

Fusing Folds and Data Structures into Animated Data: towards a Totally Functional Programming Style

Paul Bailes
School of ITEE
The University of Queensland
QLD 4072 AUSTRALIA
+61 7 3365 2097
paul@itee.uq.edu.au

Colin Kemp
School of ITEE
The University of Queensland
QLD 4072 AUSTRALIA
+61 7 3365 2097
ck@itee.uq.edu.au

ABSTRACT

The difficulty of programming indicates the value of possible further reduction in the complexity of programming languages. Functional programs are simplified through the packaging of recursion into “fold” functions, but we are led to even further simplifications by conceiving of partially-applied folds in terms of providing animated views of otherwise inert data. Animations can indeed be found in more general structures than just pure folds, but fold-theory is critical in the synthesis of these wider applications. This animated style has a strong grounding in the theoretical development of functional programming and has interesting links to object-oriented programming, subrecursive and analog computing. Its identification moreover in some of the most striking applications of higher-order functional programming leads us to suspect that this animated style is the ultimate in functional programming.

Categories and Subject Descriptors

D.1.1 [Programming Techniques]: Applicative (Functional) Programming – fold, lambda-calculus

D.3.3 [Programming Languages]: Language Constructs and Features – extensibility, types

General Terms

Languages, Theory.

Keywords

Functional Programming, Language Extension, Subrecursion

1. AVOIDING INTERPRETATION IN PROGRAMMING

Despite much research into programming methodology, tools and languages, programming remains complex. Our contention is that one hitherto-overlooked factor is the complication brought about by programs’ and programmers’ needs to interpret or “animate”

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference ’00, Month 1-2, 2000, City, State.

Copyright 2000 ACM 1-58113-000-0/00/0000...\$5.00.

inert data. If data could be “pre-animated”, i.e. represented in such a way that the inherent underlying computation would not have to be revealed by interpretation, but rather be directly available, then at least some of the complexity of programming could be avoided.

Our ultimate goal accordingly is to discover a radical simplification of software development by exploring the functional paradigm to its limits. What distinguishes this ultimate style of functional programming is its replacement to the greatest extent possible of non-functional components, i.e. data and data structures, by appropriate functional alternatives that implement the essential applicative behaviour that is hypothesised to be inherent to the processing of each data (structure) type.

The approach that we take in what follows is to begin with functional programming, in which animated representations and operations thereon (functions, including higher-order functions) may at least be manipulated freely. We then consider a existing and proven stylised scheme for processing symbolic data – “fold” functions – and transform it into an actual view of data as animations, as opposed to hitherto inert objects. Direct definitions of animated interfaces to data types are relatively easily derived from specifications. We are careful not to suggest that animated programming would serve as a comprehensive substitute for the interpreted style, but rather that there seem to be (many) occasions when animation is highly advantageous. Perhaps in large systems, the two styles may need to co-exist, with overall improvement being effected by the choice wherever possible of animation over interpretation.

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

2. PROGRAMMING WITH FOLDS

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

2.1 Folds simplify Programming

A key insight in the development of programming methodology was that common programming patterns should be encapsulated by “structured” control constructs [1]. A key advantage of functional programming is that this encapsulation can be achieved

by programmer-definable higher-order functions [2]. Thus, the well-known list “fold” function [3] (earlier known as “reduce” in APL), definable (here, as throughout this paper, in Haskell [4]) as

$$\text{fold op } b \ [] = b$$

$$\text{fold op } b \ (x:xs) = \text{op } x \ (\text{fold op } b \ xs)$$

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

1. $\text{sum} = \text{fold } (+) \ 0$
2. $\text{map } f = \text{fold } (\lambda x \ xs \rightarrow f \ x : \ xs) \ []$
3. $\text{reverse} = \text{fold append } [] \ \text{where } \text{append } x \ xs = xs ++ [x]$

respectively define the functions which:

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

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

2.2 Folds support Formal Methods

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

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

One of the most useful such laws is the “fusion” property [4]: the identity

$$H (\text{fold } G \ W \ Xs) = \text{fold } F \ V \ Xs$$

holds provided that there also holds the conjunction of

$$H \ W = V$$

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

Consider for example, to prove that list concatenation is associative: “ $(A ++ B) ++ C = A ++ (B ++ C)$ ”. First, instead of the usual recursive definition (in pseudo-Haskell, with infix “ $++$ ” for concatenation):

$$[] ++ bs = bs$$

$$(a:as) ++ bs = a:(as ++ bs)$$

use rather a fold-based definition:

$$as ++ bs = \text{fold } (:) \ bs \ as$$

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

$$\text{fold } (:) \ C \ (\text{fold } (:) \ B \ A) = \text{fold } (:) \ (\text{fold } (:) \ C \ B) \ A$$

This now matches the terms of the fusion law, where

$$H = \text{fold } (:) \ C$$

$$G = (:) \ C$$

$$W = B$$

$$Xs = A$$

$$F = (:) \ C$$

$$V = \text{fold } (:) \ C \ B$$

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

1. $\text{fold } (:) \ C \ B = \text{fold } (:) \ C \ B$

(trivially)

2. $\text{fold } (:) \ C \ ((:) \ Y \ Ys) = (:) \ Y \ (\text{fold } (:) \ C \ Ys)$

iff (infixing “ $(:) \ Y \ Ys$ ” to “ $Y : Ys$ ”)

$$\text{fold } (:) \ C \ (Y : Ys) = (:) \ Y \ (\text{fold } (:) \ C \ Ys)$$

(which matches the definition of “fold” with $\text{op} = “(:)”$, $b = C$, $x = Y$ and $xs = Ys$)

