Obstacles to a Totally Functional Programming Style

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

Abstract

"Totally Functional Programming" (TFP) advocates the complete replacement of symbolic representations for data by functions. TFP is motivated by observations from practice in language extensibility and functional programming. Its technical essence extends the role of "fold" functions in structuring functional programs to include methods that make comparisons on elements of data structures. The obstacles that currently prevent the immediate uptake of TFP as a style within functional programming equally indicate future research directions in the areas of theoretical foundations, supporting technical infrastructure, demonstrated practical applicability, and relationship to OOP.

1. Introduction

"Totally Functional Programming" (TFP) refers to a subset style of conventional functional programming (FP) which takes the latter's exploitation of programmer-definable higher-order functions to its extreme, through an as complete-as-possible eschewing of symbolic data in favour of representations by functions. Even though TFP appears to have well-founded motivations, and technically appears able to co-exist with any residual necessity for data *per se*, there appear to be some serious obstacles to its practical adoption.

After summarising the conceptual bases for TFP, and the essential technicalities, we are in a position to explain the obstacles in detail, which can be broadly classified as:

- *theoretical*: is TFP well-founded?
- *infrastructural*: what supporting technology is needed for practical TFP?
- *practical*: assuming all the above can be overcome, what practical use is TFP?
- *co-existence:* finally, can TFP be used with other paradigms, as typified by OOP?

Nevertheless, these obstacles also point to opportunities, for if surmounted, the promise of TFP would be appear to be that much more realisable.

2. Bases for TFP

TFP is motivated/justified by at least three independent means (see our [1] for more detail):

- generally, experience to date with FP;
- specifically, the role played by "fold" functions in structuring functional programs;
- also, by the essential affinity between programming and language design/extension.

2.1 Higher-order functions in FP

In motivating FP, the essential defining characteristic – the ability for programmers to define arbitrary higher-order functions (HOFs) which take functions as arguments and also *produce functions as results* - must be the focus. Prevailing popular justifications which focus on issues such as verification properties [2] or decomposability [3] can be thought of as exploiting this HOF-capability (respectively to define recursion and lazy evaluation), but do not engage sufficiently explicitly with the expressive power of the HOF-capability itself.

However, there are several examples of FP that do exploit HOFs, and it's significant how they involve functional representations for data:

- *combinator parsers*: [4] 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*: [5] may be achieved when a real number is represented by a function which computes a real to any required rational precision;
- *programmed graph reduction*: [6] of functional languages represent functions not as graphs into which substitutions are to be performed, but as a program which constructs the substituted graph

- *characteristic predicate representations of sets*: are
- demonstrable significance as a mathematical modelling concept.

We are thus prompted by these powerful demonstrations to speculate that FP, as characterised by the HOF-capability, at least embraces replacement of inert symbolic data by functions that capture a desired behaviour that is perhaps characteristic of the datatype. More systematic justification follows below.

2.2 Structuring functional programs with "folds"

The "fold (right)" function over lists (NB we use Haskell [18] notation wherever possible):

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

has proven to be surprisingly convenient for defining a wide range of list processing operations [7]. It emerges that analogous "fold" functions exist for every "regular recursive" datatype, e.g.

- natural numbers:

```
data Nat = Succ Nat | Zero

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

- binary trees:

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

foldT b l n  Null = n
foldT b l n (Leaf v) = l v
foldT b l n (Branch t1 t2) =
            b (foldT b l n t1) (foldT b l n t2)
```

- etc. Indeed there is a straightforward correspondence between the signature of a type and its "fold" [7].

These generalised folds are similarly useful for structuring functions over these datatypes in addition to lists, to the limits of the expressiveness of folds [8]. It's therefore arguable that the norm for FP is, as far as possible, to write programs in terms of folds. For example, instead of:

```
length [ ] = 0
length (x:xs) = 1 + length xs

size Null = 0
size (Leaf v) = 1
size (Branch t1 t2) = size t1 + size t2
```

write

```
length xs = fold (1+) 0 xs

size t = foldT (+) 1 0 t
```

Now, it's perfectly reasonable to apply some simple, essentially syntactic restructuring of the "fold" definitions. comparatively simple to the above, but are of First, invert the order of operands so that the structure being folded appears first, e.g.

