Fundamentals, Prospects and Challenges for Totally Functional Programming Style

Paul Bailes, Leighton Brough, Colin Kemp
School of Information Technology and Electrical Engineering
The University of Queensland,
St Lucia, QLD Australia
{p.bailes, l.brough, c.kemp}@uq.edu.au

***Abstract*—General recursive definitions contribute to the complexity of programming. This complexity could be reduced by reliance on established, well-understood programming patterns. Catamorphism-based recursion patterns simplify programming with little practical loss in expressive capability compared to general recursion, including the capability of defining new recursion patterns. Partial application of catamorphisms, sub-catamorphic recursion patterns and methods to symbolic data allows a comprehensive replacement of symbolic data with functional, or what we describe as "zoetic", representations that inherently adopt the benefits of catamorphism-based programming. The considerable promise of this "Totally Functional" style confronts us with some exciting technical challenges.**

Keywords-component; Catamorphism, Fold, Functional, Recursion.

I. INTRODUCTION

We contend that software is unduly complicated by the pervasive need to program interpreters for the computations inherent to symbolic data. By using instead functional representations that embody the fusion of characteristic interpretations into data, this pervasive complication can be minimized if not avoided, and programming thus significantly simplified.

Our essential argument develops in logical sequence as follows:

- recursion patterns such as list "foldr", which generalize to catamorphisms on regular recursive datatypes, suffice to express and simplify a very wide range of common recursive definitions;
- other useful and simplifying recursion patterns are also definable in terms of catamorphisms;
- catamorphisms thus embody practically as well as theoretically (in terms of initial algebra semantics) the behaviours characteristic to abstract data types;
- partial application of catamorphisms to typical symbolic representations of data yield functional representations that inherently possess these characteristic behaviours, i.e., a kind of liveness which we describe as "zoetic" from the Greek "zoion" meaning "animal" (as in "zoology");
- partial application of behaviours that are more specialized than the generic catamorphism, but are definable inevitably in terms of catamorphisms, also yield zoetic data;
- programming with zoetic data simply involves their application to appropriate operands (just as with recursion patterns), rather than also having to program with explicit recursion the characteristic behavior of the datatype;
- creation of zoetic data can be effected by generator functions which are the derived counterparts of symbolic data constructors;
- this enables a new style of programming (which to emphasise its distinctiveness from an earlier related development of "Total Functional Programming" [1] we call "Total**ly** Functional Programming", or TFP), in which a comprehensive supercession of symbolic data by functional representations can be achieved;
- while the comprehensiveness of the foregoing program is unprecedented, important aspects of it are discernable in (and thus validated by) related fields of computer science.

The presentation of the argument in this paper follows the above sequence.

The consequent comprehensive replacement of symbolic data by functions requires first-class functions, hence we implicitly adopt functional programming [2] and functional languages. We choose Haskell [3] for purposes of illustration.

II. CATAMORPHIC PROGRAMMING

An approach to programming based entirely on canonical recursion patterns known as "catamorphisms" [4] is beneficial, viable and self-sufficient. Catamorphisms are more familiar as the list "reduce" or "foldr" functions of functional programming, but apply to all regular recursive types.

A. General Recursion Too Complex

Recursion patterns simplify and clarify programming, compared to the use of general iteration/recursion. Consider the case of recursively defining basic arithmetic operations on the simplest recursive datatype, of Natural numbers, in Fig. 1. This example exposes some key aspects of Haskell as follows:

- declaration of datatypes (e.g., Nat) in terms of constructor functions (e.g., Zero and Succ) and their operand types where appropriate (i.e., Nat, thus defining a recursive type);

 ISBN: 978-1-61208-367-4

```
data Nat = Zero | Succ Nat

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

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

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

Figure 1. General recursive renditions of basic arithmetic operations.

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

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

Figure 2. Catamorphic renditions of basic arithmetic operations.

- definition of functions by recursion equations;
- branching by pattern matching on function arguments;
- function application by juxtaposition of operator and operand(s).

(Further key novelties of functional languages and Haskell in particular will be explained as introduced in examples below.)

In this general recursive rendition of arithmetic operations, the following deficiencies are apparent.

Apart from the suggestive naming of the type and its constructors, there is nothing in the definition that compels treatment of members of the type as naturals, or indeed numbers of any kind;

Instead, the obvious isomorphism between the concrete members of Nat and the abstract natural numbers needs to be implemented by an implicit interpreter that converts symbols into actions (in this case, iterative applications of other functions);

A programmer needs to repeat the implementation of this interpreter at each usage of Nat entailing not just extra effort but the risk of inconsistent implementations leading to inconsistent (erroneous) behavior.

Using however the "catamorphism" recursion pattern on Nat - cataN - the rendition becomes that of Fig. 2 which significantly remedies the above deficiencies, in that a uniform interpretation of the symbolic data is provided - i.e., cataN - which moreover derives directly from the type definition.

B. Catamorphisms as Practical Basis

The catamorphic pattern defined on Nat above generalises for regular recursive types. For example, the catamorphism - cataL - for (polymorphic) lists is as in Fig. 3. Note how in Haskell the type polymorphism on type List is signified by parameterization on list element type 't'.