2.3 Generality of Fold

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

Booleans:

$$\text{data Bool} = \text{True} \mid \text{False}$$

$$\text{foldB } t \ f \ \text{True} = t$$

$$\text{foldB } t \ f \ \text{False} = f$$

Natural numbers:

$$\text{data Nat} = \text{Succ Nat} \mid \text{Zero}$$

$$\text{foldN } s \ z \ \text{Zero} = z$$

$$\text{foldN } s \ z \ (\text{Succ } n) = s \ (\text{foldN } s \ z \ n)$$

Binary trees:

$$\text{data BinT } t = \text{Null} \mid \text{Leaf } t \mid \text{Branch } (\text{BinT } t) \ (\text{BinT } t)$$

$$\text{foldBT } n \ l \ b \ \text{Null} = n$$

$$\text{foldBT } n \ l \ b \ (\text{Leaf } elt) = l \ elt$$

$$\text{foldBT } n \ l \ b \ (\text{Branch } st1 \ st2) =$$

$$b \ (\text{foldBT } n \ l \ b \ st1) \ (\text{foldBT } n \ l \ b \ st2)$$

In general, for an ADT T with constructors $C1 \dots Cn$ where each

C_i has arity m and the j th operand is of type T_j (without loss of generality, this also applies for polymorphic T), i.e.,

$$\text{data } T = \dots | C_i T_{i1} \dots T_{im} | \dots$$

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

$$\text{fold}_T c_1 \dots c_n (C_i a_1 \dots a_m) = c_i A_1 \dots A_m$$

where

- each equation includes formal parameters c_i corresponding to constructors C_i
- the body of the equation for C_i applies corresponding parameter c_i to operands A_j , each in turn corresponding to the operands of C_i
- each operand A_j of c_i is derived from operands a_j of the prevailing C_i : if a_j corresponds to a nested element of T , then A_j is a_j “folded”, i.e. of the form “fold T $c_1 \dots c_n$ a_j ”; otherwise A_j is just a_j

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

$$\begin{aligned} \text{numnodes } \text{Null} &= 0 \\ \text{numnodes } (\text{Leaf } \text{elt}) &= 1 \\ \text{numnodes } (\text{Branch } \text{sub1 } \text{sub2}) &= \\ &\quad \text{numnodes } \text{sub1} + \text{numnodes } \text{sub2} \end{aligned}$$

is more simply defined as

$$\text{numnodes } t = \text{fold}_{BT} 0 (\text{elt} \rightarrow 1) (+) t$$

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

Booleans:

$$H (\text{fold}_B W_1 W_2 B) = \text{fold}_B V_1 V_2 B$$

provided that

$$H W_1 = V_1$$

$$H W_2 = V_2$$

Natural numbers:

$$H (\text{fold}_N G W N) = \text{fold}_N F V N$$

provided that

$$H W = V$$

$$H (G N) = F (H N)$$

Binary trees:

$$H (\text{fold}_{BT} W G_1 G_2 T) = \text{fold}_{BT} V F_1 F_2 T$$

provided that

$$H W = V$$

$$H (G_1 X) = F_1 X$$

$$H (G_2 U_1 U_2) = F_2 (H U_1) (H U_2)$$

and so on, for all regular recursive types.

3. INERT VS ANIMATED DATA

Recall that our underlying thesis is that the presence of inert symbolic data complicates programming. In order to understand how this complication reveals itself, we first consider what is the purpose of “data”? Apparently, to serve as models for objects being modelled in an application, and upon which operations of the application domain can be performed. However, there is at least one other purpose for data: as representations for operations, such as are processed, or “animated”, by interpreters. Attention to such representations may be thought of as being confined to interpretive implementations of programming languages, or to hardware instruction set design and implementation. However, such attention is objectively in fact a matter of interest to programming in general, as we reveal in the light of the nature and significance of language extensibility.

First, we show that language extension is a valid metaphor for programming in general. Next, we show that there are significantly different kinds of language extension. Finally, we consider how these different kinds of language extension project into different kinds of programming.

3.1 Language Extension is Significant

It emerges that language extension is not reserved for programming language designers and implementers, not does it only make programming language development accessible to “mere” programmers. Rather, it provides a basis for understanding and performing software development in general. That is, as well as language extension being inherently a kind of programming, it is also the case that programming is a kind of language extension.

3.1.1 “Program” = “Language”

Programming artefacts directly correspond to language design & extension artefacts. The correspondence between language design and programming is initially discernable in correspondences between the natures of the outputs of these processes. Language extension is implicitly included in the correspondence, as its products are the result of both programming and language design.

- Language longevity vs. software component longevity: our first observation is that the named (or otherwise identified) components of a software system remain objects of interest over extended durations throughout the entire life cycle (specification, implementation, maintenance) of a software system. In fact, the life cycle of a significant (in the sense that it actually gets used) software system, in the course of which its components are studied and learned by its developers and maintainers, outlives the vitality of some language designs of unquestionable significance (how many now care practically about Algol68 after less than 25 years?).
- Language complexity vs. software component complexity: a significant applications programming language typically provides at least 10^2 basic constructs. The number of components at the module, let alone procedure, level of a significant software system is at least as large.
- Language distribution vs. software component distribution: as well as the involvement of a succession of members of the one organisation in the development and maintenance of a software system during its lifetime, the phenomenon of customer sites wanting/needing to make local adaptations or emergency repairs is widespread, thus providing another

Comment [ITEE1]: best seen in context of LX: D vs I???

increase in the magnitude of the user population for a program's components. Moreover, the abovementioned hierarchical programming methodologies and the concept of software packages encourage a view of programmer-defined constructs as objects of interest to large user populations, just like those of programming languages.

3.1.2 "Programming" = "Language Design"

Further cause for associating programming and language design is the correspondence in the respective approaches taken to them by their practitioners. Our key observation [16] is that there is a correspondence between the criteria by which the quality of language designs is measured, compared to those by which the quality of construction of software systems is measured, as summarized below.

