Approaching "Totally Functional Programming" through Rigorous Application of the Reuse Principle

Paul A. Bailes,

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

Abstract. The notion of reuse is fundamental to software engineering. One of the most pervasive reuse patterns that is most obviously (but not exclusively) accessible from the functional programming paradigm, is the family of functions on regular recursive types known as "fold". Partial applications of folds yield further reuse patterns, but changing the usual order of parameters yields even more interesting partial application reuse patterns. That is, by partially-applying folds to data, an animated or "zoetic" view of data is implemented. The conceptual essence of this approach, also known as "totally functional" programming, is that because there are inherent operations characteristic to data, why not represent data in terms of these operations rather than having to interpret the operations from the data each time the data are used. Partial applications of folds are only the simplest zoetic data: partial application of other operations/methods provides for more complex and realistic examples, including language processing. Despite the evident potential of totally functional programming, and its connection to some practical successes of functional programming, a number of challenges remain before it might be ready for widespread use outside the laboratory or classroom.

Keywords: Combinator, Fold, Functional Programming, Parsing, Reuse

1 Introduction

We have earlier [1, 2] advocated "Totally Functional Programming" (TFP) as an extreme style of Functional Programming (FP) [3] that exploits FP's defining capability – of treating functions as first class objects, in particular as the results of programmer-defined functions – in order to eschew entirely the use of symbolic data. The fundamental hypothesis underlying TFP is that symbolic data frequently if not invariably signify a characteristic computational behaviour, and consequently programs and programming with symbolic data are needlessly complicated by the need to include what amount to interpreters for the mini- or micro-languages embodied by symbolic datatypes. If symbolic data are instead represented by functions that directly represent the intended computational behaviours, then the need for programmers to write interpreters can be dispensed with.

As independent corroboration of our hypothesis, we contend in this paper that the

"reuse principle", when applied with rigour to typical patterns of programming on regular recursive data structures at least, leads inevitably to essentially the same conclusion as the TFP hypothesis, again that symbolic data are superfluous and can be replaced by direct functional representations of the computations they signify.

By way of illustration, consider the following definitions:

```
data Nat = Succ Nat | Zero

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)
```

which while familiar and apparently innocuous are in fact highly unsatisfactory in the following respects:

- the type "Nat" and its symbolic values "Zero", "Succ Zero", etc don't inherently denote the natural numbers, rather the explicit decisions reflected by the branching and recursive structure of "add" are what makes "Zero" behave like zero, and what makes "Succ a" behave like a+1
- in other words, these symbols are assigned meaning as naturals only a result of their interpretation by usage in operations "add", "mul", etc., the definitions of which each entail an embedded interpreter for the mini-language represented by the symbols of the "Nat" datatype.
- there are multiple such interpretations, the consistency of which is by no means guaranteed.

It is not too far-fetched to appreciate how these problems are applicable to programming with symbolic data types other than naturals, and indeed in general. This paper addresses the problems by successive application of the reuse principle:

- first, by isolating the interpretation of symbolic datatypes into single interpreters for each type;
- second, by replacing the resulting repeated application of these interpreters to symbolic data with the fusion of the interpreters with the symbolic constructors, leading to function representations of data;
- finally, by the extension of this technique to wider classes of datatypes beyond pure structures such as "Nat".

We acknowledge (see 6.5 Related Developments below) that we are hardly alone in advocating a programming style that exploits FP's capability for programmer-definable function-valued functions and thus functional representations for data. We claim to be contributing however on two counts: first, with this connection to the broad software engineering principle of Reuse as a further justification for the TFP approach; and second, with a comprehensive goal of eradicating the interpretation of symbols from programming, leading (as we shall see) to the exploitation of a wider class of functional representations than the other related developments may have contemplated.

2 Reuse Patterns in Recursion

A key insight in the development of programming methodology has been that common programming patterns can beneficially be encapsulated and applied. Imperative programming was thus "structured" in terms of control constructs [4], and the corresponding development in functional programming has been the identification of various recursion patterns [5] to replace direct recursive definitions.

