Making Sense of Recursion Patterns

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

Abstract—**Recursion patterns (such as "foldr" and elaborations thereof) have the potential to supplant explicit recursion in a viable subrecursive functional style of programming. Especially however in order to be able to eschew explicit recursion entirely, even in the definition of new recursion patterns, it's essential to identify and validate a minimal set of basic recursion patterns. The immediate plausibility of foldr is validated by its application to the implementation of functions and recursion patterns, and especially by an abstract characterization of the programming devices used in these applications used to overcome complementary information deficiencies in data and control.**

Keywords - foldr; functional programming; recursion

I. Introduction

Recursion [1] is a key programming technique, explicitly in declarative (applicative/functional and logic) programming, and at least implicitly in imperative programming. While the inductive proofs to which recursive functions over recursive datatypes are amenable [2], are arguably more accessible than some other forms of program verification [3], induction remains a non-trivial method. Identification in particular of an appropriate inductive hypothesis can be problematic.

An alternative to recursion is the use of "canned" recursion in the form of patterns [4], by which useful forms of recursion can be identified, and their behaviours captured in terms of laws which are easier to use for program verification and derivation than "raw" recursion/induction. However, the definition and use of recursion patterns entails a problem of adequacy, with manifestations as follows:

a) which (if any) set of recursion pattern is theoretically adequate for the range of programs (and metaprograms) we want to write?

b) which (if any) set of recursion pattern are pragmatically adequate for the range of programs (and metaprograms) we want to write?

c) in particular, how are any needed new recursion patterns created?

In that context, the purpose of this paper is to contribute to the realization of the promise that recursion patterns offer to the use of formal methods in software engineering, by answering these questions. A solution emerges by which it appears that the synthesis of necessary adaptations to the basic "foldr" recursion pattern depends upon the identification of information deficiencies in terms of (a) control and (b) data, and their respective rectification in terms of complementary adaptations that (a) return extra data from functions for later selection when control information is available and (b) further parameterize functions for later instantiation when required data is available.

A functional programming context is critical to our development to the extent that programmer-definable function-valued functions are supported, with specific examples in Haskell [5].

II. Recursion Patterns on Lists

Much (but by no means all) of our discussion will be about recursion patterns on list data. This focus is theoretically and pragmatically legitimate for a number of reasons. One of these is that lists are capable of representing other structures without the obfuscation of alternatives (e.g. Gödel numbering). Another is that recursion patterns on lists are special cases of abstract recursion patterns over regular recursive types [4, 6, 7], implying that results for lists apply to this wider class of structures. Note that in the below we take care to give choice examples of generalisations from lists to other structures.

A. Foldr

A simplest pattern of recursion over lists directly reflects the structure of lists, as either an empty list or as a construction comprising a head element and a tail (sub-)list. Foldr combines list elements with the result of recursive processing of sub-lists:

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

B. Inverted Foldr

At least because the focus in the rest of this presentation will be on the functional operands of "foldr" other than the specific structure to which it will be applied, we will be using a variant of "foldr" that reorders arguments to give the structure first so that it can be followed uniformly by the other operands without the occasional interference of the structural argument. At the same time, we will adopt a naming convention that accommodates the counterparts of "foldr" on structures other than lists, whereby a fold is by default a "right" fold and is distinctively named to suggest the structure to which it applies.

Thus:

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

or

```
foldXs xs op b = foldr op b xs
```

We shall nevertheless continue to use "foldr" to reference this recursion pattern, both on lists and, as we shall see, all other regular recursive types.

C. *Foldr-derivatives*

Many other patterns of recursive list processing are easily-expressible in terms of foldr:

```
map f xs = foldXs xs (\ x xs′ -> (f x):xs′) []

filter p xs =
    foldXs xs
    (\ x xs′ -> if p x then x:xs else xs)
    []
```

Et cetera.