- Adequacy vs. hierarchy: the whole point of hierarchical program development is to bridge the gap between the actual expressiveness of a development's host language and the hypothetical expressiveness required of an application, and is therefore transparently an adequacy-enhancing language extension exercise.
- Orthogonality vs. loose coupling: orthogonal language constructs of the right type combinations etc. can be freely composed without exceptional behaviours in specific cases. Loose-coupling of modules is a means of reducing like accidental erroneous interactions in programs.
- Simplicity vs. cohesiveness: cohesive program components are easy to understand (because the relevant information is conveniently-located), and predictable in their use (at least partly because they will tend to be loose-coupled). Simple language constructs are likewise so because they are easily-described and tend towards orthogonality as a means of dealing with inherent complexities.
- Readability vs. nomenclature: the derivation of correct semantic cues from lexical appearance is the essence of a language's readability (e.g. use of familiar mathematical symbols). The ability of a reader to understand what a programmer has written similarly depends upon a programmer's choice of appropriate identifiers for his/her creations. Indeed, it seems hard to distinguish between the processes by which a language designer chooses an appropriately-suggestive keyword for a construct and by which a programmer chooses a name for a procedure/datatype/etc.

3.1.3 Basis in denotational semantics

The idea that "mere" programming is actually and objectively a kind of language design is further demonstrable via denotational semantics, which employed to reveal that declarations, the essence of any modular programming discipline, effect language extensions. First, consider the typical denotational meaning function M with signature " $M :: \text{Rep} \rightarrow \text{Envt} \rightarrow \text{Dom}$ " 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. Thus $\text{Envt} = \text{Id} \rightarrow \text{Dom}$. For example, the semantics of an identifier occurrence I is determined by accessing the environment ρ :

$$M \llbracket I \rrbracket \rho = \rho \llbracket I \rrbracket$$

Accordingly, the semantics of a (non-recursive) declaration can be expressed by the following equation for M that updates the prevailing environment ρ with the additional binding:

$$M \llbracket \text{let } I = E \rrbracket \rho = (\lambda i . \text{if } i = \llbracket I \rrbracket \text{ then } M \llbracket E \rrbracket \rho \text{ else } \rho \llbracket I \rrbracket)$$

Then, for a simple applicative language where an expression E is evaluated in the context of a declaration D , the top-level equation for M would be rendered

$$M \llbracket D ; E \rrbracket \rho = M \llbracket E \rrbracket (M \llbracket D \rrbracket \rho)$$

Now, consider simply interchanging the operands of M , i.e. changing the signature to " $M :: \text{Envt} \rightarrow \text{Rep} \rightarrow \text{Dom}$ ". Semantics of expressions are unchanged, save that operands of M are reordered, e.g.

$$M \rho \llbracket I \rrbracket = \rho \llbracket I \rrbracket$$

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

$$M \rho \llbracket \text{let } I = E \rrbracket = M (\lambda i . \text{if } i = \llbracket I \rrbracket \text{ then } M \rho \llbracket E \rrbracket \text{ else } \rho \llbracket I \rrbracket)$$

and semantics for a program rewritten consistently:

$$M \rho \llbracket D ; E \rrbracket = M \rho \llbracket D \rrbracket \llbracket E \rrbracket$$

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

$$\begin{aligned} M \rho &= MM \\ \text{where} \\ MM \llbracket D ; E \rrbracket &= MM \llbracket D \rrbracket \llbracket E \rrbracket \\ MM \llbracket \text{let } I = E \rrbracket &= \\ &M (\lambda i . \text{if } i = \llbracket I \rrbracket \text{ then } MM \llbracket E \rrbracket \\ &\text{else } \rho \llbracket I \rrbracket) \end{aligned}$$

In a real sense, MM (or " $M \rho$ ") 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, " $MM \llbracket D \rrbracket$ " defines MM extended by D . That is, declaration D truly extends language MM .

3.1.4 Conclusion: Programming as Language Extension

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, which contention is cemented from a formal semantic perspective.

3.2 Language Extension is Differentiated: Interpretation vs Definition

Having established that programming reveals itself as a kind of language extension, it mustn't be forgotten that the motivating concept of language extension is of course as a kind of programming, i.e. that enhancements to the design of some base

or host language can be implemented by “mere” programmers. Thus, important distinctions between different kinds of language extension can be identified in terms of the programming effort they entail.

3.2.1 Othophrase vs Metaphrase vs Paraphrase

The association of language extension with programming implies that language extensions should be as far as possible achieved by the sorts of means that a programmer would normally use. Standish [2] identifies the following kinds of extension:

1. *orthophrase*: the new “guest” constructs are achieved through extending the implementation of the “host” language;
2. *metaphrase*: guest constructs are modifications of host constructs achieved through corresponding modification of the host’s implementation;
3. *paraphrase*: guest constructs are defined in terms of existing constructs on the host.

Of the above, it is clear that the third, “paraphrastic” style is preferable on the ground that it precisely identifies the kind of language extension feasibly performed by programmers through declarations in the course of “normal” programming by normal programmers – the other two identify with specific software technologies requiring specialized skills. Further, because they require access to the language’s interpreter (by which we include implementation via compiler or other translator), they threaten the integrity of the resulting software, in that unintended changes in behaviour of host constructs may be effected.

3.2.2 Definitional Paraphrase vs Interpretational Paraphrase

The essence of our difficulty with non-paraphrastic extensions is that they involve interpretation. Accordingly, it’s important to recognise the further distinctions that exist within paraphrase between direct and indirect, interpretive extensions. It follows from the Church-Turing thesis that any effective host language will serve as a host for any conceivable (and computable) guest, but this allows for indirect paraphrastic definitions through encoding guest constructs as data with corresponding interpretation. Direct paraphrase on the other hand is where the semantics of guest constructs are directly represented in actual host language terms.