A key recursion pattern is catamorphism [6], long-known to functional programmers as the list "foldr" function [7], even earlier known as "reduce" in APL, and definable (here, as generally throughout this paper, in Haskell [8]) as

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

Just as "fold" above captures a common pattern of recursion for lists, the same essential catamorphic pattern can be captured by variants of fold for other regular recursive datatypes.

For natural numbers as above, define:

```
foldn s z Zero = z
foldn s z (Succ n) = s (foldn s z n)
```

For booleans, define:

```
foldb t f True = t
foldb t f False = f
```

For binary trees

```
data BinT t = Null | Leaf t | Branch (BinT t) (BinT t)
```

define:

```
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)
```

The underlying commonality of these variants on list "foldr" as catamorphisms is rooted in category theory, and programming language technology has evolved to the stage whereby a single generic polytpic definition can capture the diversity of fold across different recursive datatypes [9].

While a feature of our TFP is that it supports functional views of data beyond fold, we will adopt fold as the default behavior for pure data structures because:

- fold is at least sufficiently expressive to serve as a basis for other practical recursion patterns (up to primitive recursion);
- generally the simplest choice among alternatives is preferable;
- specifically, fold has several useful calculational properties with implications for both efficiency and expressiveness. In particular, we will exploit the identity property to implement a kind of inheritance relationship between functional data.

3 Inverting Fold: Complementary Reuse Patterns of Partial Applications

Further to the key role of folds in enabling reuse of recursive patterns, the order of arguments to these folds combined with the opportunity for partial application provides a bias in favour of a reuse pattern whereby the various operation and value arguments to folds are reused, but applied to different data structures. A straightforward variant therefore of the various kinds of "fold" is to re-order the operands, the rationale being that whereas partial application of fold reuses the operations, partial application of inverted folds reuses the data.

Consider for example how, canonically-presented as above, partial application of "fold" simplify the abstraction of a particular data structure from a pattern of operations on the structure, e.g. for lists,

```
foldr O B
```

(for some specific O, B) is an abstraction from the data to which the "fold" is applied. Eta-conversion may make this more obvious:

```
\ xs -> foldr O B xs
```

Thus, specific partial applications all represent operations to be applied to some as-yet-unspecified structure, e.g.

```
-- sum of elements of any list yet to be supplied
fold (+) 0

-- apply f to each element of any list yet to be supplied
fold (\ x xs -> f x : xs) [ ]
```

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

```
\ op b -> foldr op b Xs
```

The same effect of this alternative abstraction may be achieved more simply directly by defining an "inverted" variant of list fold, say "lfold", which inverts operands of foldr such that

```
lfold xs op b = foldr op b xs
```

Thus "lfold 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 "tfold".

This alternative pattern of abstraction, as facilitated by inverted folds, gives rise to an interesting pattern of reuse exemplified as follows.

3.1 Inverted Folds for Nats

The inverted fold as if

```
nfold n s z = foldn s z n
```

is of course directly defined

```
nfold Zero s z = z
```

```
nfold (Succ n) s z = s (nfold n s z)
```

Returning to our running illustrative example of defining operations on naturals, we can now transform the earlier recursive definitions of "add" and "mul" where their recursion pattern is now captured by "nfold":

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

Thus, we have accomplished the first of our goals in remedying the typical definitions of these operations, by replacing the multiple individual interpretations of "Zero" and "Succ" as naturals with the single interpretation given by "nfold". The same effect is of course observable for canonical (non-inverted) "foldn", but the inversion makes things clearer. In particular, for some Nat N, the partial application "nfold N" interprets symbol N into a function, the N-fold iterator, takes some function F and some value X and applies F to X N times. In these terms the respective new definitions of "add" and "mul" above are easy to understand:

- "add" applies "Succ" to "b", "a" times
- "mul" adds "b" to "Zero", "a" times

In other words, partial application of (inverted) fold to symbolic data transforms the symbols into a function that captures a key behavioural characteristic of the data.