```
data List t = Cons t (List t) | Nil

cataL Nil o b = b
cataL (Cons x xs) o b = o x (cataL xs o b)
-- versus
foldr op b [] = b
foldr op b (x:xs) = op x (foldr op b xs)
```

Figure 3. Catamorphisms and operations on lists.

```
sumR Nil = 0
sumR (Cons x xs) = x + sumR xs
-- versus
sumC xs = cataL xs (+) 0

appendR Nil ys = ys
appendR (Cons x xs) ys =
   Cons x (appendR xs ys)
-- versus
appendC xs ys = cataL xs Cons ys
```

Figure 4. List processing examples.

Observe also that (aside from a reordering of the usual presentation of operands) cataL is exactly the familiar "foldr" of functional programming (also known as "reduce").

The reader will also observe that just as with Nat above, operations on lists may be programmed using the uniform interpretation offered by cataL applied to other operations and data pertaining to the specific applications. See Fig. 4 for a comparison of explicit recursive vs. catamorphic definitions of some basic list processing functions. (Note how in Haskell the form "(θ)" denotes the function represented by operator 'θ', in this case binary addition represented by '+'.)

What make catamorphisms attractive as a practical as well as a theoretical basis for programming are their properties as follows:

- generality - existence for all regular recursive types, not just Nats or Lists
- expressiveness - sufficient to define at least any function provably-terminating in second-order arithmetic [5], i.e., practically-speaking any reasonable function other than a Universal Turing Machine or other programming language interpreter;
- essentiality - their embodiment of the initial algebra semantics [6] of the respective underlying datatypes, as the unique homomorphisms that define the applicable notion of initiality itself;

C. Catamorphisms as Pragmatic Basis

There are however other recursion patterns that appear to be necessary for the natural solution of programming problems. For example, compare the catamorphic renditions in Fig. 5 of the "insert" and "reverse" operations with their definitions in Fig. 6 using respectively the paramorphic [4] and "left fold" [2] recursion patterns. (N.B. our adoption henceforth of customary concrete syntax for the List type.)

 ISBN: 978-1-61208-367-4

```
-- insert element into ascending list
insert e xs =
  fst $ cataL xs
  (\x (exs,xs) ->
    (if e<x then e:x:xs else x:exs, x:xs)
  )
  ([e],[])

-- reverse order of list elements
reverse xs =
  cataL xs
  (\x xs' -> (\rxs -> xs' (x:rxs)))
  (\rxs -> rxs)
  []
```

Figure 5. Catamorphic definitions of "insert" and "reverse".

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

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

-- definitions of new recursion patterns

-- like cataL but op also has list tail xs
paraL (x:xs) o b = o x (paraL xs o b) xs
paraL [] o b = b

-- like cataL but op grouped from left
lfold (x:xs) o b = lfold xs o (o b x)
lfold [] o b = b
```

Figure 6. Alternative definitions of "insert" and "reverse".

```
paraL xs o b =
  fst $ cataL xs
  (\x (pxs,xs)->(\o b->(o x pxs xs, x:xs))
  (b, [])

lfold xs o b =
  cataL xs
  (\x lxs -> (\b -> lxs (o b x)))
  (\b -> b)
  b
```

Figure 7. Catamorphic definitions of other recursion patterns.

Important new Haskell features used here are as follows:

- anonymous "lambda" functions, of the form `(\ arguments -> body)`
- built-in list datatype, with constructors ':' (for Cons) and "[ ]" (for Nil);
- n-tuple data, with elements selected by pattern-matching or by selector functions (e.g., "fst");
- low-precedence function application denoted by '$'.