D. *Foldl*

On the other hand, some patterns are at least at first glance inimical to foldr. Arguably the simplest of these is the one which in contrast to foldr's combination of elements "from the right" instead combines elements "from the left"

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

However, once the difference in recursion pattern is embodied in foldr vs foldl, the similarity between different function definitions can become apparent, as exemplified by the rendition of the length function in terms of each:

```
length xs = foldXs xs (\ _ xs′ -> 1 + xs′) 0
```

vs.

```
length xs = foldl (\ b′ _ -> 1 + b′) 0 xs
```

The foregoing "map" and "filter" patterns are similarly expressed as "foldr" vs. "foldl". On the other hand, there are cases when the difference in recursion pattern induces significant differences in usage, viz. the respective differences between alternative definitions of ++ and rev(erse):

```
xs ++ ys = foldXs xs (\ x xs′ -> x:xs′) ys
rev xs = foldrXs xs (\ x xs′ -> xs′ ++ [x]) []
```

vs.

```
xs ++ ys = foldl (\ b′ x -> x:b′) ys (rev xs)
rev xs = foldl (\ b′ x -> x:b′) [] xs
```

In the case of ++, the combination of elements of xs with those of ys follows the traversal of xs undertaken by foldr/foldXs. The list traversal undertaken by foldl is not in that sense natural to the problem, requiring a conceptually-complex and inefficient reversal of xs prior to processing by foldl.

In the case of rev(erse), it is foldl that offers the traversal pattern natural to the problem; whereas the foldr-based solution needs a costly (if maybe not so conceptually-complex) concatenation of each element of xs to the end of ys.

E. *Paramorphism*

Our final sample recursion pattern extends foldr by supplying to the combining op(eration) the unprocessed list tail, in addition to the head and the result of recursion on the tail as provided by foldr:

```
paraXs [] op b = b
paraXs (x:xs) op b = op x xs (para op b xs)
```

Paramorphisms thus allow definitions of functions otherwise not directly conveniently available with foldr. For example, to insert an element in sorted position in a list:

```
insert e xs =
    paraXs xs
    (\ y ys yse ->
        if e < y then e:y:ys else y:yse
    )
    [e]
```

(Just how to achieve the convenient expression of paramorphism in foldr will be covered among the major themes to be developed below.)

F. *Summary*

Based on the variety among simple recursion patterns, it's unsurprising that an infinity of others potentially exists. A simple demonstration of this infinitude is to generalize foldr beyond paramorphism with options that present for recursive combination not just the result of processing the tail of the current list, but also the tail of the current list itself, the tail of the tail of the current list (if defined), the result of processing the tail of the tail, etc. etc. etc. Not all of these possibilities may be useful – our next problem is to identify which one(s) is (are) sufficiently useful to be promoted as a basic language construct in place of explicit general recursion.

III. Foldr as the Basic Recursion Pattern

Having established that programs are better structured, verified and synthesized in terms of recursion patterns than explicit recursive definitions, the question then arises of adequacy, i.e. which if any pattern should be used as a basis but as the basis for programming.

In the scenario that we (along with others as cited above) are advocating, ordinary programmers would strictly eschew explicit recursion in favour of recursion patterns. A remaining question however concerns the provenance of the potential infinitude of new recursion patterns awaiting discovery and application: is there an ongoing use for explicit recursion, e.g. by an elite of language designers and implementers (say, of standard libraries) for these new patterns? Alternatively, should explicit recursion never be available, in which case a finite set of basic recursion patterns needs to suffice, not only for ordinary programming but for the metaprogramming of new recursion patterns.

A. *Programming as Language Extension*

The choice of the latter alternative is effectively made for us by the objective affinity between programming on the one hand and language design/implementation under the guise of language extension on the other.

1) "Program" = "Language"

Programming artefacts directly correspond to language design & extension artefacts. The correspondence between programming and language design (and thus language extension, as its products are the result of both programming and language design) is discernable in correspondences between the natures of the outputs of these processes, in summary as follows:

- software component libraries can be at least as long-lived as programming languages;
- software component libraries can have as many components (and be as difficult to learn) as programming languages;
- software component libraries can be as widely-distributed as programming languages.