3.2 Other Inverted Folds

We now see how partial application of the relevant inverted fold to values of some other symbolic datatypes likewise ascribes applicative behavior to these symbols.

For booleans: the inverted fold is

```
bfold b t f = foldb t f b
```

or as directly defined

```
bfold True t f = t
bfold False t f = f
```

Thus the partial application "`bfold B`" is a function that subsequently takes some values X and Y and chooses between them based on B. In other words, the partial application of bfold to B turns B into a binary branching operator.

For lists: the direct definition of inverted fold is

```
lfold Nil o b = b
lfold (Cons x xs) o b = o x (lfold xs o b)
```

Thus the partial application "`lfold L`" is a function that that composes a binary operation O and a value B in a particular pattern with the elements of L. The important difference however between our inverted list fold and the familiar "foldr" is that the partial application of "lfold" subsequently applies the same list to possibly different O, B.

4 Partial Applications of Other Methods

The behavioural characteristic that is captured by partial application of folds is not the only characteristic that can and should be captured. Partial application of inverted folds above is a special case of partial application of methods to objects, which we can generalise as shown by the following examples.

4.1 Sets and Characteristic Predicates

Let's begin by adapting binary trees "BinT t" (for some element type t) to represent sets (of elements of type t). A sufficient set of constructors can be defined on this representation as follows:

```
data Set t = Empty | Singleton t | Union (Set t) (Set t)
```

Next, we choose to define just one method on these sets, namely a membership test

```
member :: Set t -> t -> Bool
member Empty e = False
member (Singleton x) e = x == e
member (Union s1 s2) e = member s1 e || member s2 e
```

Note how our signature for member broadly corresponds to that for our inverted folds: the first operand is the object to which the method is applied, with other arguments following. This allows for partial application of the method only, in this case for some set S

```
member S
```

is a function that determines if some putative element E is a member of S by subsequent application

```
member S E
```

In other words, "member S" transforms S into a key characteristic behaviour, i.e. the characteristic predicate of S.

4.2 Grammars and Parsers

In what we will eventually demonstrate to be the connection between our work and one of the triumphs of higher-order functional programming, let's investigate the connection between grammars and parsers. A context-free grammar is defined by the following structure:

```
data CFG = Empty | Tok String | Alt CFG CFG | Cat CFG CFG
```

A parse method for this structure is accordingly

```
parse :: CFG -> String -> [String]
parse Empty s = [s]
parse (Tok t) s = if prefix t s then [chop t s] else []
parse (Alt g1 g2) s = parse g1 s ++ parse g2 s
parse (Cat g1 g2) s
      = concat (map (parse g2) (parse g1 s))
```

This is a straightforward recursive-descent lazy backtracking parser of the kind employed by Wadler [10] (we shall see below how the congruent style of "combinator parsing" derives naturally from our approach). In particular, parsing matches a nonterminal or token with a string and results in the residue of the string not matched. Ambiguity is handled by producing a list of residual strings, one for each possible parse, which strategy handles parsing failure *en passant*: indicating same by an empty list of residual strings.

For completeness, the auxiliary functions used for parsing Tok(en)s above are:

```
prefix [] ys = True
prefix (x:xs) [] = False
prefix (x:xs) (y:ys)
       = if x==y then prefix xs ys else False
chop [] (' ':ys) = chop [] ys
chop [] ys = ys
chop (x:xs) [] = []
chop (x:xs) (y:ys) = if x==y then chop xs ys else y:ys
```

Thus, the partial application "parse G" for some CFG G transforms G from a data structure into a behaviour/function - the parser for G.

