WHY FUNCTIONAL PROGRAMMING *REALLY* MATTERS

Paul A. Bailes
Colin J.M Kemp
Ian D. Peake
Sean Seefried
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072
AUSTRALIA
{paul, ck, ian.peake, seefried}@itee.uq.edu.au

ABSTRACT

The significance of functional programming is revealed as that the feasible approach to language extensibility which it enables is further applicable to programming in general and beyond. The essence of functional programming is its enablement of programmer-defined function-valued functions. The feasibility of language extension by normal programmers depends upon the exclusion of interpretation in favour of direct definition, and higher-order functional programming is the key to enabling definitional rather than interpretational extensions. Functional programming thus offers the opportunity for the exclusion of the interpretation that otherwise pervades programming in general, and may be applicable beyond to analog computing and systems in general. Nevertheless, specific technical challenges need to be met before "totally functional programming" can realise its promises.

KEY WORDS

Programming Tools and Languages; Functional Programming; Language Extensibility; Software Design and Development

1. WHAT IS FUNCTIONAL PROGRAMMING REALLY ABOUT?

Functional programming [1] embraces numerous topics and themes, such as "applicative" programming, "data-flow", first-class data structures, programming with "combining forms", lazy evaluation and first-class functions. Because all of these effects can be derived from the last, it's reasonable to propose that the facility to define and process functions as first-class values, including function-valued functions, is the defining characteristic of functional programming.

In that context, the purpose of this paper is to explore the significance of higher-order-functions, not just as an "optional extra" besides the rest of functional programming, let alone programming in general, but as *the* basis for software development. In particular, we aim to replace as far as theoretically- or practically-possible, the use of inert symbolic data with animated, functional representations. In so doing, we explain the wider significance of our aversion to data, provide links back to mainstream functional programming, and indicate the potential of this extreme functional style beyond mere programming.

2. FUNCTIONAL PROGRAMMING ENABLES DEFINITIONAL EXTENSION

Functional programming enables feasible language extension by normal programmers, excluding the need for complex interpretation in favour of simple direct definition.

2.1. Language Extension as Programming

Language extension is desirably a normal "programming" activity. This is because extensibility is a response to the tension between basic language design criteria of expressiveness (tending toward large full-featured languages) vs simplicity (tending to smaller possibly special-purpose alternatives). If programmers can extend base languages to support preferred applications and methodologies, then both criteria can be met. While some extreme goals have remained unmet, extensibility is a practical programming task, e.g.:

- it's a long-standing practice for programming languages to define core infrastructural components (such as input-output) through libraries (so-called "standard preludes");
- increasingly expressive programming languages allow increasingly "deeper" infrastructure to be provided likewise (e.g. the Haskell library).

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 these, 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.

2.2. Feasible Extension: Definition (vs Interpretation)

The essence of our difficulty with non-paraphrastic extensions is that they involve interpretation. Accordingly, we must further distinguish 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. Because indirect paraphrase involves programming of an interpreter to the same degree as orthophrase or metaphrase, it is similarly rejected.

That is, the effective principle of feasible language extension is that a new operator should be represented by direct paraphrastic definition (hence "Definitional") rather than by data that needs somehow to be interpreted through modification or recreation of an interpreter (hence "Interpretational").

2.3. Functional Programming Enables Definitional Extension

The basic significance of higher-order functional programming is thereby revealed in that it enables definitional rather than interpretational extensions, as follows.

One approach to formalizing the notion of "direct definition" in paraphrase is in terms of the expressiveness requirement on a supporting host language: the host needs to be able to express directly all conceivable extensions. Technically, this requirement that a host language directly express all elements in its semantic domain is to require that it be expressively complete [3]. Expressive completeness thus means that all entities in a language's semantic domain are directly expressible (without recourse to writing an interpreter) by terms in the language, and importantly is not the same as Turing-completeness.

For example, a first-order language such as Pascal is just as Turing-complete as an higher-order language such as Haskell [4], but the latter is in some sense more extensible/expressive (formally: more expressively complete). Whereas Haskell can directly represent and operate on non-cofinite sets (as characteristic predicates):

```
empty elt = False
singleton x elt = x==elt
union s1 s2 elt = s1 elt or s2 elt
```

Pascal can't, because set-theoretic operations on the predicates – function-valued-functions – can't be directly expressed in Pascal.