2) "Programming" = "Language Design"

Further substantiation comes from the correspondence in the respective approaches taken to them by their practitioners. Our key observation [8] is that there is a correspondence between programming language design criteria and program quality criteria, in summary:

- adequacy of language constructs vs. application-focus of program components;
- orthogonality of language constructs vs. loose coupling of program components;
- simplicity of language constructs vs. cohesiveness of program components;
- readability of languages vs. choice of naming convention, layout etc. in programs.

3) Basis in denotational semantics

The ultimate unity of programming and language extension is objectively demonstrable via denotational semantics [9], which shows that declarations, the essence of any modular programming discipline, effect language extensions.

First simply interchange the operands of the usual denotational meaning function M, i.e. changing the signature to "M :: Envt $\rightarrow$ Rep $\rightarrow$ Dom" where: Rep is the domain of representations/syntax of programs; Dom is the domain of meanings/semantics of programs; and Envt is the domain of environments, i.e. mappings from identifiers Id to elements in Dom to which they are bound by prevailing declarations and thus Envt = Id $\rightarrow$ Dom. Semantics of expressions are unchanged, save that operands of M are reordered from the usual, e.g.

$$M\ \rho\ [\![I]\!] = \rho\ [\![I]\!]$$

The significant difference however is that semantics for declarations can be expressed in terms of partial application of M to the updated environment:

$$\begin{aligned}&M\ \rho\ [\![\textbf{let}\ I = E]\!] = \\ &\qquad M\ (\lambda\ i\ .\ \textbf{if}\ i = [\![I]\!]\ \textbf{then}\ M\ \rho\ [\![E]\!]\ \textbf{else}\ \rho\ [\![I]\!])\end{aligned}$$

and semantics for a program rewritten consistently:

$$M\ \rho\ [\![D\ ;\ E]\!] = M\ \rho\ [\![D]\!]\ [\![E]\!]$$

Critically, this arrangement can be restructured to identify explicitly the partial application of M to an environment, say M':

$$\begin{aligned}&M\ \rho = M' \\ &\quad \textbf{where} \\ &\quad M'\ [\![D\ ;\ E]\!] = M'\ [\![D]\!]\ [\![E]\!] \\ &\quad M'\ [\![\textbf{let}\ I = E]\!] = \\ &\qquad M'\ (\lambda\ i\ .\text{if}\ i = [\![I]\!]\ \textbf{then}\ M'\ [\![E]\!]\ \textbf{else}\ \rho\ [\![I]\!]\ )\end{aligned}$$

In a real sense, M' (the partial application "M ρ") effectively defines the prevailing programming language, ascribing as it does meanings to all symbols, both built-in syntax and identifiers defined so far by declaration. Correspondingly, "M' [[D]]" defines M' extended by D. That is, declaration D truly extends language M'.

4) Conclusion

We've now demonstrated in a number of ways that programs, or their components, can be viewed as if components of programming languages. Hence, programming can be viewed as a language development process. Because of the limited means by which programmers can effect language development (typically by definition/declaration) this means programming can be viewed as language extension. As shown, this contention is cemented from a formal semantic perspective.

The implication for the current investigation is to question the suggestion of any distinction between ordinary programmers whose role it is to use recursion patterns, from elite programmers whose job it is to create recursion patterns using more sophisticated explicit recursive techniques. In other words we need to have a set of basic recursion patterns and critically, an understanding of them, that is strong enough to sustain the creation of any new recursion patterns that may be desired.

B. Foldr as Basis

Identification of a basic set of recursion patterns first needs to satisfy a broad expressiveness requirement. In eschewing general recursion for recursion patterns, we acknowledge that we forsake Turing-completeness, but point to ongoing work on the broad area of subrecursion (e.g. Turner's Total Functional Programming or TFP [10]) that justifies this position. In particular, it's most significant that the simplest recursion pattern – foldr – is powerful enough (in the presence of polymorphism and programmer-definable higher-order functions) to express arcana such as Ackermann's function [11] and indeed any function provably-terminating in second-order arithmetic [12].