5 Direct Generation of Zoetic Data by Partial Application Reuse

From the above, we are led to the conclusion that partial application of methods to objects (be the objects lists or other structures, and be the methods folds or more specialised operation) form useful, indeed significant abstractions, that is functional representations of the objects that encapsulate key behaviours, e.g.

- for natural numbers: iteration
- for sets: characteristic predicates
- for context-free grammars: parsing
- etc. etc. etc.

We call these variously "zoetic data" in that they give animation or life (as functions) to otherwise inert symbolic structures, or perhaps more pretentiously "platonic combinators" in that they encapsulate behaviours that are essential to the data structure. For structures for which we assert the existence of and then partially apply a single method (membership for sets, parsing for context-free grammars), the notion of "essence" follows directly. For other pure structures i.e. for which we choose the relevant fold as the method to partially apply, we repeat our earlier arguments about expressiveness, simplicity and calculational properties as the basis for our claim that fold-behaviour represents the essence of the related type.

There are however significant deficiencies in our present means of creating and using these abstractions. In particular, compositionality is lacking in that e.g. while Sets can be created from values and other sets (e.g. using the "union" function to

access the Branch constructor), there is as yet no way of composing characteristic predicates from characteristic predicates. Rather, if

```
cp1 = member s1
cp2 = member s2
```

then to create the characteristic predicate for the union of s1 and s2, it's necessary to work at the symbolic (s1, s2) level rather than the zoetic (cp1, cp2) level:

```
member (Union s1 s2)
```

In other words, having created cp1, cp2, they are not available for reuse. In other words, from our thematic "Reuse" point of view, we need to find a means of reusing the partial application pattern exemplified by "member (Union s1 s2)".

Accordingly, we look to synthesise what we call "generators" - the zoetic counterparts to constructors for symbolic data structures. Generators will allow us to construct zoetic data directly from their components, including other zoetic data where appropriate, without having recourse to explicit partial application of methods. Straightforward reasoning gives results in each example case as follows. Generalisation of the approach to other objects and methods should be apparent.

5.1 Synthesising Iterators

We begin by giving specifications for the generators in terms of the partial applications. For each symbolic constructor (Zero, Succ) there is a corresponding zoetic generator (zero, succ).

Specification:

1. zero = nfold Zero
2. succ (nfold n) = nfold (Succ n)

Note the subtlety in the second case: the homomorphic requirement that the "succ" generator applies to zoetic naturals (i.e. iterators) is indicated by expressing its argument as the partial application of "nfold" to symbolic natural n - the same operand of constructor Succ on the right-hand-side of the specification.

Derivation: the derivation of effective source code (Haskell) definitions for the generators proceeds by applying each side of the specification to the complete argument list, i.e. the putative element to which the characteristic predication applies.

1. zero f x
   = nfold Zero f x
   = x

The case of recursive structure with respect to "succ" requires a generalisation/implication step, here because of the complexity of the operand for "succ" in the specification.

2. succ (nfold n) f x
   = nfold (Succ n) f x
   = f (nfold n f x)
   (generalising "nfold n" to iterator i)
   <= succ i f x= f (i f x)

Implementation: extracting the results

```
zero f x = x
succ i f x = f (i f x)
```

Accordingly, the definitions of zoetic versions arithmetic operators "add" and "mul" can now be rendered free of explicit application of the "nfold" interpreter of iterative behaviour from naturals:

```
addz a b = a succ b
mulz a b = a (addz b) zero
```

5.2 Synthesising Characteristic Predicates

As before we give specifications for the generators in terms of the partial applications. For each symbolic constructor (Empty, Singleton, Union) there is a corresponding zoetic generator.

Specification:

1. empty = member Empty
2. singleton x = member (Singleton x)
3. union (member s1) (member s2) = member (Union s1 s2)

Note the subtlety in the third case: the homomorphic requirement that "union" generator applies to zoetic sets is indicated by expressing the arguments as the partial applications of "member" to symbolic sets s1, s2 - the same symbolic sets that appear as operands to constructor "Union" on the right-hand-side of the specification.