While it follows from Turing’s thesis etc. that any computable function can be interpreted in a Turing-complete host language, direct definition is the form usually performed by programmers in extending languages by means of declarations (as above). We note in passing that while from a “Turing” point of view, virtually all programming languages are the same, it is possible to differentiate between them in terms of the direct definitions (as opposed to the interpretations) of which they are capable of expressing [14]. In particular, higher-order functions such as the various partial applications throughout this paper of “fold” and its relatives are not expressible in first-order languages such as C, Java, etc.

3.3 Programming is Differentiated: Interpretation vs Definition

The implication for programming (as a kind of language extension) is that just as language extension should be prosecuted (executed, evaluated and constrained) as a kind of programming

(i.e., definitionally not interpretationally), so should programming be prosecuted as a kind of (paraphrastic) language extension. Different kinds of (paraphrastic) language extension will be reflected in different kinds of programming. Now, if a particular style of language extension is to be avoided, so should the corresponding kind of programming. If extension via direct definition is to be preferred over extension by interpretation, so should programming via direct definition be preferred over programming by interpretation.

Because interpretation is all about animating operations from inert functional representations, it follows that a preferred direct-definitional programming style should be all about providing pre-animated representations for what we previously inert data.

4. FOLDS ANIMATE DATA

“Fold” functions reveal themselves as more than just a notational simplification, but as a window on this new view of data as animated, as opposed to inert objects.

4.1 Inverting Fold

First, it’s straightforward to re-order the operands to “fold”.

Consider how as commonly-presented (e.g. above), “fold” is an abstraction of a particular data structure from a pattern of operations on the structure. As above, the partial application

$$\text{fold } O B$$

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

$$\backslash xs \rightarrow \text{fold } O B xs$$

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

$$\backslash op b \rightarrow \text{fold } op b Xs$$

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

$$\text{ifold } xs \text{ op } b = \text{fold } op b xs$$

or explicitly recursively

$$\text{ifold } [] \text{ op } b = b$$

$$\text{ifold } (x:xs) \text{ op } b = op x (\text{ifold } xs \text{ op } b)$$

Thus:

$$\text{ifold } Xs$$

Obviously, corresponding to the folds “foldT” that exist for all types T, there may be defined inverted folds “ifoldT”:

$$\text{ifoldT } (C_i a_1 \dots a_m) c_1 \dots c_n = c_i A_1 \dots A_m$$

where, as for the general “foldT”

- each equation includes formal parameters c_i corresponding to constructors C_i
- the body of the equation for C_i applies corresponding parameter c_i to operands A_j , each in turn corresponding to the operands of C_i

but where different in the inverted fold

- each operand A_j of c_i is derived from operands a_j of the prevailing C_i : if a_j corresponds to a nested element of T , then A_j is a_j “folded” but inverted, i.e. of the form “ $ifoldT\ a_j\ c_1\ \dots\ c_n$ ”; otherwise A_j is just a_j

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

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

provided that as before

$$H\ W = V$$

and

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

4.2 Inverted Partial Applications of Fold

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

The hitherto conventional view of a data structure, as a complex of typically nested applications of constructors, is thus revealed as a special case of an inverted fold application – in this case, to the constructors themselves! For example

$$ifold\ [x1,\ x2,\ x3]\ (\cdot)\ []$$

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

$$ifold\ (x1:x2:x3:[])\ (\cdot)\ []$$

= (applying “ifold” to “x1:x2:x3:[]”)

$$(\text{op}\ b \rightarrow \text{op}\ x1\ (\text{op}\ x2\ (\text{op}\ x3\ b)))\ (\cdot)\ []$$

= (substituting for “op”, “b”)

$$(\cdot)\ x1\ ((\cdot)\ x2\ ((\cdot)\ x3\ []))$$

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

$$x1:x2:x3:[]$$

= (reverting to usual notation)

$$[x1,\ x2,\ x3]$$

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

$$ifoldT\ D\ C1\ \dots\ Cn = D$$

or equivalently, expanding D into some construction of one of the constructors C_i : “ $(C_i\ \text{opds}\ \dots)$ ”

$$ifoldT\ (C_i\ \text{opds}\ \dots)\ C1\ \dots\ Cn = (C_i\ \text{opds}\ \dots)$$

(see also “General Properties of Animated Data” below).

4.3 Inverted Partial Applications as Animated Data

The partial application “ifold Xs” therefore reveals an inherent applicative behaviour of lists: as functions, that apply to a binary operation O and a “base element” B, and apply O successively to elements of the list and the result of its application to the

remainder, with B as result on the “Nil” case. Upon reflection, it is apparent that this behaviour captures the essence of a (finite) list, as an ordered sequence of elements:

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

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

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

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

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

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

4.4 Programming with Animated Data

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

Lists

For animated lists l (corresponding to “ifold L” for some lists L), simply define e.g. operations

$$\begin{aligned} \text{sum}\ l &= l\ (+)\ 0 \\ l1\ ++\ l2 &= l1\ (\cdot)\ l2 \end{aligned}$$

Booleans

For animated booleans b (corresponding to “ifoldB B” for some booleans B), simply define e.g. operations

$$\text{not}\ b = b\ \text{False}\ \text{True}$$

Comment [ITEE2]: say something now about what this “means”, before the concrete case of the next para.

and b1 b2 = b1 b2 False
or b1 b2 = b1 True b2

Natural numbers

For animated naturals (corresponding to “ifoldB B” for some booleans B), simply define e.g. operations

add n1 n2 = n1 Succ n2
mult n1 n2 = n1 (add n2) Zero

Binary trees

For animated trees t (“ifoldBT T” for some tree T), simply define e.g.

numnodes t = t 0 (elt → 1) (+)
concatnodes t = t [] (elt → [elt]) (++)

In all these cases, the animated representations apply their inherent behaviours to operands to produce the results as further determined by the operands. For each type (lists, trees, etc.) however, the way in which the operands are composed is the same as determined by the animation supplied to the type by its (inverted) fold. No analysis or interpretation of the operands is required, the pattern of composition being inherent in the animation – this is why the definitions are so simple.