Foldr is the simplest of a class [4, 6, 7] of recursion patterns (on regular recursive types) known as "catamorphisms" (often this term is used to refer to foldr specifically). While each of these generalisations (including in addition to paramorphisms as above, exotica such as zygomorphisms and histomorphisms) offer convenience of expression appropriate to the level of application (as indeed does foldr to its level), it's to say the least unclear (e.g. http://knol.google.com/k/catamorphisms#) what if any useful additional formal expressive power is offered beyond foldr.

To summarise:

- any one of the envisaged recursion patterns could conceivably suffice as a formal basis for subrecursive programming;
- but none appears to have overall specific pragmatic advantages over any of the others;
- except that foldr is the simplest;
- however, because no pattern can however be excluded as a pragmatically useful device for structuring programs, they all need to expressible in terms of whatever basis we choose.

Because foldr is distinguished by its simplicity, and because no other pattern exhibits a contending distinction, we are impelled to investigate the pragmatics of foldr as the unique basic recursion pattern. Validation of this choice, in particular how other recursion patterns can be derived from foldr without recourse to general recursion, is in essence the subject of the remainder of this paper.

C. *Foldr for Other Structures*

The categorical grounding of foldr [6, 7] provides the basis for corresponding catamorphisms for all other regular recursive types. (Indeed, a suitable polymorphic type regime admits a single generic definition of catamorphism for all such types, but that's beyond the scope of this investigation, and indeed the terseness of some of the generic definitions risks obscuring the points we wish to make.)

Accordingly, for the binary tree

```
data BT a = Node (BT a) (Bt a) | Leaf a
```

we have the inverted foldr counterpart

```
foldBT (Leaf a) np lp = lp a
foldBT (Node bl br) np lp =
       np (foldBT bl np lp) (foldBT br np lp)
```

For natural numbers (integers n >= 0), we have n-times iteration (aka Church numerals [13]):

```
foldN 0 s z = z
foldN n s z = s (foldN (n-1) s z)
```

Et cetera.

IV. Applications of Foldr

The first stage in validating foldr as basis is to expose the devices by which it can be used to render, not just theoretically but plausibly, functions and algorithms that don't immediately match the foldr pattern.

A. *List Insert*

The basic reason why element insertion is conveniently expressible as a paramorphism but not easily as a foldr is because the op(erator) argument does not have access to the tail of the list, only to the result of fold-ing the tail. Accordingly, knowledge of the tail has to be explicitly granted to the foldr result. This is achieved by having the fold return a pair (in the general case of problems of this nature, a tuple) of values:

- the current list (which will be the tail for the next higher level)
- the value that would otherwise be returned by the fold.

Thus:

```
insert e xs =
    snd $ foldXs xs
    (\ y (ys, yse) -> (
        y:ys,
        if e < y then e:y:ys else y:yse
    ))
    ([],[e])
```

B. *List Tail*

The tupling device can be seen in its essence in the rendering of a total version of the list tail function in terms of foldr:

```
ttl xs =
    snd $ foldXs xs
    (\ y (ys, ys') -> (y:ys, ys))
    ([], <ERROR>)
```

Here, the first element of the pair returned by the foldr is, as for insert above, the current entire list which is to serve as the tail at the next upward level. The second element is the tail of the current list, which is selected by ttl as the overall result. Each stage of the foldr simply returns the entire list at the next lower level (the first tuple element argument) as the tail of the current level (the second tuple element result).

Note that the definition of ttl of [ ] as some <ERROR> value can easily be replaced by a more elegant solution e.g. involving "Maybe" types if desired, or even [ ] as some might have it.

C. *Factorial*

A celebrated example in the natural number domain is the foldr (i.e. iterative) rendition of the counterpart to ttl – the arithmetic predecessor function [13]. We will here however explicate a slightly more interesting example of an iterative definition:

```
fact k =
    snd $ foldN k
    (\(n, n') -> (n+1, (n+1)*n'))
    (0,1)
```

During the course of this iterative foldr, there is created a succession of tuples (0, 1), …, (k, k * … * 1), from the last of which the second element is selected at the top level. FORTRAN programmers familiar with DO-loops should feel at home here.