What allows us to continue to treat catamorphisms as a basis in the face of the above is that these other recursion patterns can be synthesized from catamorphisms without recourse to general recursion. The recursion patterns (such as paraL, lfold, etc.) can be defined using abstractions (higher-order, as needed) from the definitions of the methods (such as insert, reverse, etc.), e.g., as in Fig. 7.

D. Recursion Pattern/Application Hierarchy

The consequence of the above is that all the reasonable methods (on regular recursive datatypes, such as natural numbers, lists, trees, etc.) we would want to program, and all the recursion patterns besides catamorphisms that we would want to use to program them, can be derived in a hierarchical manner, starting from catamorphisms and supplying instantiating operands at each level of refinement.

For example, for lists:

- the root, catamorphism level of the hierarchy is represented by cataL;
- the intermediate, recursion pattern level of the hierarchy is represented by patterns derivable from the root, e.g., paraL; lfold; etc.;
- the lowest, application level of the hierarchy is represented by actual list operataions, e.g., length; append; insert; reverse; etc.

Note that members of the hierarchy at all levels are directly accessible from the root catamorphism, in some cases more conveniently (e.g., length) and in some cases less so when one of the intermediate recursion patterns is more convenient (e.g., reverse). In particular, the identity property of catamorphisms is that application of the catamorphism for a type to the constructors of that type returns the catamorphism, e.g., as follows:

```
cataN n Succ Zero = n
cataL xs (:) [] = cataL xs
```

III. ZOETIC DATA

The foregoing catamorphism-based recursion pattern approach to programming enables us to bypass completely symbolic data, and their interpretation. In the end, we directly construct "zoetic" representations of data, i.e., as functions, rather than the usual symbolic forms.

A. Catamorphic Zoetic Data (CZD)

CZD are the basic kind of zoetic data. They are formed by the partial application to symbolic values of the standard interpretation of their datatypes. The standard interpretation of each datatype remain exactly as exposed above, i.e., its catamorphism. As a result, each CZD is a function that implements that catamorphism on the underlying symbolic datatype.

Thus, the advantage of CZD is that their usages no longer require any interpretations as reflected by explicit recursive definitions or by the explicit application of the interpreter for the type, i.e., its catamorphism. Instead, the CZD are simply applied to appropriate catamorphism operands. For example, compare the above definitions of arithmetic operations to

```
-- zoetic naturals
zero = cataN Zero
one = cataN (Succ Zero)
-- etc

-- zoetic arithmetic operations
addz za zb = za succ zb
mulz za zb = za (addz zb) zero
expz za zb = zb (mulz za) one
```

Figure 8. Zoetic natural numbers and operations.

those on natural number CZD as in Fig. 8. Observe how zoetic naturals are simply the partial applications of cataN to the symbolic values of type Nat.

A final key Haskell feature found in the above is partial application of "curried" functions. For example, addz can equally be thought of as a function of one parameter "za" that returns a function of a further parameter "zb", as well as a function of the same two parameters. Thus, e.g., application of addz to the "zb" parameter of mulz denotes a function that will add "zb" to its further actual parameter.

Now, we can define generators, i.e., the zoetic counterparts of symbolic data constructors but independent of them. For example, from specifications as in Fig. 9, we derive the respective zoetic counterparts zero and succ of Zero and Succ as in Fig. 10 (likewise for cons and nil). Note that the identity property for catamorphisms and symbolic data constructors applies for CZD and their generators, e.g., as per the identities as in Fig. 11.

```
-- generally
zn = cataN n

-- specifically
zero s z = cataN Zero -- as above
succ (cataN n) = cataN (Succ n)
```

Figure 9. Specifications of zoetic natural number generators.

```
zero f x
= cataN Zero f x
= x

succ zn f x
= succ (cataN n) f x
= cataN (Succ n) f x
= f (cataN n f x)
= f (zn f x)

-- similarly derivable
cons x zxs o b = o x (zxs o b)
nil o b = b

-- etc. for other types
```

Figure 10. Zoetic natural number generators.

```
succ zn succ zero = succ zn
zero succ zero = zero

cons z zxs cons nil = cons z zxs
nil cons nil = nil
```

Figure 11. Identities for CZD.

Just as with zoetic arithmetic, zoetic list processing also entails simple, non-interpretive provision of relevant catamorphic operands, e.g., as follows.

```
zappend zxs zys = zxs cons zys
zsum zxs = zxs addz zero
```

B. Subcatamorphic Zoetic Data (SZD)

The interpretation of symbolic data is not always given by a catamorphism, but maybe by some other method that can be defined catamorphically, i.e., found below catamorphisms in the recursion pattern/application hierarchy, hence "subcatamorphic".