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

ifoldN Zero s z = z
ifoldN (Succ n) s z = s (ifoldN n s z)

etc.
```

Second, and critically, fuse the inverted fold with what is now its first argument – the data structure to which it applies, e.g.

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

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

etc.
```

That is, rather than applying (inverted) folds to applications of data structure constructors, apply counterparts of the constructors – say "generators" – which have the effect of automatically folding their operands (which in turn will be applications of generators rather than constructors. For example:

```
length xs = xs (1+) 0
size t = t  (+) 1 0
```

Observe how data structures are thus represented as functions, in a manner similar to the functional representations for grammars, sets etc. above. In other words, fold-based functional programming equates to programming with functional, rather than symbolic, representations of data.

2.3 Programming = language extension

The idea underlying the above is that symbols, and the necessity within programs to interpret them, can be replaced by functional representations embodying a characteristic behaviour. This idea can be explained in a general context – of a perception of programming as a kind of language extension, and a consequent realisation that programming is pervaded by interpretation which it's desirable to avoid as much in programming as it is in language extension.

2.3.1. Language extension as programming. This is a proposition that follows from the basic idea of language extensibility – in order to provide languages that are adequate to arbitrary application domains, but remain simple (in contrast, say, to "omnibus" languages like PL/I), "simply" facilitate the extension by programmers of simple base languages. Of Standish's [22] three classes of language extension:

- *orthophrase* – add new construct by modifying the implementation;
- *metaphrase* – change the meaning of existing construct by modifying the implementation;
- *paraphrase* – add new construct by declaration in the language being extended;

only the last of these qualifies as "practical", the others requiring language technology skills and indeed inviting all sorts of problems with language processor integrity and reliability. Moreover, even within paraphrase, it is conceivable for a programmer to become entangled with the complexities of writing/modifying language interpreters, as the universality of programming languages entails. This leads us to posit a distinction between two kinds of paraphrastic extension:

- *interpretational*: in which all the devices available in programming universal Turing machines are accessible;
- *definitional*: in which only direct definitions of required functions are permissible.

A theoretical basis for "definition" vs "interpretation" in language extension comes from the notion of "expressive completeness" [9]. Plotkins's PCF, which essentially adds parallel logical connectives to typed lambda-calculus, is expressively complete with respect to the standard Scott domain for computable functions in that all of these may be expressed by direct definition without recourse to interpretation.

2.3.2. Programming as language extension. Conversely, programming is a kind of language design and thereby a kind of language extension, as follows. First, 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.

Secondly, there is the correspondence in the respective approaches taken to them by their practitioners. Our key observation [10] 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.

Finally, a modified form of denotational semantics shows that declarations, the essence of any modular programming discipline, effect language extensions. Simply interchange the operands of the usual denotational meaning function M, i.e. changing the signature to "M :: Envt → Rep → Dom" where as usual 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 Envt = Id → Dom). Semantics of expressions are unchanged, save that operands of M are reordered from the usual, e.g.

$$\boxed{\mathrm{M}\ \rho\ [[\ \mathrm{I}\ ]] = \rho\ [[\ \mathrm{I}]]}$$

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

$$\boxed{\begin{array}{l}\mathrm{M}\ \rho\ [[\ \textbf{let}\ \mathrm{I} = \mathrm{E}\ ]] = \\ \quad \mathrm{M}\ (\lambda\ \mathrm{i}\ .\ \textbf{if}\ \mathrm{i} = [[\ \mathrm{I}\ ]]\ \textbf{then}\ \mathrm{M}\ \rho\ [[\ \mathrm{E}\ ]]\ \textbf{else}\ \rho\ [[\ \mathrm{I}\ ]])\end{array}}$$

and semantics for a program rewritten consistently:

$$\boxed{\mathrm{M}\ \rho\ [[\ \mathrm{D}\ ;\ \mathrm{E}\ ]] = \mathrm{M}\ \rho\ [[\ \mathrm{D}\ ]]\ [[\ \mathrm{E}\ ]]}$$

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

$$\boxed{\begin{array}{l}\mathrm{M}\ \rho = \\ \quad \mathrm{MM} \\ \quad \textbf{where} \\ \quad \mathrm{MM}\ [[\ \mathrm{D}\ ;\ \mathrm{E}\ ]] = \mathrm{MM}\ [[\ \mathrm{D}\ ]]\ [[\ \mathrm{E}\ ]] \\ \quad \mathrm{MM}\ [[\ \textbf{let}\ \mathrm{I} = \mathrm{E}\ ]] = \\ \qquad \mathrm{M}\ (\lambda\ \mathrm{i}\ .\textbf{if}\ \mathrm{i} = [[\ \mathrm{I}\ ]]\ \textbf{then}\ \mathrm{MM}\ [[\ \mathrm{E}\ ]]\ \ \textbf{else}\ \rho\ [[\ \mathrm{I}\ ]]\ )\end{array}}$$

In a real sense, MM (or "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, "MM [[D]]" defines MM extended by D. That is, declaration D truly extends language MM.