D. List Indexing

While the previous examples have all been variations on the "tupling" theme, a different kind of solution is required for the problem of replacing each element of a list with its index (which can be thought of as a very abstract version of a formatted display of a data structure – shown more clearly in the context of tree indexing next below).

```
idx xs =
    foldXs xs
    (\ _ f -> (\ m -> m : f (m+1)))
    (\ m -> [])
    0
```

Here, the foldr is "functionalised", i.e. adapted not to return a tuple but a function, with the index of the current element of the foldr as parameter. The resulting index list is formed by prepending the current index to the application of the function resulting from the foldr of the tail, to the incremented index. In the case of the empty list, the function returns empty. The entire foldr result is then applied to the initial index element (here 0).

E. Tree Indexing

An indexed binary tree is a binary tree with each node or leaf decorated with its depth from the root (as would form the basis for an indent-formatted display):

```
data XBT = Xl Int | Xn Int XBT XBT
```

The above functionalised solution for indexed lists is simply adapted to provide for the binary branching in these trees as well as indexing Leaf elements.

```
idxBT bt =
    foldBT bt
    (\ bl br ->
        (\ m -> Xn m (bl (m+1)) (br (m+1)))
    )
    (\ a -> (\ m -> Xl m))
    0
```

V. Other patterns as Foldr

From the above renditions of specific foldr-applications, it's a simple step to abstract the definitions of other recursion patterns.

1) List Paramorphism

Just as operations, such as sorted element insertion which are conveniently rendered as paramorphisms, require tupling for their rendition in foldr (or foldXs as we now express it it), so does the generic rendition of paramorphism in foldr terms require tupling:

```
paraXs xs o b =
    snd $ foldXs xs
    (\ x (xs, xs') -> (x:xs, o x xs xs'))
    ([], b)
```

2) Natural Number Paramorphism

Correspondingly (i.e. abstracting from fact(orial) above):

```
paraN n f x =
    snd $ foldN n
    (\ (m, m′) -> (m+1, (f m m′)))
    (0, x)
```

Thus:

```
fact k = paraN k (\n n′ -> (n+1)*n′) 1
```

3) Binary Tree Paramorphism

Just as indexing for binary trees adapts from indexing for lists, so we can adapt paramorphisms for lists to binary trees.

```
paraBT bt np lp =
    snd $ foldBT bt
    (\ (bl, bl′) (br, br′) ->
        (Node bl br, np bl bl′ br br′)
    )
    (\ a -> (Leaf a, lp a))
```

4) Functionalisation as Alternative to Tupling

Tupling has been promoted [14] among other things as a means of combining multiple foldr scans of a data structure, rather than an expressive device. Nevertheless, it seems that tupling is necessary for the definition of list tail as a foldr (above). But then, however tail is provided, it can be used in conjunction with functionalization instead of tupling. For example:

```
paraXs xs o b =
    foldXs xs
    (\ y ys' ->
       (\ys ->
        o y ys (ys' (tail ys))
       )
    )
    (\ ys -> b)
    (tail xs)
```

Here, the tail of the initial list xs is passed as the ys parameter to the functionalized fold and successive tails to the recursive functionalized calls. Having demonstrated the fold-expressiveness of tail (albeit perhaps inefficiently), this approach to paramorphisms might be suitable if tail were implemented efficiently as a language primitive.

5) List Foldl

In the foldr/foldXs rendition of foldl, knowledge of the left-to-right accumulated foldl result is functionalized as the 'c' (for "context") parameter, for which the b(ase) value of the fold is supplied as initial value :

```
foldl op b xs =
    foldXs xs
    (\ x xs′ c -> xs′ (op x c))
    (\ c -> c)
    b
```

6) Combined Tupling and Functionalisation

Tupling and functionalization are not just alternatives, but can be combined productively. Consider the replacement of list elements by some function g of their index position, scaled by the length of the list (as might be the case on some non-trivial formatting application). The pure functionalized solution is:

```
idx g xs =
    let
        f =
            foldXs xs
            (\ x f ix ln ->
                ((g ix)/ln) : f (ix+1) ln
            )
            (\ ix ln ->[])
    in
        f 1 (length xs)
```

Note that the functionalization involves multiple parameters: the index position of the current element, as well as the list length (which is invariant).