Note however that as defined above, these operations all yield “original-style” inert data. In order to animate the results, it is merely necessary to apply the appropriate “ifold” to the result. A more elegant solution that avoids even this minor inconvenience emerges in the context of “generators” below.

4.5 General Properties of Animated Data

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

$$\text{ifoldT } D \text{ } C_1 \dots C_n = D$$

is derived from a deeper relationship between animated data and their operands. Specifically, it follows from the definitions of “ifoldT” that applying animated data derived from constructions of some C_i , to operands $X_1 \dots X_n$, results in an the X_i corresponding to C_i , possibly applied to operands as determined by the structure of the type and reflected in the definition of the (inverted) fold.

Formally:

$$\text{ifoldT } (C_i \text{ opdsC } \dots) X_1 \dots X_n = (X_i \text{ opdsX } \dots)$$

where

- C_i denotes one of the (original) data structure constructors for type T
- “ $(C \text{ opdsC } \dots)$ ” denotes a construction (D above), i.e. application of C_i to operands “opdsC ...” to construct a value of T
- “ $\text{ifoldT } (C_i \text{ opdsC } \dots)$ ” therefore creates the animated version of this value
- $X_1 \dots X_n$ denote operands of animated data, note the correspondence with the constructors $C_1 \dots C_n$ of T (following the definition of fold)
- “ $(X_i \text{ opdsX } \dots)$ ” denotes application of X_i to operands “opdsX ...”

...”, as determined by the equation of foldT for C_i

- note that while “opdsC ...” and “opdsX ...” are not the same, they are not arbitrarily distinct, the relationship between being determined by the equation of “foldT” for C_i , which as we have seen follows the type-signature of C_i .

The reason why (as from further above)

$$\text{ifoldT } D \text{ } C_1 \dots C_n = D$$

or equivalently

$$\text{ifoldT } (C_i \text{ opds } \dots) C_1 \dots C_n = (C_i \text{ opds } \dots)$$

is that the operands “opds ...” to C_i are the same on both sides. This is clearly not generally the case, viz. the distinction between “opdsC ...” and “opdsX ...” just above. However, following the definition of “ifoldT”, it’s clear that the difference between “opdsC ...” and “opdsX ...” is just that in the latter, “ifoldT” is applied to recursive/nested occurrences of operands to C_i along with the other operands to “ifoldT”. Thus, when these other operands are the C_i themselves, the individual members of “opdsX ...” are transformed into the corresponding “opdsC ...”.

For example:

$$\begin{aligned} &\text{ifoldBT } (\text{Branch } T_1 \text{ } T_2) \text{ } \text{Null Leaf Branch} \\ &= \\ &\text{Branch} \\ &(\text{ifoldBT } T_1 \text{ } \text{Null Leaf Branch}) \\ &(\text{ifoldBT } T_2 \text{ } \text{Null Leaf Branch}) \\ &= \\ &\text{Branch } T_1 \text{ } T_2 \end{aligned}$$

because each the applications

$$\text{ifoldBT } T_i \text{ } \text{Null Leaf Branch}$$

($i = 1 \ 2$) will correspondingly transform into T_i .

5. DIRECT GENERATION OF ANIMATIONS

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

5.1 Animated Sets

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

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

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

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

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

The “inverted fusion” law for sets is

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

provided that

$$H R = S$$

$$H (G1 X) = F1 X$$

$$H (G2 R1 R2) = F2 (H S1) (H S2)$$

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

5.2 Composing Impure Animations with Generators

5.2.1 Composing Animations

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

For example, given

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

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

$$member (Union S1 S2)$$

rather than in terms of some direct “union” of the animated SAi, i.e.

$$union SA1 SA2$$

The problem is that aside from the inconvenience of having to retain inert data whilst programming in “animated mode”, it’s not

guaranteed that in a non-trivial program, constructions Si would be visible in all contexts in which animations SAi are required.

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

$$ifoldT D C1 \dots Cn = D$$

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

5.2.2 Generators

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

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

$$member (Ci opdsC \dots) = (Gi opdsG \dots)$$

where

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

Thus, the specification for sets is:

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

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

5.2.3 Derivation of Generators

Now, using the specific identity that for any type T

$$ifoldT D C1 \dots Cn = D$$

i.e.

$$ifoldT (Ci opdsC \dots) C1 \dots Cn = (Ci opdsC \dots)$$

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

$$member (ifoldS (Ci opdsC \dots) C1 \dots Cn)$$

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

$$ifoldT (Ci opdsC \dots) X1 \dots Xn = (Xi opdsX \dots)$$

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

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

$\begin{aligned} \text{member } ((\text{ifoldS } \text{Empty}) \text{ Empty Single Union}) \\ &= \text{ifoldS } \text{Empty } \text{empty single union} \\ \text{member } ((\text{ifoldS } (\text{Single } X)) \text{ Empty Single Union}) \\ &= \text{ifoldS } (\text{Single } X) \text{ empty single union} \\ \text{member } ((\text{ifoldS } (\text{Union } S1 \text{ } S2)) \text{ Empty Single Union}) \\ &= \text{ifoldS } (\text{Union } S1 \text{ } S2) \text{ empty single union} \end{aligned}$
--

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

$\begin{aligned} \text{member } ((\text{ifoldS } S) \text{ Empty Single Union}) = \\ (\text{ifoldS } S) \text{ empty single union} \end{aligned}$

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

$\begin{aligned} \text{member } \text{Empty} &= \text{empty} \\ \text{member } (\text{Single } X) &= \text{single } X \\ \text{member } (\text{Union } S1 \text{ } S2) &= \\ &\text{union } (\text{member } S1) (\text{member } S2) \end{aligned}$
--