A summary of distinctive technical features of an expressively-complete version of a typical modern programming language is

1. programmer-definable higher-order functions over typed basis including at least booleans
2. non-strict evaluation
3. symmetric (or "parallel") implementations of basic logical connectives.

While 3. is regrettably rare, 1. (and 2.) prove that functional programming as characterized above is a necessary condition for expressive completeness, i.e. for feasible language extensibility by definition rather than interpretation.

3. PROGRAMMING AS LANGUAGE EXTENSION

Just as language extension is a kind of programming, so programming is a kind of language extension, and programming practice is subject to analysis from a language design/extension perspective.

3.1. "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.

3.1.1. 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 ?).

3.1.2. 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.

3.1.3. 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 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.

3.2. "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 [5] 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.

3.2.1. 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 therefsore transparently an adequacy-enhancing language extension exercise.

3.2.2. 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.

3.2.3. 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.

3.2.4. Readibility 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.

3.3. Basis in denotational semantics

It's demonstrable via denotational semantics that declarations, the essence of any modular programming discipline, effect language extensions. First, consider the typical denotational meaning function M with signature "M :: Rep → Envt → 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 Envt = Id → Dom. For example, the semantics of an identifier occurrence I is determined by accessing the environment ρ:

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

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:

$$M\ [\![\textbf{let}\ I = E]\!]\ \rho = (\lambda\, i\ .\ \textbf{if}\ i = [\![I]\!]\ \textbf{then}\ M\ [\![E]\!]\ \rho\ \textbf{else}\ \rho\ [\![I]\!])$$

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

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

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

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

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

$$M\ \rho\ [\![\textbf{let}\ I = E]\!] = M\ (\lambda\, i\ .\ \textbf{if}\ i = [\![I]\!]\ \textbf{then}\ M\ \rho\ [\![E]\!]\ \textbf{else}\ \rho\ [\![I]\!])$$

and semantics for a program rewritten consistently:

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

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

$$M\ \rho = MM\ \textbf{where}\ MM\ [\![D\ ;\ E]\!] = MM\ [\![D]\!]\ [\![E]\!]\ MM\ [\![\textbf{let}\ I = E]\!] = M\ (\lambda\, i\ .\ \textbf{if}\ i = [\![I]\!]\ \textbf{then}\ MM\ [\![E]\!]\ \textbf{else}\ \rho\ [\![I]\!])$$

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.

3.4. Conclusion

Just as language extension is a kind of programming, we have now assembled the argument for the converse:

1. the pragmatic correspondences between programs and languages and between programming and language design (which extends implicitly to language extension as a language design process) show that programming and language extension are conceptually identical;
2. it should follow that language extension is a kind of programming, the independent establishment of which above serves to corroborates 1.;
3. from 1. thus corroborated, the converse should follow i.e. that programming is a kind of language extension;
4. independent corroboration of 3. derives from our demonstration of a formal basis for treating programming as a kind of language extension.

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 language extension.

4. FUNCTIONAL PROGRAMMING ENABLES DEFINITIONAL PROGRAMMING

Programming is a kind of language extension, and thereby inherits any constraints/requirements/conditions applicable to language extension. Recall that the conditions on language extension, derived from its perception as a kind of programming, are that it be prosecuted definitionally not interpretationally. These same conditions thus now reflect back onto programming, i.e. programming should eschew interpretation in favour of direct definition. The further significance of functional programming is thus revealed in terms of the opportunity it offers to pursue programming subject to these language extension conditions, i.e., definitionally not interpretationally.

4.1. Pervasiveness of Interpretation

To date however interpretation is rife throughout programming. Consider the following simple datatypes and operations thereon:

```
data Bool = True | False
not True = False
not False = True
and True x = x
and False x = False

data Nat = Zero | Succ Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
mul Zero n = Zero
mul (Succ m) n = add m (mul m n)
```

Observe how the essential structure of an interpreter (as emerges from the operation definitions:

- different behaviours are achieved by branching on the symbolic value of operands (just like how an interpreter branches on different operation codes);
- nested structures are processed recursively;
- for each type there are actually multiple interpreters acting in concert to provide consistent semantics for the "micro-languages" that each data type comprises: "not", "and" for "Bool"; "add", "mul" for "Nat".