A drawback to the above is however the dual foldr scan of xs: first to calculate the length, and then to transform the list based on the former result. The alternative is to use tupling to combine the calculation of the length with the formatting traversal, which then is passed back along with the index as functionalization parameter as before:

```
idx g xs =
    let
        (f, n) =
            foldXs xs
            (\ x (f, n) -> (
                (\ ix ln ->
                    ((g ix)/ln) : f (ix+1) ln
                ),
                n+1
            ))
            ((\ ix ln ->[]),0)
    in
        f 1 n
```

See [14] for validation of even more complex kinds of apparent "circular" programming.

VI. Foldr-based Program Design

Adapting foldr to return functional values is not unprecedented. Tupling especially is well-known for improving the expressiveness of foldr and the efficiency of foldr-based programs. The question for us is one of how to know when to use tupling, functionalization, or a combination, which knowledge might be equally applied either in informal program development or in formal calculation.

A. Tupling in Programming

Consider how tupling is introduced into the solution of the sorted list insertion problem, the essence being to determine determine the <op> and <b> operands to foldXs in

```
insert e xs = foldXS xs <op> <b>
```

- In the case of <b>, applicable when the [ ] at the of the list is reached by foldXs, there is insufficient information to determine whether or not the new element e is to be inserted at this point. The solution is to return all possible results, for later determination when sufficient control information is available. Thus for the <b> parameter we return the pair of possible values "([ ], [e])".
- Then, in the case of the <op> applicable to any non-empty list, we have to synthesise a solution that deals with the necessary pair passed as a parameter (representing the result from the lower foldr result) and its connection with the corresponding pair returned as <op>'s result. Consistent with the result from <b>, the "ys" component of this second parameter to <op> refers to a result from which the inserted value "e" is absent, the "yse" component to one in which it is present (in the appropriate place). Synthesis of <op> has to maintain this relationship in the presence of the extra information at this level of recursion, namely the current list element 'y'. Thus, the first element of the result pair of <op>, i.e. that which would apply when higher-level control indicated that 'e' was not applicable, has to be "y:ys". The second element, being that which applies when higher-level control indicates that 'e' is applicable for insertion at this level, implements this insertion by comparing "e" with "y", duly exploiting as necessary the presence of 'e' in ""yse".
- Finally the top-level result is adapted to select the second element of the paired result, i.e. with 'e' inserted:

  ```
  insert e xs = snd $ foldXS xs <op> <b>
  ```

B. Functionalisation in Programming

Consider again binary tree indexing, where we have to solve for <np> and <lp> in

```
idxBT bt = foldBT bt <np> <lp>
```

- In the case of <lp>, the problem is that we don't know what index value, say "m", to provide to the leaf index constructor "Xl". Hence, this handler for a leaf node needs to be functionalized by parameterisation on the unknown index m, i.e. "(\ a -> (\ m -> Xl m))".
- Similarly in the case of <np>, the value provided to the node index constructor "Xn" also needs to be parameter. But in addition, <np> needs to maintain the relationship between indices at different levels of the tree, ie. it applies the knowledge that the index of subtree is to be incremented by one over its own current (but as yet unknown) index value.
- Finally, the functionalised result of the tree foldr needs to be applied to the value of the index parameter that is determined by context, i.e. 0:

  ```
  idxBT bt = foldBT bt <np> <lp> 0
  ```

C. Extension of Tupling and Functionalisation to Recursion Patterns

Essentially, once a sufficient basis of examples of the use of tupling and/or functionalization have been developed in specific cases, the usual process of abstraction applies to develop a generic example, i.e. the recursion pattern. For example, enough list insert-like cases will lead to the recognition of paramorphism; enough list rev(erse)-like cases will lead to the recognition of foldl, etc.

Critically, the functional language context places minimal obstacles on the functionalization technique and more broadly in the recognition of abstractions and the embodiment of recursion patterns in new higher-order functions that enables them to be reused conveniently.