For example, in Fig. 12 the method "memb" interprets binary trees as sets, with constructors Nd, Lf and Tip respectively signifying set union, singleton and empty sets. However, within the catamorphic programming paradigm essential to TFP, these other methods (exemplified here by memb) will be expressible as catamorphisms, e.g., as in Fig. 13.

As with CZD, we form zoetic data by the partial application to the symbolic data of the interpreter for the required characteristic behaviour. In this case, the partial application "memb bt" (for some bt :: Bt) yields a function that tests if some putative element e is actually a member of the set represented by bt. That is, the SZD form of a set is the familiar characteristic predicate representation.

```
data Bt t = Nd (Bt t) (Bt t) | Lf t | Tip

memb (Nd t1 t2) e = memb t1 e || memb t2 e
memb (Lf x) e = x==e
memb Tip e = False
```

Figure 12. Trees as sets.

```
memb s e =
  cataBt s
    (\t1' t2' -> t1' || t2')
    (\x -> x==e)
    False

-- catamorphism on Bt
cataBt (Nd t1 t2) n l t =
  n (cataBt t1 n l t) (cataBt t2 n l t)
cataBt (Lf x) n l t = l x
cataBt Tip n l t = t
```

Figure 13. Set membership as a catamorphism.

 ISBN: 978-1-61208-367-4

```
memb s e =
  cataBt s
    (\s1 s2 e -> s1 || s2)
    (\x e -> x==e)
    (\e ->False)
    e
```

Figure 14. Catamorphic set membership with closed terms.

```
memb s =
  cataBt s
    (\s1 s2 e -> s1 || s2)
    (\x e -> x==e)
    (\e ->False)
```

Figure 15. Catamorphic set membership as a characteristic predicate.

```
union s1 s2 e = s1 e || s2 e
single x e = x==e
empty e = False
```

Figure 16. Declarations of generators for zoetic sets.

Further, if we write the catamorphism operands as closed terms as in Fig. 14, then, as a corollary of the identity property of catamorphisms, these closed operands serve as generators of characteristic predicates.

First, eta-reduction of the definition of memb exposes the zoetic set/characteristic predicate clearly as in Fig. 15. Then recognizing that the significance of the identity property is that catamorphism operands serve as constructor replacements, we see that catamorphism operands are inherently generators of whatever is the result of the catamorphism, in this case the zoetic set. So, finally rewriting the above operands in more convenient equational format gives the generator declarations of Fig. 16. The same technique applies for any SZD, subject of course to the condition that the characteristic method is definable as a catamorphism (which as we have seen is practically always).

C. Recursion Patterns as SZD

Just as applications such as "memb" give rise to SZD, so do the recursion patterns found below catamorphisms. For example, partial applications of the form "lfold xs" give rise to a class of list-like SZD, but which instead of having the catamorphic/foldr behavior of list CZD, behave as "left folds" with the binary operator 'o' grouped from the left.

Further, just as with zoetic sets above, when we express the catamorphic definitions of recursion patterns with closed operands e.g. as in Fig. 17, these operands are also effective as generators. Continuing the example, first eta-reduce as in Fig. 18 which exposes the zoetic left-folding list as an identity between a partial application of the lfold method and a catamorphism. Then we simply read off the operands to the catamorphism and re-present them as generator declarations as in Fig. 19.

```
lfold xs o b =
  cataL xs
  (\x lxs -> (\o b -> lxs o (o b x)))
  (\o b -> b)
  o b
```

Figure 17. Left fold as a catamorphism with closed terms.

```
lfold xs =
  cataL xs
  (\x lxs -> (\o b -> lxs o (o b x)))
  (\o b -> b)
```

Figure 18. Left fold as catamorphism partial application.

```
lcons x lxs o b = lxs o (o b x)
lnil o b = b -- NB same as nil CZD
```

Figure 19. Declarations of generators for left-folding zoetic lists.

D. Zoetic Data Hierarchy

The hierarchy of zoetic data (CZD and SZD) naturally parallels that of catamorphisms, other recursion patterns, and catamorphic methods as detailed above, in which descent in the hierarchy from most general CZD to more specialized SZD in achieved by application to appropriate operands.