2.3.3. "Definition" vs "interpretation" in language extension and programming. If programming is a kind of language extension, then programming should be subject to the constraints of language extension. We have already seen that "language extension" should eschew interpretation, but how does interpretation manifest itself in a "programming" context? Consider for example the following standard definitions of elementary arithmetic operations: given successor "Succ" and zero "Zero", we have

```
add m Zero = m
add m (Succ n) = Succ (add m n)

mul m Zero = Zero
mul m (Succ n) = add m (mul m n)

exp m Zero = Succ Zero
exp m (Succ n) = mul m (exp m n)
```

In each case, we see patterns typical of interpreters:

- recursion on the structure of data (just as a programming language interpreter may recurse over an abstract syntax tree);
- branching on the symbolic representation of data (in this case the symbols "Zero" and "Succ").

It is just as if the symbols "Zero" and "Succ" formed a miniature language, the semantics of which (as natural numbers) were realised by the interpretation performed within operations "add", "mul", "exp", etc. In this simple example, the depth of problem with interpretation in language extension in general may be hard to discern, but we can perceive instances of more general problems:

- repeated unrelated occurrences of interpretive patterns increase programmer effort and the possibility of error;
- the semantics of the "natural number" mini-language has to be interpreted from the symbols each time a term in the language is used.

Because this interpretational treatment of naturals is typical of the treatment of an approach to structured data which pervades programming, interpretation and its problems pervade programming. But, as a kind of language extension, programming should be freed from the need for interpretation!

3. TFP essentials – "platonic combinators"

The general conclusions we draw from the above are:

- that for every data type, there is a characteristic behaviour pattern that embodies the essential semantics of the type (e.g. iteration for naturals);
- and, that rather than representing elements of data types as symbols that require interpretation in order to animate these behaviours, instead represent data as the functions that possess these characteristic behaviours

In other words, the essence of each member of a data type should be represented by a function, and the type recast into an appropriate function type. We call these characteristic functions "Platonic Combinators" (PCs), and within the class of PCs, there seem to be two distinct subclasses as follows.

3.1. Pure "platonic combinators"

The motivation for platonic combinators is to avoid interpretation. PCs that do so entirely we call "Pure Platonic Combinators" (PPCs), for example the data types/structures considered in terms of "folds" above (e.g. numbers, lists, trees). To reiterate the key points:

- the functions – PPCs - that embody the characteristic behaviours of data are formed by partial applications of the inverted form of the fold functions (for the relevant datatypes) to the data;
- these partial applications (of inverted folds to data formed by application of data constructors) can be fused to yield "generators" that directly create PPCs without involving "folds" explicitly. For example, we can give PPC (generators) for the Boolean and Cons-pair types directly: from

```
data Bool = True | False

data Pair t1 t2 = Cons t1 t2
```

we synthesise

```
true x y = x
false x y = y

cons a b c = c a b
```

The essential behaviours of (various types of) data met so far can thus be summarised:

- natural numbers are iterators
- booleans are selectors from two alternatives (generally N-element enumerations are selectors from N alternatives);
- ordered pairs are functionals that combine their elements according to their operand (which generalises to ordered N-tuples);
- lists are functionals that combine their elements under the influence of a binary operator with a nominated base value (and generally for other structures such as trees etc.)

TFP with the appropriately-characterised PPCs seems simple compared to interpretive functional programming. Consider for example:

- *arithmetic operations*

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

- *logical operations*

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

- *list operations (in addition to the earlier)*

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

Because the various "data" are represented by their characteristic behaviours, there is no need for the various operations on them to animate these behaviours from inert symbols. Compare especially the arithmetic operations above to their interpretational implementations earlier on, but similar benefits apply to the other types, more so with the complexity of each type.