VII. RELATED WORK

We have concentrated on catamorphic recursion patterns over regular recursive structures, and acknowledge that a similar treatment remains to be given for anamorphic recursion patterns that build structures. Again see [4] for an excellent treatment of how various catamorphisms and anamorphisms correspond, with algebraic laws for program calculation and verification.

While we are working to establish recursion patterns as an exclusive alternative to explicit recursion, practical use of recursion patterns is well-established. Foldr first appeared in APL [15] as the list reduce operation. We've noted above that foldr-ing over natural numbers is essentially FORTRAN DO-looping. In that context, Backus's embrace of foldr in his FP system [16] seems more of an evolution than the revolution it may have appeared to some. More generally, our attempt to restrict programming to a limited number of what are in effect control constructs is evocative of classical "structured programming" [17], save of course that foldr has been shown to be extensible to support new recursion patterns as required.

Finally it remains to develop a comprehensive subrecursive programming style. In a development that to some extent parallels [10], we have been working on a scheme that comprehensively replaces inert symbolic data with their functional or "zoetic" counterparts [18]. A key technique of this other kind of TFP ("Totally Functional Programming") is the partial application of foldr functions to data structures: the above justification of foldr is a key argument in validating both approaches to TFP.

VIII. CONCLUSIONS AND FUTURE DIRECTIONS

The key enabling result of this investigation has been to show that the devices by which the extended expressiveness of foldr can be realized, namely tupling and functionalization, and their uses, have comprehensible pragmatic motivations in terms of compensating for lack of information (either control or data) at various points in a recursion pattern. These motivations seem to be independent of potential options for employment by software engineers, in any of:

- informal development of foldr-based programs for subsequent verification using the various laws pertaining to foldr;
- calculation of foldr-based programs using these laws;
- development of more complex recursion patterns and the laws pertaining to same.

We acknowledge that it remains to for us to give a wide-ranging and concrete demonstration of these formalities, e.g. matching our implementations of recursive functions and recursion patterns in foldr, as well as a corresponding treatment of anamorphic recursion patterns.

Beyond that, now that the plausibility of foldr as *the* basic recursion pattern has been established, we can proceed to take it seriously as the basis of subrecursive TFP.

ACKNOWLEDGMENTS

We are grateful to contributors to other, earlier stages of the Totally Functional Programming project, in particular Colin Kemp whose PhD studies and resulting thesis [19] has provided indispensable contribution and stimulation.

REFERENCES

[1] S.C. Kleene, "Introduction to Metamathematics", Van Nostrand Rheinhold, 1952

[2] R. Bird, R. and P. Wadler, P., "An Introduction to Functional Programming", Prentice Hall, 1988

[3] E. Dijkstra, "A Discipline of Programming", Prentice-Hall, 1976

[4] E. Meijer, M. Fokkinga and R. Paterson, "Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire", Proc. FPCA 1991, LNCS, vol. 523, pp. 124—144, 1991

[5] S. Thompson, "Haskell: The Craft of Functional Programming". Addison-Wesley Longman, Harlow, 1996

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

[7] T. Uustalu, V. Vene, A. Pardo, "Recursion schemes from Comonads", Nordic Journal of Computing. Volume 8 , Issue 3 (Fall 2001). 366--390, 2001

[8] P. Bailes, "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

[9] J.E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics" MIT Press, Cambridge, Massachusetts, 1977

[10] D. Turner, "Total Functional Programming", Journal of Universal Computer Science 10 (7), pp. 751—768, 2004

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

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

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

[14] R. Bird, "Using circular programs to eliminate multiple traversals of data", Acta Informatica, 21:239 -250, 1984.

[15] K.E. Iverson, "A Programming Language", Wiley, 1962

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

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

[18] P. Bailes and C. Kemp, "Fusing Folds and Data Structures into Zoetic Data, "Proc. 23rd IASTED International Multi-Conference on Applied Informatics (AI 2005)", pp. 299-306. Acta Press, Calgary, 2005.

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