Derivation: the derivation of effective source code (Haskell) definitions for the generators respectively proceeds by applying each side of the specification to the complete argument list, i.e. the putative element to which the characteristic predication applies.

1. empty e
   = member Empty e
   = False
2. singleton x e
   = member (Singleton x) e
   = x == e

The first two cases proceeded by straightforward equational reasoning, but the third requires a generalisation/implication step because of the complexity of the operands for "union" in the specification.

3. union (member s1) (member s2) e
   = member (Union s1 s2) e
   = member s1 e || member s2 e
   (generalising partial applications "member s1", "member s2" to characteristic predicates cp1, cp2)
   <= union cp1 cp2 e = cp1 e || cp2 e

Implementation: extracting the results yields Haskell definitions

```
empty e = False
```

```
singleton x e = x == e
union cp1 cp2 e = cp1 e || cp2 e
```

Similar patterns will appear in the specification, derivation and implementation of other generators.

5.3 Synthesising Folded Lists

Derivation of generators for zoetic lists (partial application of lfold) parallels that zoetic naturals above.

Specification:

1. nil = lfold Nil
2. cons x (lfold xs) = lfold (Cons x xs)

Derivation:

1. nil o b
   = lfold Nil o b
   = b
2. cons x (lfold xs) o b
   = lfold (Cons x xs) o b
   = o x (lfold xs o b)
   (generalising "lfold xs" to zoetic folded list fxs)
   <= cons x fxs o b = o x (fxs o b)

Implementation:

```
nil o b = b
cons x fxs o b = o x (fxs o b)
```

5.4 Synthesising Parsers

The pattern established above is followed as a matter of course.

Specification:

1. empty = parse Empty
2. tok t = parse (Tok t)
3. alt (parse g1) (parse g2) = parse (Alt g1 g2)
4. cat (parse g1) (parse g2) = parse (Cat g1 g2)

Derivation:

1. empty s
   = parse Empty s
   = [s]
2. tok t s
   = parse (Tok t) s
   = if prefix t s then [chop t s] else [ ]

3. alt (parse g1) (parse g2) s
   = parse (Alt g1 g2) s
   = parse g1 s ++ parse g2 s
   (generalizing partial applications "parse g1" and "parse g2" to parsers p1 and p2)
   <= alt p1 p2 s = p1 s ++ p2 s
4. cat (parse g1) (parse g2) s
   = parse (Cat g1 g2) s
   = concat (map (parse g2) (parse g1 s))
   (generalization as above)
   <= cat p1 p2 s = concat (map p2 (p1 s))

Implementation:

```
empty s = [s]
tok t s = if prefix t s then [chop t s] else []
alt p1 p2 s = p1 s ++ p2 s
cat p1 p2 s = concat (map p2 (p1 s))
```

As is evident, we have thus derived the parsing combinators discovered by Wadler [10] and subsequently elaborated and popularized notably by Hutton with Meijer [11, 12], but here from the zoetic data/platonic combinator/totally functional programming point of view!

6 Summary: Totally Functional Programming

To summarise, TFP is about replacing symbolic data and operations thereon (particularly constructors) by zoetic data and corresponding operators (particularly what we term "generators"). The underlying hypothesis is that for every symbolic datatype there is an inherent behavior reflected in the corresponding zoetic form. For pure data structures (booleans, naturals, lists, trees, etc.) this behavior derives from the appropriate fold functions (which exist for every such type). For "impure" structures (where some other characteristic method is explicitly defined, e.g. membership for sets, parsing for grammars), the datatype's behavior derives from that characteristic method.