3.2. Impure "platonic combinators"

In some cases, it's not possible or at least not convenient to remove interpretation of symbolic data entirely. However, we make progress to remove what interpretation we can, yielding interpretational/definitional hybrids that we call "Impure Platonic Combinators" (IPCs). By way of illustration, let us consider sets with the sufficient following simple signature:

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

The membership operation is specified:

```
member Empty x = false
member (Singleton e) x = x==e
member (Union s1 s2) x = member s1 x or member s2 x
```

A feasible representation for such sets is as characteristic predicates, rather than the "tagged" symbolic representation usually associated with signatures as above, so that:

```
member s x = s x
```

and by which predicates may be generated

```
empty x = false
singleton e x = x==e
union s1 s2 x =member s1 x or member s2 x
```

Obviously:

- characteristic predicates are platonic combinators, just as iterators were for naturals, in that they embody the essential behaviour of the type; in the case of sets, membership testing;
- "empty", "singleton" and "union" are PC generators, just as "zero" and "succ" were for naturals.

However, set membership testing necessarily involves comparison of actual to putative set member values, isolated here in "singleton". Value equality is inherently based on symbolic comparison (function equality is not generally computable), so this aspect of interpretation is inextricable from the concept of "set". However, besides elements, the other, structural aspects of sets are represented definitionally, i.e. in terms of the behaviour of functions.

Generators for IPCs can't be derived from type-signatures as simply as for PPCs via fold; however fold-theory can be employed for derivation of IPCs [11]. The examples of combinator parsers and exact real arithmetic considered above are in fact examples of IPCs.

4. Theoretical bases for TFP

The first class of obstacles to TFP concerns the well-founded-ness of the very idea:

- is the motivating distinction between "definition" and "interpretation" an objective one?
- then, what is an appropriate formal semantic domain for an interpretation-free TFP-subset of FP?

4.1. Formalising "interpretation"

The centrality to TFP of avoiding the interpretation of symbolic data begs the question of what actually comprises "interpretation". The fact that at least one apparently data-less system – the untyped lambda-calculus [12] - is not just theoretically viable but also universal, means that simply abolishing data is insufficient to prevent the writing of interpreters. In other words, it must be possible to recreate the substance of symbolic data and interpretation thereof from a system consisting purely of functions. For example, there are simple lambda-terms that treat functions exactly as if they were symbolic data. The following test for zero applicable to Church numerals, which gives an interpretation of numbers in terms other than of their inherent applicative behaviours, is arguably one of the simplest, but its very simplicity indicates that interpretation is (dangerously) easy to synthesise:

```
isZ n = n ( \ x . True) False
```

such that

```
        isZ zero
        = zero ( \ x . True) False
        = False
```

and

```
        isZ (succ n)
        = succ n ( \ x . True) False
        = ( \ x . True) (n ( \ x . True) False)
        = True
```

Without an objective characterisation of "interpretation" that includes such simple examples, and so allows us to exclude them from TFP, what we mean by TFP is ultimately ill-founded. Accordingly, we seek a syntactic characterization of definition vs. interpretation in order to define implicitly a conformant (necessarily subrecursive) functional language subset.

Because the difference between "definition" and "interpretation", as informally characterized above at least, seems to depend upon how usages of functions exploit their applicative behaviours (or not), we are currently exploring the concept of "residuals" in lambda-calculus theory. Residuals embody the idea of how the operand of an operator leaves some kind of image (or residue, hence "residual") in the application's result. Hence the idea occurs that a definitional usage of a function (as operand to some operator), which exercises the function's applicative behaviour, will be reflected by some part of the function appearing in the application's residual. Conversely, an interpretational usage, in which the behaviour of the operand-function is absent in the result, should leave no residual derived from the operand.

4.2. Semantic modelling of TFP

The "Scott" domain provides an elegant explanation for typed lambda-calculus, and as noted above illuminates the construction of an adequate ("expressively complete") basis for purely definitional programming of all of its elements. However, as we have seen just above, TFP must be less expressive, even subrecursive, in order to exclude the writing/simulation of interpreters.

A domain construction for the syntactically-characterised FP subset that comprises TFP would conceivably have the following benefits:

- a semantics that elegantly matched TFP, in the same way that the Scott domain matches Plotkin's PCF (so that the latter is expressively complete with respect to the former), would serve to validate any syntactic characterization of interpretation vs definition
- it would additionally provide 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 extant domain constructions alternate to the usual "Scott" domain and its derivatives [13], but which if any to use even as a basis for development remains to be researched. At the same time, it's worth noting something of a contradiction in our conception of TFP: "definitional" extensibility requires an expressively-complete base language, but to exclude interpretation TFP needs to be subrecursive. Thus, the search for a new domain against which the TFP subset of FP is expressively complete seems obligatory.