For example, from a zoetic list zxs we can first derive the variant lzxs with the same elements in the same sequence but with left-folding behavior, by applying zxs to the left-folding zoetic list generators thus:

```
lzxs = zxs lcons lnil
```

Next, we can calculate the reverse of zxs by applying lzxs to the appropriate operands to left-fold as follows:

```
rzxs = lzxs (\rzxs x -> cons x rzxs) nil
```

Note that the zoetic nature of the resulting list is achieved by use of the zoetic list generators cons and nil in the above, rather than symbolic list constructors (:) and [ ]. That is, if conventional lists were the required result, we would have written instead the following:

```
rxs = lzxs (\rxs x -> x:rxs) []
```

(Further note how in this case the operands to the left-folding zoetic list lxzs are precisely those given to lfold above in order to reverse a conventional list.)

If desired, we can define a self-contained reverse operation, by application of successive sets of (sub-) catamorphism operands in stages reflective of the above, as per Fig. 20. A one-stage definition of zrev in Fig. 21 echoes the direct definition of list reversal as a catamorphism further above. This version however loses some of the transparency of the two-stage definition that results from being able to express zrev in its more natural left-folding form.

Finally, it one exists, we can always recover the symbolic form of a zoetic datum by applying it to the symbolic constructors, e.g., as in Fig. 22.

 ISBN: 978-1-61208-367-4

```
zrev zs =
  -- start with zs
  zs
  -- next transform into left-folding list
  lcons lnil
  -- finally give left-folding operands
  (\rzxs x -> cons x rzxs) nil
```

Figure 20. Staged defintion of list reverse.

```
zrev zs =
  zs
  (\x zxs'->(\rzxs -> zxs' (cons x rzxs)))
  (\rzxs -> rzxs)
  nil
```

Figure 21. Direct catamorphic definition of list reverse.

```
-- an identity, not a function definition
cons 'a' (cons 'b' nil) (:) [] = "ab"
```

Figure 22. Recovery of symbolic from zoetic data.

IV. TOTALLY FUNCTIONAL PROGRAMMING

Supported by the techniques presented above, our key key proposition in Totally Functional Programming (TFP) is the combination of three complementary factors.

First, every data type has a characteristic behavior (for pure structures such as naturals, lists, trees, etc. it is their catamorphism; for more specialised types it is the characteristic method for the type e.g., for sets it is the memb(er) function);

Second, the complexity of conventional programming derives significantly from the need to program the interpretation of these inherent behaviours from symbolic datatypes, which are typically intertwined with application-specifics (e.g., the explicit recursive definitions of arithmetic operations far above);

Finally, direct zoetic representations of data that embody these behaviours are specified as the partial applications of characteristic methods to the symbolic representations, and can be exploited simply by application to the further operands of the methods. The feasibility of TFP is enhanced by direct generation of CZD and SZD without having explicitly to apply the characteristic methods to symbolic data.

A potential criticism of the key proposition of TFP is that whereas it posits a single behavior or characteristic method for each datatype, instead multiple methods are normal in programming. Our response is that the zoetic data hierarchy for each type adequately expresses any need for multiple behaviours: at the summit of the hierarchy is the catamorphism, from which all other behaviours can be derived; more specific behaviours can be found lower in the hierarchy. The designer of a zoetic datatype is thus free to choose a relatively general (= more methods) or specific (= fewer methods) behavior as circumstances require.

V. RELATED WORK

Aside from our own work (recently [7][8][9]), some aspects of TFP have been presaged (and therefore in a sense pre-validated) by others. However, none of these propose the comprehensive replacement of symbolic data with functional/zoetic representations as we do.

A. TFP in Functional Programming

Our conception of TFP can already be discerned in various aspects of functional programing: Church numerals [10] are CZD for the Nat type above; combinator parsers [11] are SZD for context-free grammars.

B. Turner's Total Functional Programming

Turner's already-cited [1] related conception of TFP has a common basis with ours in the avoidance of general recursion in favour of recursion patterns such as catamorphisms (and additionally anamorphisms - see below), but does not eschew symbolic data as we do.

C. Language Design

The history of language design can be thought of as a progressive retrofitting of "Church" concepts into a "Turing" context. TFP culminates that process by the complete replacement of Turing-style interpretation of symbolic data with Church-style direct definitions of (higher-order) functions. Some highlights of this process with particular relevance to TFP are as follows (in reverse chronological order).

Backus [12] repudiated general recursion for a fixed set of "combining forms" (including list catamorphisms), but without generalization to other types.

Dijkstra's [13] emphasis on fixed control structures rather than arbitrary control flows ("goto" statements) can be thought of a similar in sprit to our (and Turner's and Backus') repudiation of general recursion.