The approach to TFP reported here proceeded through the following steps:

- folds encapsulate recursion patterns for reuse
- through partial application, folds further encsapsulate patterns of reuse of operations applied to different data
- inverting the order of arguments to fold gives a complementary reuse pattern, of (zoetic) data applicable to different operations
- partial application to objects of methods additional than fold but which nonetheless characterize object behaviors widens the scope of these zoetic reuse patterns

- compositionality and reuse of zoetic data is facilitated by direct generation of same using "generator" functions corresponding to constructors on the corresponding symbolic data
- derivation of generators is straightforward.

A number of other observations about TFP offer enlightenment as follows.

6.1 Theoretical Antecedents

Some of the constructions of TFP were discovered during the early development of the lambda-calculus [13], where the existence only of pure functions in the typeless calculus and the consequent absence of symbolic data induced the need for creativity in demonstrating universality. In the key case of functions over natural numbers our zoetic naturals/iterators in the guise of "Church numerals" were an early discovery. Subsequently, other kinds of zoetic data have been prominent in theoretical systems such as as Reynolds' second-order polymorphic typed-lambda-calculus [14]. The principal difference with TFP is that whereas these earlier presentations of zoetic data may have been to ensure a degree of conceptual parsimony, our aim is to develop and promote zoetic data through TFP as a viable software development style.

6.2 The Role of Fold

Even though we cannot emphasise enough that zoetic data arise from the partial application of methods additional to folds, the fold-family nevertheless plays an important theoretical and practical role in TFP.

Firstly, folds naturally represent the behaviours inherent in pure data structures (i.e. those without any other characteristic method such as set membership):

- booleans select
- naturals iterate
- lists fold (in the specific "foldr" sense)
- etc.

Secondly, the identity property of folds allows us where necessary to traffic freely from the symbolic to the zoetic domain and vice versa. For example using the inverted counterpart of foldr:

```
lfold L O B
```

replaces all occurrences in L of Cons by O and Nil by B. Pure zoetic structures i.e. for which fold is the characteristic behavior inherit the same property. Thus for example, for symbolic list L and zoetic list Lz

```
lfold L Cons Nil = L
Lz Cons Nil = L
Lz cons nil = Lz
lfold L cons nil = Lz
```

This device is very useful for the practical purpose of being able to display the contents of a pure zoetic structure; otherwise as a function the internal details would not be printable.

Finally, the identity property also supports a useful kind of inheritance mechanism, in particular allowing us to create impure structures from pure ones. For example, noting that our earlier binary trees BinT have a structure isomorphic to Set, for some binary tree BT we can synthesise the corresponding zoetic set by folding with the set generators, i.e.

```
foldbt empty singleton union BT
```

Similar syntheses would be possible with an inverted fold for BinT, and for the consequent zoetic BinT. This role of fold as the "mother of all zoetic data" serves to confirm our proposition that folds are the correct choice for the inherent characteristic behaviours of pure structures.

6.3 Loop- and Branch-Free Programming Style

In view of the grounding of TFP in fold functions, the rationale of which is to package, reuse and indeed remove the need for explicit use of recursion, it should be no surprise that zoetic data evade explicit recursion inherently.

By way of illustration, recall how the zoetic arithmetic operators are built on iterator generators:

```
addz a b = a succ b
mulz a b = a (addz b) zero
```

Note that it's not just recursion/looping that's avoided; branching can be dispensed with also. Consider the direct definitions of logical connectives on zoetic Booleans tz and fz defined as follows (but whose derivations from partial application of bfold are left as an even more trivial exercise):

```
-- generators
tz t f = t
fz t f = f

-- operations
notz bz = bz fz tz
andz bz1 bz2 = bz1 bz2 fz
orz bz1 bz2 = bz1 tz bz2
```

6.4 Programming without Interpretation

The "reuse-based" motivation for TFP that is presented here parallels an earlier approach, based on an evident correspondence between programming and language extension [15]. The relevant consequence of this correspondence is that programmers should avoid programming techniques that reflect unsound language extension practice. Our contention is that the outstanding such unsound technique is the use of symbolic interpretation to define new constructs, rather than direct definition. In the

extreme case, symbolic interpretation would involve extending a language by writing a new interpreter with the new constructs as additional primitives. Even though the effort may be managed by access to the existing interpreter (as in Reflection [16]), interpreter extension entails levels of complexity and risk significantly in excess of the alternative, that is by simple direct definition and declaration of the new constructs as source code in the host language of the extension.