5. Infrastructure for TFP

The second class of obstacles to TFP concerns the technology that enables different kinds of efficiency: in software development; and of program execution.

5.1. Type-checking for TFP

Experience with even some elementary TFP examples indicates that more powerful type-checking than usual in FP is required. For example,

```
exp m n = n (mul m) (succ zero)
```

fails to pass the Hindley-Milner implicit polymorphic type-checker [14] that is the *de facto* standard in modern functional languages.

Reynolds' second-order polymorphic types [15] admit a much-wider class of functions than Hindley-Milner, 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 undecideable, but that for important subclasses (e.g. "rank 2") this is not the case [16]. Even if "rank 2" is suitable for TFP, a better implementation than is currently available [17] is indicated, that avoids unnecessary encoding of functions as data structures!

That is, the suitability to TFP of "rank 2" 2nd-order polymorphic types needs to be assessed, and a more integrated implementation produced.

5.2. TFP implementation optimisation

Despite advances in functional language implementation [6], it's to be expected that (just as in the case of type-checking above) at least some of the extreme combinations of higher-order function usage will exceed the anticipations of implementers. In other words, we can expect TFP to be inefficient.

Note by the way that other, naïve expectations of the inefficiency of TFP may not be realised. For example, "it's obvious" that the unary representation of natural numbers implicit in Church numerals is inefficient, but not necessarily so:

- if, as we hypothesise, proper usages of natural numbers necessarily involve iteration, then all of these will reduce naturals N to N-fold iterations as per the Church representations;
- that unary representations are space-linear rather than logarithmic (in alternative – positional – representations) could be overcome by compression;
- indeed, representations as Church numerals could be more time-efficient, avoiding the computation steps needed to animate the iterations from the symbolic representations!

Nevertheless, the risk of inefficiency from pathological combinations of higher-order functions remains very high. We remain hopeful that the identification of a TFP-subset of full FP will be of use here, as a TFP-tuned implementation would not have to cater for all of FP. It is reasonable to expect that this subsetting will admit some specific optimisations, but we are as yet in no position to estimate their extent, e.g. in terms of significant TFP applications that would be enhanced thereby.

6. Practical applicability of TFP

A further class of obstacles to TFP concerns the demonstrability of the practical applicability of TFP.

6.1. How much of FP is actually TFP?

We've cited above some "sightings" of non-trivial examples of the essential idea of TFP – replacement of symbolic data with functional representations – within FP, in domains as diverse as numerical analysis and language processing. However, these admittedly represent a limited base from which to extrapolate such a radical departure as TFP proposes:

- as motivations for TFP, they are too few to compel;
- generally, they fail to exemplify adequately the problems canvassed in the abstract throughout this paper;
- in particular, more guidance is needed in how to construct TFP-style solutions for a wider range of practical problems. To date, it's too easy to conceive of practical issues for which the responses of TFP can at best be described as "limited" (see also "What is TFP good for?" below), rather than "exemplary".

The benefits from discovering significantly more examples of TFP within FP would of course be the converses of the above. Possible avenues for this line of investigation might include:

- generally, inspecting (i.e. reading) as many significant functional programs as possible in order to discern any TFP patterns (fortunately, the canard that FP is of limited practical significance is now well-confounded by the evidence to the contrary [18]);
- specifically, we could concentrate on discovering usages of other FP "structuring paradigms" besides "fold", and translating them into fold-equivalents would yield platonic combinators as the basis of a TFP formulation.

6.2. Interpretation and hybrid TFP

There are some simple applications that at least appear to defy a purely TFP approach. Non-negative integers are plausibly iterators as Church numerals, but what is the characteristic function of a negative integer? Perhaps an iterator of the "predecessor" function on integers, but this raises the further problem that "predecessor" seems to require the use of simulated interpretation (see 'Formalising "interpretation"' above) in its definition.

Developing the above theme further, there would appear to be some applications that are inherently interpretational, for which TFP can never provide solutions. For example, data modelling and processing of human information will inevitably involve identification by symbolic data representing names (or equivalents), and making decisions based on that symbolic data.

