st IASTED International Multi-Conference on Applied Informatics (AI 2003)*, Acta Press, 2003, 919-926.