Instead, as we have seen, the replacement of symbolic representations of data with functional/zoetic counterparts that inherently embody the characteristic behaviours of datatypes, eliminates the need for programmers to write interpreters to extract these behaviours from the symbolic representations.

It's notable that in order to eschew interpretation, the host language needs to be expressive enough to permit direct definition of extensions. In the case of TFP, the characteristic capabilities of functional languages are indispensable: when data are represented by functions, operations on these zoetic data are necessarily higher-order.

6.5 Related Developments

Our TFP may be thought of as a realization of Reynolds' [14] prophetic words, that: "… using [second-order] polymorphic functions [as] data may be the key to a novel programming style".

In a development that ours complements, Turner's coincidentally similarly-named "Total Functional Programming" [17, 18] exploits the expressive power of second-order polymorphism to avoid general recursion. In this respect Turner's arguments are considerably more well-developed than ours; on the other hand our arguments in favour of zoetic data seem relatively more well-developed. In the long run however we are optimistic that the two approaches will evidently converge.

7 Conclusions and Future Directions

Our demonstrated connection between TFP and non-trivial functional programming (from all kinds of fold to combinator parsing) is meant to suggest that TFP is more than an academic curiosity and indeed that our goal of promoting "Church numerals" and all their ilk from the laboratory to the programmer's workbench (so to speak) is worth pursuing. There however remain numerous challenges to the attainment of this goal, most prominently the following. They form the basis of the next stage of our TFP research program.

How Impactful is TFP Already? We've been greatly encouraged by the realization that combinator parsing is an example of TFP with independent existence and recognition within the FP community. How many more such encouraging examples can be revealed and used to expand our knowledge of TFP development?

Type Limitations. The now-standard Hindley-Milner type-inference system does not accept some even relatively simple TFP expressions. For example, the straightforward TFP definition for exponentiation (in the spirit of "add" and "mul") is

```
powz a b = b (mulz a) (succ zero)
```

However, expressions like

```
powz (succ (succ zero)) (succ zero)
```

then result in type errors. What kind of more expressive but still simple type system(s) will handle a wide enough class of TFP expressions to be useful? While second-order polymorphic type-checking is decidable [19], it's however unclear to what extent programmers need to add relatively burdensome type annotations to source code in view that type inference is not available.

What Other Primitives are Useful? For example, folds play a key role in TFP, but there are generalisations of fold [7] for other forms of types. Do any of these capture essential behaviours in the apparently beautiful and thus true way that basic folds do?

Relation to OOP. Implicit in TFP is that each object has a single method (other than constructors/generators), whereas "real" programming (as exemplified by OOP) permits multiple methods, e.g. grammars may have "print" as well as "parse" methods. How can e.g. the device by which multiple kinds of zoetic data are derived by instantiating folds be extended to the derivation of the relevant generators, or some such?

Soundness of TFP. Is the kind of function definition by symbolic interpretation that TFP characteristically eschews something that can be defined objectively and detected mechanically? What would be the semantic impact of the syntactic restrictions implicit in such a mechanism?

Implementation Optimisation. What opportunities for extraordinarily-efficient implementations of TFP are offered by the avoidance of recursion (directly, or indirectly through fixed-point operators, assuming occurrences of the latter can indeed be mechanically detected and prevented e.g. by suitable static type-checking).

What Kinds of Modelling Techniques are Effective for TFP? Hitherto, modeling of symbolic data structures has been at the core of software development. In the absence of same, and instead with exclusively functional entities at hand, how can (large) systems be synthesized, especially avoiding explicit partial application of methods to symbolic data. Is there anything to guide us from the days of analog computing, when programs were also constructed from just the computational behaviours of their components?