A possible response is to suggest that modelling of human information by name is fundamentally wrong, and that human data should be represented by their characteristic functions. Maybe so, but it's far from clear what these would be, and in any case would seem to require such a profound development, not just in software technology but also in human society, as to indicate that early access to the benefits of TFP should be provided by alternate means if possible.

A more realistic response rather is to concede that a hybrid approach to TFP is also potentially beneficial. When a complete problem is not entirely amenable to the complete elimination of interpretation, perhaps parts of the solution can be "TFP-ised", i.e. expressed in definitional terms. For example, a problem involving natural numbers and other types could have the naturals represented as Church numerals, and the other types left as symbols. We've clearly demonstrated that definitional TFP admits hybrid co-existence with interpretation, at least in principle through IPCs, and the earlier-cited cases of functional representations of data (combinator parsers for grammars, characteristic predicate representations of sets, etc.) support the principle by practical example. We suspect that many further significant successful examples of co-habitation remain to be demonstrated.

6.3. TFP beyond "programming"

TFP appears to have applications beyond mere software development, some of which are identified here.

6.3.1. Canonical software design representation. One of the detriments of interpretational language extension/programming is that it allows the characteristics of a software system to be disguised in varying degrees in the data, while the program is correspondingly more or less generic. This poses dual problems in a software reverse engineering/design recovery context [19], where (i) the data-disguised design needs to be uncovered, and (ii) it's essential that the recovered design itself retain no data-disguised design information. It would appear that TFP, which by eschewing interpretation minimises the possibility for such disguises, is an ideal candidate for a canonical representation for software designs, in reverse- as well as forward-engineering.

6.3.2. Analog design. There appears to be an interesting connection between analog computing, where computations are composed from physical components whose behaviours model the domains being computed with, and TFP where data are represented by (the behaviours of) platonic combinators. Indeed, the existence of TFP suggests that the hitherto one-dimensional division of computation into the poles Analog vs. Digital should be replaced by a two-dimensional structure: (Interpretational vs Definitional, Discrete/Symbolic vs Continuous). Conventional "digital" computing is identified by the (Interpretational, Discrete/Symbolic) point, analog by (Definitional, Continuous), and TFP by (Definitional, Discrete/Symbolic), as in the following diagram:

	Discrete/Symbolic	*Continuous*
Interpretational	*Digital*	*???*
Definitional	*TFP*	*Analog*

TFP would seem to have potential as a design/specification/prototyping language for analog systems: functionality can be developed in the relatively relaxed Discrete/Symbolic domain before being built in the exacting electronic manifestation of the Continuous domain.

6.3.3. Systems engineering. The TFP-analog computing connection seems capable of further generalisation to any system composed in terms of the behaviours of its components. The tools and techniques envisaged above for analog design could therefore be applicable to systems engineering in general.

7. TFP vs OOP

The final class of obstacles to TFP concerns whether or not other beneficial programming styles can be combined with it, or if TFP requires the radical reformulation of much or even all of the science of software development to date. In addition to the issue considered above (if interpretation can't be avoided, how can it be handled in TFP?), we now consider the question of to what extent is arguably *the* prevailing software development paradigm – object-orientation – compatible with TFP?

Obviously, PCs correspond to objects (and the function types of PCs correspond to classes). The problem to be overcome is that such TFP objects appear to have serious limitations on the methods definable on them (i.e. one method per class/object). However, the means by which this limitation may be overcome has potential ramifications for the concept of cohesiveness of classes and their methods for OOP in general.

7.1. Methods and PCs

PC generators obviously correspond to constructors, but what about PC versions of other methods (selectors/extractors/etc.)? The answer is that the PC itself is its single selector/extractor, the "characteristic method" of the object it represents:

- for naturals, there is a single iteration method, embodied in the PC representation;
- for sets, the single method is membership testing (our term "characteristic method" in fact derives from generalising the characteristic predicate that applies in this case);
- for grammars, the method is the parser, i.e. a function which when applied to some input builds the parse tree of its input according to the grammar.