But long before, Backus equipped FORTRAN with the catamorphism on natural numbers, in the form of the DO-loop. Our TFP of course offers the programmer significantly more facility than DO-loop programming.

It is evident from this paper that modern functional languages (such as Haskell) at least encourage TFP. However, in order to avoid surprising limitations on zoetic operations, it will be necessary to adopt more powerful type systems (see Future Directions/Type-checking below). Also, in order to dispense with symbolic data completely, it will be necessary to handle infinite structures zoetically (see Future Directions/Codata and Corecursion below).

VI. FUTURE DIRECTIONS

TFP's promise is also a stimulus to address some key technical challenges, in the following respects.

With respect to computer science education: the simplicity of recursion-pattern-based programming (no need to program iteration or recursion; just "complete the blanks" by supplying catamorphic recursion patterns with the appropriate operands as in the examples above) suggests applicability to introductory programming teaching.

 ISBN: 978-1-61208-367-4

```
anL nxt end seed =
   if end seed then []
   else
      let (nxtelt, nxtseed) = nxt seed
      in nxtelt : anL nxt end nxtseed
evens =
  anL (\s->(s+2,s+2)) (\_->False) 0

odds =
  anL (\s->(s+2,s+2)) (\_->False) 1

fibs =
  anL
  (\(fa,fb)->(fa,(fb,fa+fb)))
  (\_->False)
  (0,1)
```

Figure 23. Anamorphic defintions of infinite streams.

Regarding type-checking: the convenient type inference found in Haskell and other modern functional languages does not accept some simple CZD (arithmetic on Church numerals). It's not yet clear if the existing candidates for the necessary more complex type systems are prohibitively inconvenient [14].

Regarding formal methods: just as catamorphisms (and CZD) possess more specific (and useful) laws than induction [15], what kind of more specialised laws are derivable among more specialised zoetic data (i.e., SZD)? With respect to course code refactoring: if zoetic data represent a clearer way to write programs, they should equally represent a good refactoring target, as indicated by some potentially useful results already [16][17].

Finally, regarding processing of infinite structures: catamorphisms are total functions on finite structures ("data"), but for practical computing, processing ("corecursion") of (potentially) infinite structures ("codata") is clearly necessary (e.g., a stream of transactions against a database; events to which a real-time operating system has to respond; etc.). The clear path to a solution [1] entails "anamorphisms" [4], as the categorical dual to catamorphisms, to provide the effective basis for zoetic representations of codata. For example, the anamorphism on lists ("anL" a.k.a. "unfold") can be used to define (infinite) streams, as in Fig. 23. Note that in Haskell, the '_' denotes an ignorable formal parameter, useful in defining constant functions. We are however yet to develop a presentation of anamorphism-based zoetic codata in the same comprehensive way that we have achieved for catamorphism-based zoetic data.

VII. CONCLUSIONS

Totally Functional Programming has the promise to fulfill the prospects of functional programming in several ways. Fundamentally, the essence of functional programming - "first class" functions - is exploited to simplify programming by bypassing pervasive interpretation with zoetic data that encapsulate the behaviours essential to data.

Higher-order functions are also instrumental in realizing the pragmatics of TFP - for each type, a hierarchy from general (CZD) to specific (SZD) entities exists, the specialization relationship being implemented by application to catamorphism operands.

Finally, as signified by the "front of stage" role it gives to zoetic data (compared to their hitherto relegation as theoretical curiosities as "Church" data representations), TFP completes an important stage in the last sixty or so years of work of restoring the Church perspective programming into the otherwise Turing-dominated worldview.

REFERENCES

[1] D. A. Turner, "Total Functional Programming", J. Universal Computer Science, vol. 10, no. 7, 2004, pp. 751-768.

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

[3] The Haskell Programming Language, http://www.haskell.org/haskellwiki/Haskell, retrieved: 11 August 2014.

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

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

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

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

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

[9] P. Bailes, L. Brough, and C. Kemp, "Higher-Order Catamorphisms as Bases for Program Structuring and Design Recovery", Proc. IASTED SE, 2013, pp. 775-782.

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

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

[12] J. Backus, "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs", Comm. ACM, vol. 9, 1978.

[13] E. Dijkstra, "Goto Statement Considered Harmful", Comm. ACM, vol. 11, 1968, pp. 147-148.

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

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

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

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

 ISBN: 978-1-61208-367-4