Acknowledgments. The author is indebted to several colleagues who have contributed to the development of the above ideas over the years: Ian Peake, Sean Seefried and Leighton Brough. Special citation needs however to be made of Colin Kemp whose PhD studies and resulting thesis [20] has provided indispensable contribution and stimulation to the Totally Functional Programming project.

References

1. Bailes, P., Kemp, C.: Formal Methods within a Totally Functional Approach to Programming. In: B.K. Aichernig, B., Maibaum. T. (eds.), Formal Methods at the Crossroads: from Panacea to Foundational Support. LNCS, vol. 2757, pp. 287--307. Springer, Heidelberg (2003)
2. Bailes, P. and Kemp, C.: Fusing Folds and Data Structures into Zoetic Data. In: Proc. 23rd IASTED International Multi-Conference on Applied Informatics (AI 2005), pp. 299-306. Acta Press, Calgary (2005.)
3. Thompson, S.: Haskell: The Craft of Functional Programming. Addison-Wesley Longman, Harlow (1996)
4. Dijkstra, E.: Goto Statement Considered Harmful. In: Comm. ACM, vol. 11, pp. 147--148 (1968)
5. Uustalu, T., Vene, V. and Pardo, A.: Recursion Schemes from Comonads. In: Nordic J. of Comput. vol. 8(3), pp. 366--390 (2001)
6. Meijer, E., Fokkinga, M. and Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In: FPCA 1991, LNCS, vol. 523, pp. 124--144 (1991).
7. Hutton, G.: A Tutorial on the Universality and Expressiveness of Fold. In: Journal of Functional Programming, 9, pp. 355--372 (1999)
8. The Haskell Programming Language, http://www.haskell.org/haskellwiki/Haskell
9. Backhouse, R., Jansson, P., Jeuring, J. and Meertens, L.: Generic Programming - An Introduction. In Swierstra, S., Henriques, P. and Oliveira, J. (eds.), Advanced Functional Programming, LNCS, vol. 1608, pp. 28--115 (1999)
10. Wadler, P.: How to Replace Failure by a List of Successes. In: FPCA 1985. LNCS, vol. 201, pp. 113--128 (1985).
11. Hutton, G.: Higher-order functions for parsing. In: Journal of Functional Programming, vol. 2, pp. 323--343 (1992)
12. Hutton G, Meijer, E.: Monadic parsing in Haskell. In: Journal of Functional Programming, vol. 8, pp. 437--444 (1998)
13. Barendregt, H.: The Lambda Calculus - Its Syntax and Semantics 2nd ed. North-Holland, Amsterdam (1984)
14. Reynolds, J.: Three approaches to type structure, Mathematical Foundations of Software Development, LNCS, vol. 185, pp. 97--138. Springer, Heidelberg (1985)
15. Bailes, P.: The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design). In: Proc. 1986 Austrln. Software Eng. Conf., pp. 14--18. IEAust, Canberra (1986)
16. Jefferson, S., Friedman, D.: A simple reflective interpreter. In: Lisp and Symbolic Computation, vol. 9, pp. 181--202 (1996)
17. Turner, D.: Elementary Strong Functional Programming. In: Plasmeijer, R. and Hartel, P. (eds.), First International Symposium on Functional Programming Languages in Education, LNCS, vol. 1022, pp. 1--13 (1995)
18. Turner, D.: Total Functional Programming. In: Journal of Universal Computer Science 10 (7), pp. 751--768 (2004)
19. Vytiniotis, D., Weirich, S., Jones, S.L.P.: Boxy types: inference for higher-rank types and impredicativity. In ICFP 2006, pp. 251--262 (2006)
20. Kemp, C.: Theoretical Foundations for Practical "Totally-Functional Programming". PhD Thesis, The University of Queensland, St Lucia (2009)