For example, the Hutton-style combinator parser for the grammar:

```
Sentence -> Subject Predicate
Subject -> “Paul” | “Colin”
Predicate -> “eats” | “sleeps” | “works”
```

can be rendered by (Haskell) definitions

```
sentence = subject `cat’ predicate
subject = tok “paul” `alt` tok “colin”
predicate =
tok “eats” `alt` tok “sleeps” `alt` tok “works”
```

where the combinator parser generators "cat", "alt" and "tok" are defined (note infixing of "cat" and "alt"):

```
(p1 `alt` p2) xs = p1 xs ++ p2 xs

(p1 `cat` p2) xs =
foldr (++) [ ] (map (\ (s1,s2) . cross Fork (p1 s1) (p2 s2))
(split xs))

split xs = take (length xs ) (split' xs)
split' [ ] = [([ ], [ ])]
split' (x:xs) =
    map (appfst (x:)) (split' xs) ++ [([], x:xs)]
    where
    appfst f (x,y) = (f x, y)

cross f  xs [ ] = [ ]
cross f [ ] ys = [ ]
cross f  (x:xs) ys = map (f  x) ys  ++  cross f  xs ys

tok :: String -> Parser a
tok ts  xs = if (ts == xs) then [Leaf ts] else []
```

The "parse tree" has internal nodes: "Fork" for the binary branch for a "Sentence"; and "Leaf" for specific words (these could be made generic if required):

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

The parser is nondeterministic in that all possible parses of the input are generated, as a list of parse trees – this is why "alt" generates a composite parser from parsers "p1" and "p2" which simply concatenates the results of parsing "xs" according to "p1" and "p2". For "cat", arrangements need to be somewhat more complex: the role of auxiliary functions "split", "cross" etc. is to consider all divisions of the string "xs", parse them with "p1" and "p2", and the combine the trees resulting from successful parses with "Fork". "tok" simply matches its input with the given token string.

The problem is that this "single selector" rubric inherent to PCs seems to be at odds with practical requirements. Consider for example this class of context-free grammars. We've identified combinator parsers as their PC version, i.e. parsing as the single method on grammars. However, grammars reasonably have other methods defined, e.g. recognition (instead of generating a parse tree, return true/false depending upon whether the string is generated by the grammar); and unparsing (given a parse tree, produce the input string that would have been parsed into the tree). How can the reasonableness of having multiple methods on classes/objects be reconciled with PC-as-single-method dogma of TFP?

7.2. Generalised "characteristic methods"

Our solution exploits functional languages' adherence to Tennent's principles of abstraction and correspondence [20]. In particular, any component can be parameterised in terms of any of it sub-components. This means in particular that multiple methods on a class/object can be instantiated from a single "mother" characteristic method by suitable selection of formal and actual parameters, and that it is this "mother" that is the single characteristic method of the class.

For grammars, let us consider just one of the legitimate alternate methods besides parsing: simple "yes/no" recognition of an input string. From an implementation standpoint, a recogniser differs from a parser essentially in that the means of combining the results of sub-parsers only needs to handle the Boolean result of recognition:

```
(p1  `alt` p2) xs = p1 xs || p2 xs

(p1 `cat` p2) xs =
foldr (||) False (map (\ (s1,s2) . p1 s1 && p2 s2) (split xs))

tok ts xs = ts == xs
```

Thus, whereas with parsing the results of alternated parsers are concatenated (to simulate union of sets of nondeterministic results), with recognition the Boolean results are disjoined (i.e. overall success if at least one recognition succeeds). Analogous changes apply for concatenation of recognisers.

We can now move to posit the characteristic method of the class of context-free grammars as the common "mother" of parsing and recognising, by abstracting the above differences from the definitions of the generators:

```
 (p1  `alt` p2) tokf join base combine xs =
   p1 tokf join base combine xs
   `join`
   p2 tokf join base combine xs

(p1 `cat` p2) tokf join base combine xs =
   foldr join base (map f (split xs))
   where
   f (s1,s2) =
      combine
      (p1 tokf join base combine s1)
      (p2 tokf join base combine s2)

tok ts tokf join base combine  xs = tokf ts xs
```

- "tokf" describes the result of processing a token
- "join" describes how to unite the results of alternate grammars
- "base" is a default result for concatenated grammars, when no options matches
- "combine" describes how to connect the results of concatenated grammars

For a grammar 'g' generated with these generic "alt", "cat" and "tok", we can instantiate a recogniser:

```
g (\ts xs -> ts == xs) (||) False (&&)
```

and a likewise a parser:

```
g
(\ts xs -> if (ts == xs) then [Leaf ts] else [ ])
(++)
[ ]
(cross Fork)
```