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

$\begin{aligned} \text{empty } \text{elt} &= \text{False} \\ \text{single } x \text{ elt} &= x = \text{elt} \end{aligned}$
--

The derivation of the third generator

$\text{union } s1 \text{ } s2 \text{ elt} = s1 \text{ elt or } s2 \text{ elt}$
--

requires however further explication. Noting that the forms “member Si” denote the desired impure animated sets, proceed

$$\begin{aligned} &\text{union } s1 \text{ } s2 \\ &= (\text{as required by fusion}) \\ &\quad \text{member } (\text{Union } S1 \text{ } S2) \\ &= (\text{following “member”}) \\ &\quad \backslash \text{elt} \rightarrow \text{member } S1 \text{ elt or member } S2 \text{ elt} \\ &= (\text{re-expressing “member Si” as the corresponding animated set}) \\ &\quad \backslash \text{elt} \rightarrow s1 \text{ elt or } s2 \text{ elt} \end{aligned}$$

as required. Note how the operands of generator “union” are, correctly, the animated versions of sets.

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

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

1. specify the structure of the type (“signature”);
2. select and specify the selector operation O on the type that

provides the characteristic behaviour of the animated data;

3. to the identity “O (ifoldT D C1 ... Cn)” = “ifoldT D G1 ... Gn”, apply fusion to relate Gi to defining equations for O on constructions from corresponding Ci;
4. solve these relations to yield defining equations for Gi (provided that solutions exist – it’s not guaranteed that they always will, in which case other derivation paths for impure animations remain to be researched).

Comment [ITEE3]: they exist when the fn. is fold-expressible, i.e. anything other than “cdr” etc. or requiring multi-passes (what does, if anything?)

5.3 Generators for Pure Animations

Now that the necessity of generators has been achieved for impure animations, it’s attractive to provide their convenience for pure animations. These generator definitions are straightforward, noting that account must be made taken for the fact that nested structures will be the animated version. For example, for lists, we seek generators “nil” and “cons” corresponding to the obvious counterparts. From the specifications

$$\begin{aligned} \text{nil} \\ &= \text{ifold Nil} \\ &= \backslash \text{op } b \rightarrow b \end{aligned}$$

and

$$\begin{aligned} \text{cons } x (\text{ifold } xs) \\ &= \text{ifold } (x : xs) \\ &= \backslash \text{op } b \rightarrow \text{op } x (\text{fold } xs \text{ op } b) \end{aligned}$$

we can thus derive defining equations

Lists:

$\begin{aligned} \text{nil op } b &= b \\ \text{cons } x \text{ xs op } b &= \text{op } x (\text{xs op } b) \end{aligned}$
--

Generators for the other “pure” types considered above follow similarly.

Booleans:

$\begin{aligned} \text{true } t f &= f \\ \text{false } t f &= f \end{aligned}$

Natural numbers:

$\begin{aligned} \text{zero } s z &= z \\ \text{succ } n s z &= s (n s z) \end{aligned}$
--

Binary trees:

$\begin{aligned} \text{null } n l b &= n \\ \text{leaf } \text{elt } n l b &= l \text{ elt} \\ \text{branch } \text{sub1 } \text{sub2 } n l b &= b (\text{sub1 } n l b) (\text{sub2 } n l b) \end{aligned}$

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

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

$\text{ifoldT } D \text{ } G1 \text{ ... } Gn = \text{ifoldT } D$

or more generally

Comment [ITEE4]: are these right???

$(Gi \dots) Xi \dots Xn = (Xi \dots)$

where

- Gi denotes the generator corresponding to some Ci
- “*ifoldT D*” as usual identifies the pure animation of D
- “*(Gi ...)*” denotes application of Gi to operands to construct an animated counterpart to a member of T
- “*(Xi ...)*” denotes application of Xi to operands as determined by the equation for Gi , in turn derived from that of *foldT* for the corresponding Ci

6. TOTALLY FUNCTIONAL PROGRAMMING IN CONTEXT

Recall our ultimate goal is to simplify programming by obviating the need to interpret inert, symbolic data through the use of animated representations. Some positive indicators of the validity and potential of this approach though its implicit appearance and connections to other themes in computer science are as follows.

6.1 History of Animated Data

The original “Church numeral” representations of natural numbers in the lambda-calculus [5] were in terms of the generators “zero” and “succ” above. Thus, a natural N would be represented as N -fold composition; in other words, the computation inherent in a number N , as reflected in the animated representation, is iteration. Examples of simplified animated programming in this representation include definitions of basic arithmetic operations:

Natural numbers:

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

These definitions are isomorphic to those given far above for naturals animated with “fold”, save that whereas the former definitions applied constructors “Succ” and “Zero” and yielded inert data, these new definitions apply generators “succ” and “zero” and yield animations.

For other types similarly, yielding animated data by using animations and applying generators rather than constructors throughout, e.g.:

Lists:

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

Booleans:

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

Binary trees:

numnodes t = t zero (\wedge elt \rightarrow succ zero) add
concatnodes t = t nil (\wedge elt \rightarrow cons elt nil) (++)

In general, the pure animations of data and their definition in terms of generators have been known as long as Reynolds’

second-order polymorphic typed-lambda-calculus [6], which incidentally provides a type system adequate to the demands of animated programming. Likewise, impure animations such as characteristic predicates for sets are well-known (see also “Applications of Animated Data” below). This history does not however extend to our presentation of the connections between pure and impure animations in terms of folds, and in particular of their significance in the context of a comprehensive animation-based programming style and its motivation from a programming-as-language-extension perspective.

6.2 Applications of Animated Data

As well as the extensive use of pure animations in functional programming (albeit disguised as folds), there are some significant examples of impure animations.