This last point is brought out most clearly if we define single interpreters for each type and then define the operations in terms thereof. Thus:

```
if_then_else True x y = x
if_then_else False x y = y
...
not x = if_then_else x False True
and x y = if_then_else x y False

iter Zero f x = x
iter (Succ m) f x = f (iter m f x)
...
add m n = iter m Succ n
mul m n = iter m (add n) Zero
```

Salient points:

- note how the characteristics of interpretation (branching, recursion/looping) are absent from the otherwise direct definitions of "not", "and", "add", "mul"; rather these characteristics have been localised in the respective interpreters "if_then_else" and "iter";
- each interpreter embodies a proposed universal or objective behaviour for its respective type;
- the proposed objective behaviour for booleans is branching; and that for naturals is iteration;
- other types can be expected to have their own such behaviours and interpreters.

To summarise:

- operations on symbolic data take the essential structure of interpreters, thus symbolic datatypes can be thought of as "micro-languages" interpreted by their operations
- alternatively, the interpreter for each micro-language can be consolidated into a single interpreter ("universal/objective behaviour") in terms of which the other operations can be non-interpretively defined.

This process of having to define interpreters when extend-

ing languages with new types and operations is exactly the kind of undesirable extension we have committed to avoid.

4.2. Introducing "Platonic Combinators"

As with the difference between interpretational and definitional language extensions, functional programming admits the complete replacement symbolic data in favour of totally functional representations, as follows.

At the worst, the pervasiveness of interpretation means that each operation on a datatype is obliged to include an interpreter for the symbolic data. In mitigation, it is conceivable to supply for the datatype a single interpreter/"universal behaviour" in which the operations may be defined directly, thus quarantining them from the need for including interpretation. Why not proceed to the conclusion, and represent symbolic data by their interpreters partially applied to them. These "pre-animated" data can then be applied without further interpretation. We propose to call these "platonic combinators" to convey the implicit hypothesis that underlying every datatype there is an essential applicative nature, i.e. the objective/universal behaviours as described above.

Platonic combinators are thus formed by the partial application of interpreters to data, e.g.

```
true = if_then_else True
false = if_then_else False

zero = iter Zero
one = iter (Succ Zero)
etc.
```

and are used without further citation of the interpreter e.g.

```
and x y = x y false

add m n = m succ n
```

The last above cites "succ" which is the counterpart to constructor "Succ" but which applies to platonic combinators, and in general begs the question of direct generation of platonic combinators rather than by partial application of interpreter to data. Fortunately, the partial applications can be transformed straightforwardly, e.g.

```
true    = if_then_else True     = \ x y → x
false   = if_then_else False    = \ x y → x

zero    = iter Zero             = \ f x → x
succ n  = iter (Succ n)         = \f x → f (n f x)
```

(Note we follow Haskell concrete syntax here; in particular "\ args → body" equates to "(λ args . body)")

Thus e.g. for naturals N, there is an infinity of platonic combinators of the form "(λ f, x . f^N x)" for each natural N (the "Church numerals"), but a finite set of generators:

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

4.3. Pure vs. Impure Platonic Combinators

There is no reason to suppose platonic combinators are restricted to such simple examples. For example, platonic combinators exist for data *structures*. The platonic purpose of a list is to order its elements X1 … Xn for processing by some operation "op". Accordingly, such a list-takes the (platonic) form

\ op b → op X1 (op X2 (...(op Xn b)...))

where 'b' is the value for the empty case. Platonic lists have generators "nil" and "cons" accordingly:

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

Operations on platonic lists may be programmed definitionally rather than interpretationally, e.g.

```
total xs = xs (+) 0
length xs = xs (\ x n → n+1) 0
concat xs ys  = xs (:) ys
```

Structures such as lists can be regarded as definitionally "pure" platonic combinators (PPCs), in that there is nothing inherent to the structure that compels interpretation. However, some structures are not so pure, for example sets. It will be realized that the version of sets as defined above, in terms of characteristic predicates, attempt to embody the "definitional" ideal, in that the structure is animated, i.e. represented in terms of a function that embodies its essential objective behaviour (membership testing). However, the equality test on putative elements in "singleton" necessarily involves ultimate interpretation, i.e. symbolic comparison of data in the predicate for "single"(ton).