7.3. Characteristic methods and cohesion?

It may be argued that the approach taken above, i.e. identification of a characteristic method as the common "mother" abstraction to the various apparently distinct methods, is a vacuous exercise in that it's always possible to find some mother for any set of methods. For example, in the limiting case arbitrarily-distinct methods can always be regarded as children of the identity function as their mother: for methods Mi:

```
(\x.x) Mi = Mi
```

This is indeed so, but rather than invalidating our approach, it suggests that the approach has wider applicability as a measure of the cohesion of multi-method classes, as follows. "Cohesion" is a valued attribute of software components, classes and objects not excepted. Measures of cohesion as developed to date (e.g. [21]) can be regarded as somewhat "syntactic" in that they don't depend on knowledge of the functions of various methods. We're optimistic that "mother" classes could serve as the basis of a more semantic approach to measuring cohesion, in that they at least embody abstractions of the behaviours of their instances/descendants, as follows. The arbitrarily-distinct methods Mi that would require the identity function as their "mother" should be regarded as not cohesive. On the other hand, methods that admitted a "mother" that embodied substantial common structure, with relatively minor differences expressible in terms of a few parameters, would be regarded as semantically cohesive. For example, the above treatment of parsers, recognisers and generic grammars suggests that parsing and recognition are cohesive.

We envisage that it should be possible to establish some kind of (partial) ordering of "mother" characteristic methods, in terms of their specificity. The "identity" function would be least specific, functions without parameters being most specific. The less specific that the "mother" required to cover a set of methods would be, then the less cohesive would the methods be deemed.

8. Conclusions

TFP seems to possess enough objective foundations in functional programming theory and practice to be worth further exploration.

- *Theoretical foundations*: discover a domain against which an FP-subset, that precludes simulation of interpretation, is expressively-complete.
- *Infrastructure*: exploit the fact that TFP excludes some FP constructions to improve implementation efficiency; maybe type-checking will benefit similarly?
- *Practical application*: as well as developing new applications TFP-style, continue to search for TFP patterns in existing FP applications.
- *Relationship to OOP*: elaborate the idea that the methods of a cohesive class can be instantiated from a common "mother" method; doubtless draw upon the previous item for examples.

9. Acknowledgements

We are grateful to Ian Peake and Sean Seefried for their work on earlier developments in TFP, and for the anonymous referees for detailed comments.

10. References

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

[2] Bird, R., *Introduction to Functional Programming*, Prentice-Hall, 2000.

[3] J. Hughes, "Why Functional Programming Matters", *Comput. J., vol. 32, no. 2*, 1989, pp. 98-107.

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

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

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

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

[8] J. Gibbons, G. Hutton and T. Altenkirch, "When is a function a fold or an unfold?", *Coalgebraic Methods in Computer Science*, *Electronic Notes in Theoretical Computer Science, vol. 44.1*, 2001.

[9] G.D. Plotkin, "PCF Considered as a Programming Language", *Theoretical Computer Science, 5*, 1977, pp. 223-255.

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

[11] P.A. Bailes and C.J.M. Kemp, "Formal Methods within a Totally Functional Approach to Programming", in B.K. Aichernig and T. Maibaum (eds.), *Formal Methods at the Crossroads: from Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, The International Institute for Software Technology of The United Nations University*, Springer, 2003, pp. 287-307.

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

[13] Schmidt, D.A., *Denotational Semantics*, Allyn and Bacon, 1986.

[14] R. Milner, "A Theory of Type Polymorphism in Programming", *J. Comp. Syst. Scs., vol. 17*, 1977, pp. 348-375.

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

[16] J.B. Wells, "Typability and Type Checking in the Second-Order Lambda-Calculus are Equivalent and Undecideable", *Logic in Computer Science*, 1994 pp. 176-185.

[17] M.P. Jones, "First-class Polymorphism with Type Inference", *Proc. Symposium on Principles of Programming Languages*, 1997.

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

[19] E. Chikofsky, E. and J.H. Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software*, January, 1990, pp. 13-17.

[20] R.D. Tennent, "Language Design Methods Based on Semantic Principles", *Acta Informatica, vol. 8*, 1997, pp. 97-112.

[21] S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object Oriented Design", *Proceedings of OOPSLA'91, ACM SIGPLAN Notices, vol. 26, no. 11*, November, 1991.

[22] T. Standish, Extensibility in Programming Language Design, *Proc. Natnl. Comp. Conf.*, 1975, pp. 287-290.