- In “combinator parsers” [7] the usual combination of a grammar (or equivalent representation such as a parse table) and its interpreting “parsing engine” are replaced by a collection of parsers, one for each nonterminal symbol in the grammar. Context-free operations of grammar concatenation and alternation are implemented by appropriate higher-order functions on parsers. Compare this with the conventional situation where a grammar is a static data structure awaiting interpretation by a parsing engine.
- Exact real arithmetic [8] may be achieved when a real number is represented by a function which computes a real to any required rational precision.
- Programmed graph reduction [9] of functional languages represent functions not as graphs into which substitutions are to be performed, but as a program which constructs the substituted graph.

6.3 Relationship with Object-Orientation

There is an apparent positive connection between animated programming and some of the origins of OOP, in that one conception of methods on objects is as portals to which messages are sent and acted upon by the object, rather than as operations that manipulate the object. For example, in the animated definition of “add” above, a number m takes two messages – an operation “succ” and another number n , and replies with $m+n$. For a whimsical yet extensive development of the analogy between functional programming and message-passing, see Smullyan [10].

On the other hand, impure animations as characterised above admit only a single extractor method besides generators. Animations correspond to objects upon which a single method is defined – the applicative behaviour of the object, i.e., the behaviour inherent to the data that the animation embodies, *is* the method. Consider for example how the single “member” method on sets was subsumed into the characteristic predicate representation. For natural numbers, the single method is iteration, etc. for the other types considered. In contrast, OOP permits numerous methods to be defined on objects. We hope for a presentation of animation derivation that relates to a notion of the cohesiveness of methods for an object/class in order to justify single-method objects/classes – see also “RESEARCH DIRECTIONS” below Q10. “Is TFP compatible with the rest of software engineering?” for more discussion of how we propose to achieve reconciliation of TFP and OOP.

Comment [ITEE5]: I thought this was wrong, but the following “repair” doesn’t seem correct, so maybe OK (if too simplistic for clarity)
 e.g.
 (cons E Es) X1 X2
 =
 X1 E (Es X1 X2)

 “surely wrong? More like (Gi Xi1 ... Xin) Y = [[combination (not necly application, unless Xij are ()PCs) of Xij to Y depending on Gi]] e.g. (singleton X) Y = X=Y”

7. RESEARCH DIRECTIONS FOR ANIMATED DATA

A constructive substantiation of our hypothesis involves the development of totally functional programs as well as practical development methods and implementation techniques. The ten key questions for further investigation are as follows, grouped into topics relating to conceptual foundations, technical infrastructure to ensure the practicality of TFP, and extended application of TFP.

7.1 Foundations

Q1. How much of FP is actually TFP? We anticipate that understanding of the extent to which TFP can be discerned in advanced examples of FP beyond those cited above (combinator parsers etc.) will illuminate solutions to several of the other questions posed here, the idea being that powerful examples of higher-order function usage can actually be perceived as generation and manipulation of animations.

Q2. What is the precise distinction between Definition and Interpretation? In general, the universality of data-less functional languages (such as the lambda-calculus) means that simply abolishing data is insufficient to prevent the writing of interpreters, and in particular there are simple lambda-terms that appear to treat functions as symbolic data (e.g. testing for specific behaviours, such as zero-testing of Church numerals). What is a precise, even syntactic characterization of “interpretation”, at least so that we know certainly what TFP is, and so that a subset of FP languages can be identified that will be adequate to support TFP? NB by “defining out” the capacity to write interpreters, it would seem obvious that such a subset would be subrecursive [15].

A syntactic characterization of definition vs. interpretation will implicitly define a conformant (probably subrecursive) functional language subset, which can then be used as a basis for further technological and methodological developments, particularly with respect to Q3-8. Our current expectation is that we will be able to identify an (illegitimate) functional simulation of data as being one in which a function leaves no residual in the result of its application.

Q3. What is the semantic model for TFP? An elegant semantics for any FP subset for TFP would serve to validate the characterization of interpretation vs definition, as well as providing a good technical foundation for any specialized approach to verification of TFPs. In the long run, a relatively simpler semantic model may facilitate the development of more automated derivation/verification/transformation tools, as well as dictating the expressiveness of the FP subset for TFP. There are numerous domain constructions alternate to the usual “Scott” domain and its derivatives. Which to develop will depend upon the outcome of Q2.

7.2 Infrastructure

Q4. How may functional alternatives for data be derived systematically? What are the limits of the “fusion” technique for impure animation derivation that is elaborated above, and what techniques are necessary and feasible otherwise.

Q5. How may TFP be type-checked? Experience with even some

elementary TFP examples (arithmetic operations on natural numbers) indicates that more powerful type-checking than usual in FP in required. Reynolds’ second-order polymorphic types [6] admit a much-wider class of functions than standard Hindley-Milner type checking [11], but suffer the practical drawback of requiring explicit nomination of types by programmers, rather than allowing the convenience of type inference. What advances in type inference are mature enough for employment here? It’s known that second-order polymorphic type inference is ultimately undecidable, but that for important subclasses (e.g. “rank 2”) this is not the case [12]. Even if “rank 2” is suitable for TFP, a better implementation than is currently available [13] is indicated, that avoids unnecessary encoding of functions as data structures!

The suitability to TFP of “rank 2” 2nd-order polymorphic types needs to be assessed, and a more integrated implementation produced.

Q6. How may functional language implementations be optimized for TFP? If TFP can indeed be identified with a functional language subset, then it’s possible that the full generality of functional language implementation infrastructure may not be required, i.e. more efficient implementations may be possible by exploiting the functional language subset. It’s impossible at this stage to speculate further, beyond promising optimisations for built-in operations and additional primitives. Further specifics will have to await the outcome of Q1. and Q2. at least.

Q7. What additional primitives are necessary? It emerges that the TFP language subset will almost certainly be subrecursive (see above), in which case it may be useful to select carefully additional useful primitive operators to ensure applicability of our subrecursive functional language in popular domains.