Such "impure platonic combinators" (IPCs) thus provide a potential bridge between our somewhat rigorous dataless ideal, and a scenario where the totally functional programming style can be used in hybrid mode, if not to expunge entirely the evils of interpretation, at least to mitigate them in practical settings.

4.4. Practicability of Platonic Combinators

Indeed, IPCs can be discerned widely among some of the more interesting applications of higher-order functional programming. As well as indicating the pragmatic utility of IPCs, these examples lead us to speculate that the excitement of many more functional programming applications may be attributable to their replacement of symbolic data by platonic combinators.

4.4.1. Combinator parsers: a good example of the replacement of interpretation of data structures by direct functional representation is provided by the unification of grammars and parsers in the "combinator" parser [6]. Rather than parsing by applying a "parsing engine" (interpreter) to a grammar or some derived representation thereof (e.g. parse table), instead the means by which

grammars are constructed actually construct the corresponding parser. Thus, context-free concatenation and alternation are denoted by higher-order functions that operate on parsers to produce the appropriate composite parsers.

4.4.2. Exact real arithmetic: Boehm & Cartwright [7] represent real numbers by functions which compute reals to any required rational precision.

4.4.3. Programmed graph reduction: rather than represent a function as a graph into which an interpreter substitutes operand, represent as the program which builds the resulting graph (or its functional representation) [8].

4.4.4. Subrecursion: because the definitional style eschews recursion, there would seem to be a link with the entire "subrecursive" programming field [9]. Significantly, Turner [10] has also identified a link, albeit different, between subrecursive and functional programming.

5. RELATIONSHIP TO FOLD OPERATORS

Definitional programming can also be thought of as exploiting a particular paradigm within functional programming – structuring programming around "fold" operators. Thus, the discovery of platonic combinators derives from the specification of the datatypes for which PPCs and IPCs provide alternative representations.

5.1. PPCs

For any regular recursive datatype, generators for PPCs can be derived basically as a simplification of the "fold" function for the type [11]. For example, consider the correspondence between "nil" and "cons" above on the one hand, and the equations for the usual functional "fold" on the other:

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

The differences may be reconciled by reordering operands and fusing the "fold" with the structure itself, leaving the fold operations as remaining arguments. Earlier PPC generators for booleans and natural numbers (Church numerals) are likewise simplifications of the relevant fold-variants:

```
foldB t f True = t
foldB t f False = f

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

(It's instructive to view the earlier-proposed "universal behaviours" for Booleans and naturals – "if_then_else" and "iter" – as the intermediate stage between "fold" and PPC generators.)

A highly-valued aspect of functional programming with folds is the encapsulation of proof rules, to match the packaging of recursion in folds. One of the most significant is "fusion", which is applicable in modified form to the PPC-variants of folds e.g. for platonic lists L:

L F1 F2 = H (L E1 E2)
provided that
H (E1 X Xs) = E1 X (H Xs)
H E2 = F2

and similarly for all other PPCs.

5.2. IPCs

In the impure case the situation is a little more complicated. Whereas PPCs/generators are characterised (like folds) by the type-signatures for the datatype constructors, IPCs need to account additionally for the specification of a selector, in which any interpretation will be contained. Fortunately, it's implicit in the hypothesis of a universal behaviour for a datatype that there be exactly one selector for each datatype. Derivation of IPC generators from the specification (constructors & selector) is perhaps clever but nonetheless straightforward. We use the earlier example of the IPC version of sets (represented by characteristic predicates) to illustrate by steps as follows.

1. The overriding objective is that an IPC is equivalent to the partial application of the distinguished selector to a member of the datatype. For the type specified:

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

   the IPCs corresponding to sets S are just the characteristic predicates definable by the partial applications "member S".

2. The IPC generators we seek are accordingly the "empty", "single" and "union" that construct characteristic predicates. For each different means of constructing a characteristic predicate by partial application of "member" above to a set data structure, we seek to construct the characteristic predicates directly by means of applying IPC generators:

```
empty = member Empty
single X = member (Sngl X)
union P1 P2 = member (Union S1 S2)
```

   Note that whereas Si are conventional data structures, Pi are corresponding IPCs (characteristic predicates).

3. The solution involves reconsideration of PPCs. As inclusion of the selector is what differentiates an IPC datatype from a PPC datatype, setting aside the selector allows an underlying PPC datatype to be discerned. For these sets the PPC generators would be "em"(pty), "sin"(gle) and "un"(ion):

```
em e s u = e
sin x e s u = s x
un s1 s2 e s u = u (s1 e s u) (s2 e s u)
```

The fusion law for the "set" PPCs is

S F1 F2 F3 = H (S E1 E2 E3)

provided that

H E1 = F1
H (E2 X) = F2 X
H (E3 S1 S2) = F3 (H S1) (H S2)

4. As with fusion, it also follows from the definitions of PPCs/generators (i.e. from "fold") that there is a correspondence between datatype constructors Ci, PPC generators Gi, and PPC operands Xi (this is apparent in the PPC generators for sets above). Moreover, if some PPC derives from application of Gi:

   PPC = Gi args …

   then

   PPC X1 … Xn = Xi args …

   In particular,

   PPC Gi … Gn = Gi args … = PPC
   PPC C1 … Cn = Ci args…

   That is, application of a PPC to the generators returns the PPC; and the interpretational data structure may be recovered from the definitional PPC by applying the PPC to the data structure constructors. For example:

```
sin X Empty Sngl Union = Sngl X

un P1 P2 Empty Sngl Union
= Union (P1 Empty Sngl Union) (P2 Empty Sngl Union)
= etc as "Pi Empty Sngl Union" yield data structures
```

   We don't suggest this as a desirable programming style, but it's a key ingredient this derivation.

5. To each side of the relationship (2. above) between IPC generators and IPCs, apply the above relationships (4.) – in particular, replace each IPC (left side) by the application of the corresponding PPC to the IPC generators, and replace each data structure instance (argument to "member"- right side) by the application of the corresponding PPC to the constructors. Thus:

```
em empty single union
        = member (em Empty Sngl Union)
sin X empty single union
        = member (sin X Empty Sngl Union)
un P1 P2 empty single union
        = member (un P1 P2 Empty Sngl Union)
```

   Because these rules cover all ways of generating set PPCs S, we may amalgamate them into a single rule:

```
S empty single union = member (S Empty Sngl Union)
```

   which is in the form to which fusion (3. above) is applicable. Thus it suffices that

```
member Empty = empty
member (Sngl X) = single X
member (Union S1 S2) = union (member S1) (member S2)
```

6. Combining these equations with the earlier definitions of "member" yields direct definitions for the IPC generators. The first two are obvious:

```
empty elt = False
single x elt = x==elt
```

   but the third requires a little elaboration. First, recognise that "member Si" refers to an IPC, i.e. operand acceptable to "union", i.e. the right hand side is

```
union s1 s2
```

   Returning to the left hand side,

```
member (Union S1 S2)
= (following "member")
\ elt → member S1 elt or member S2 elt
= (consolidating "member Si" to "si" as above)
\ elt → s1 elt or s2 elt
```

   Reuniting both sides yields the definition as required:

```
union s1 s2 elt = s1 elt or s2 elt
```

6. BEYOND FUNCTIONAL PROGRAMMING

The definitional functional programming style elaborated above seems to have significance beyond mere programming as follows.

6.1. Canonical Software Design Representation

One way of conceiving of the drawbacks of conventional, interpretational programming is that the more the behaviour of software is controlled by the data, the more generic and less informative does the interpreting code become. In the limiting case, the data becomes a program and the code a generic, fully-fledged but uninformative interpreter. This situation resembles the case in the software reverse engineering/design recovery context, where as well as the need to uncover the design hidden in data, it's essential that the recovered design itself retain no data-disguised design information. It would appear that our definitional programming style may serve as candidate for a canonical representation for software designs.

6.2. Analog and Systems design

Our situation, where data are represented in terms of objective behaviours, and the operations thereon exploit these behaviours, seems to recall analog computing, where computations are composed from physical components whose behaviours model the domains being computed with. Thus, rather than the hitherto one-dimensional division of computation into the poles Analog vs. Digital we suggest two dimensions: (Interpretational vs Definitional, Discrete/Symbolic vs Continuous). Conventional "digital" computing is identified by the (Interpretational, Discrete/Symbolic) point, analog by (Definitional, Continuous), and this definitional proposal by (Interpretational, Discrete/Symbolic), as in the following diagram:

	Discrete/Symbolic	*Continuous*
Interpretational	Digital	
Definitional	Definitional	Analog

Functional programming would seem to have potential as a design/specification/prototyping language for analog systems and perhaps systems engineering more generally, whenever systems are composed of components defined by their behaviours rather than their representations.

7. REALISING TOTALLY FUNCTIONAL PROGRAMMING

Unsurprisingly, numerous technical issues remain to be surmounted before this approach to software development is to be regarded as a compelling alternative to established styles, including "mainstream" functional programming.

7.1. Objectivity of TFP

Some of our hypotheses in particular demand validation:

- is there a characterisation of a total subset of functional programming adequate to practical requirements?
- does a wide range of interesting data types each have unique platonic combinator classes? For example, in the combinator parser case, a grammar would seem to have a pair of essential operations: "parse" and "unparse" – is there a common non-trivial combinator from which the two can be derived?

7.2. Type-theoretic issues

The Hindley-Milner [12] type checker typical to modern functional languages is not sufficient to handle some simple operations on platonic combinators. For example, the obvious definition of exponentiation on Church numerals

pow m n = n (mul m) (succ zero)

fails to type check. A more subtle definition

pow m n = n m

is acceptable, but to require such subtlety seems unacceptable. Clearly a more powerful type system is required. Second-order polymorphic typed-lambda-calculus [13] is much more expressive, and seems to be able to type the above "erroneous" application, but has the drawback of not enjoying the convenience of type inference, unlike Hindley-Milner

7.3. Implementation

Because definitional style seems to be associated with a subset of functional programming, it may be possible to use this subset as the key to optimisations.

8. CONCLUSIONS

Our idea is that the significance of functional programming/languages is that it provides the key to a language-extension-based view of software development and enables the according shift from the prevailing worldview. The old worldview involves the pervasive need for programmers to create symbolic representations of data and consequently to write interpreters. The new worldview avoids encoding and interpretation by the direct representation of data by the functions which embody their essential "Platonic" behaviours. The ability to create and use IPCs means that both worldviews can productively coexist, as further indicated by some apparently significant examples (combinator parsers etc.). A conceptual link with analog computing offers much wider potential application than just what may be regarded as "programming".

REFERENCES

[1] J. Hughes, Why Functional Programming Matters, *Computer Journal, 32*(2), 1989, 98-107.

[2] T.A. Standish, Extensibility in Programming Language Design, *Proc. Natnl. Computer. Conf.*, 1975, 287-290.

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

[4] www.haskell.org

[5] 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, 14-18.

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

[7] H. Boehm & R. Cartwright, Exact Real Arithmetic: Formulating Real Numbers as Functions, in D.A. Turner (ed.), *Research Topics in Functional Programming* (Addison-Wesley, 1990), 43-64.

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

[9] J.S. Royer, & J. Case, *Subrecursive Programming Systems: Complexity & Succinctness* (Birkhauser, 1994).

[10] D.A. Turner, Elementary Strong Functional Programming", *Proc. 1st International Symposium on Functional Programming Languages in Education*, *Springer LNCS, 1022*, 1995, 1-13.

[11] G. Hutton, A Tutorial on the Universality and Expressiveness of Fold, *Journal of Functional Programming, 9*(4), 1999, 355-372.

[12] R. Milner, A Theory of Type Polymorphism in Programming, *Journal Computer and System Sciences, 17,* 1977, 348-375.

[13] J.C. Reynolds, Three approaches to type structure, *Mathematical Foundations of Software Development, Springer LNCS, 185*, 1985.