It’s well-established that a usefully-large class of arithmetic is available for TFP (e.g. from 2nd-order polymorphic typed lambda-calculus) and presumably counterparts for other simple types (e.g. lists), but we may have to invest some effort in discovering additional appropriate operations (e.g. parallel operators?) to achieve expressive completeness with respect to the semantic domain (3. above). For example, just as expressive completeness with respect to the Scott domain induces a need to introduce parallel logical connectives to functional languages [14], what may be necessary for the domain for TFP (see Q3. above)?

Q8. How may functional language verification and validation be optimized for TFP? Just as dealing with an FP subset may allow implementation optimization, so may subsetting permit simpler proofs, which we would expect to be founded in the simpler semantic model that is also sought (again Q3. above).

7.3 Extension

Q9. Is TFP more widely-applicable? By basing computation on the applicative/dynamic properties of abstractions, rather than on the interpretation of symbols, TFP may have wider applicability than just computer programming, i.e. in other domains where the composition of systems is defined in terms of the behaviours of their components.

For example, a programming style based on composing components characterized entirely by their (applicative) behaviours, as does TFP, holds promise as a basis for modeling analog electronic design and broader systems engineering. Another avenue to explore is software reverse engineering/design recovery, where the relevant problem is that interpretation of data can obfuscate the actual function of a software system.

Q10. Is TFP compatible with the rest of software engineering?

We've already emphasized that the two styles, "data-less"/definitional TFP and "data-full"/interpretational programming need to co-exist. More generally, programming methodologies that are in their essentials independent of computation model (e.g. object-oriented design should be accessible from procedural, logic, functional, etc. programming) should equally be accessible from within TFP.

Of course, the complete abolition of symbolic data would appear to be an unreasonable goal. This is why we emphasise mixed development: interfacing between totally functional components (pure animations) and those in which data and functional representations co-exist, and indeed hybrids in which TFP representations may retain attributes pertaining to symbolic data (impure animations).

While the admission of impure alongside pure animations would at least allow definitional TFP to be introduced gradually alongside conventional definitional programming, there is at least an apparent unresolved conflict, with object-oriented programming, as noted above. Likely complementary avenues for reconciling TFP and OOP include such as: from the TFP point of view, discovery of higher-order generators from which different methods permitted by OOP can be instantiated; from the OOP point of view, a better understanding of the limits on the diversity that be accommodated within an object's methods for the sake of cohesion, and alignment of that understanding with the derivation of the higher-order generators we seek for TFP. We hypothesise that for each class/type a single "mother extractor" can be found from which any others apparently desired could be synthesised, but this remains to be validated.

8. SUMMARY AND CONCLUSION

Our animated approach develops conventional functional programming in two senses of the word "total": (i) it tends to eschew data, so that programming tends to be "totally" concerned with functions; (ii) it tends also to avoid recursion, i.e. such programs (or components) tend to be total rather than partial functions, and thereby suggests a link to "subrecursive" programming [11]. Ultimately, a convergence with analog computing (where computations are likewise "animated", i.e. structured around the behaviours of their components) may be discovered and exploited.

A new approach to programming that eschews inert symbolic data for animated representations has much to commend itself: (i) as an adaptation of an established functional programming structure (folds); (ii) grounded in appropriate theory (e.g. 2nd-order

polymorphic typed-lambda calculus); (iii) recognisable practical applications; (iv) links with other broad themes in computing; and (v) well-defined paths for development.

9. ACKNOWLEDGEMENT

We are grateful to our co-workers for their influence on the above, notably Ian Peake and Sean Seefried.

10. REFERENCES

- [1] Dijkstra, E.W. "Goto Statement Considered Harmful", *Comm. ACM*, vol. 11 no. 3, pp. 147-148 (1968).
- [2] Hughes, J., "Why Functional Programming Matters", *The Computer Journal*, vol. 32, no. 2, pp. 98-107 (1989).
- [3] Hutton, G., "A Tutorial on the Universality and Expressiveness of Fold", *Journal of Functional Programming*, vol. 9, no. 4, pp. 355-372 (1999).
- [4] <http://www.haskell.org>
- [5] Barendregt, H.P., "The Lambda Calculus - Its Syntax and Semantics", North-Holland, Amsterdam (1984).
- [6] Reynolds, J.C. "Three approaches to type structure", in *Mathematical Foundations of Software Development*, LNCS Vol 185, Springer-Verlag (1985).
- [7] Hutton, G., "Parsing Using Combinators", *Proc. Glasgow Workshop on Functional Programming*, Springer (1989).
- [8] Boehm, H. and Cartwright, R., "Exact Real Arithmetic: Formulating Real Numbers as Functions", in Turner, D.A. (ed.), *Research Topics in Functional Programming*, Addison-Wesley (1990).
- [9] Peyton Jones, S., "The Implementation of Functional Programming Languages", Prentice-Hall International, Hemel Hempstead (1987).
- [10] Smullyan, R., "To Mock a Mockingbird", Alfred A. Knopf, New York (1985).
- [11] Milner, R., "A Theory of Type Polymorphism in Programming", *J. Comp. Syst. Scs.*, vol. 17, pp. 348-375 (1977).
- [12] Wells, J.B., "Typability and Type Checking in the Second-Order Lambda-Calculus are Equivalent and Undecidable", *Logic in Computer Science*, pp. 176-185 (1994).
- [13] Jones, M.P., "First-class Polymorphism with Type Inference", *Proc. Symposium on Principles of Programming Languages* (1997).
- [14] Plotkin, G.D., "PCF Considered as a Programming Language", *Theoretical Computer Science*, 5, pp. 223-255 (1977).
- [15] Royer, J.S., & Case, J., "Subrecursive Programming Systems: Complexity & Succinctness", Birkhauser (1994).
- [16] P.A. Bales, "The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design)", *Proc. 1986 Austrln. Software Eng. Conf.*, pp. 14-18 Canberra (1986).