WHY FUNCTIONAL PROGRAMMING REALLY MATTERS

Paul A. Driessen
Colin J.M. Kemp
Ian D. Peake
Susan Sedgley
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072
AUSTRALIA
(paul, ck, ian.peake, sedgley)@tetr.qe.edu.au

ABSTRACT

The significance of functional programming has revealed itself as far as theoretical or practically possible, the use of in- one symbolic data with animated, functional representations. In doing so, we explain the wide variety of applications to date, provide links back to mainstream functional programming, and indicate the potential of this functional style beyond current programming.

KEY WORDS

Programming, Tools and Languages, Functional Pro- grammers, Language Extensibility, Software Design and Development

1. WHAT IS FUNCTIONAL PROGRAM- ming REALLY ABOUT?

Functional programming [1] embraces numerous topics and themes, such as "applicative" programming, "state- flow", first-class data structures, programming with "combining forms", lazy evaluation and lazy-stdlib func- tions. Because all of these effects can be derived from the last, it's reasonable to propose that the facility to define and proct functions as first-class values, including func- tions- 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 "op- tional 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 in- one symbolic data with animated, functional representations. In doing so, we explain the wide variety of applications to date, provide links back to mainstream functional programming, and indicate the potential of this functional style beyond current programming.

2. FUNCTIONAL PROGRAMMING EN- ABLES DEFINITIONAL EXTENSION

Functional programming enables flexible language exten- sion by normal programmers, excluding the need for complex interpretation in favour of simple direct defin- ition.

1.1. Language Extension as Programming

Language extension is similarly a normal "programming" activity. This is because extensibility is a response to the tension between basic language design criteria of expres- siveness (tending toward large and mature languages) vs simplicity (tending to small, 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 lan- guages 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 means means that a programmer would normally use. Standard [2] identifies the following kinds of extension:

1. orthographic: the new "guts" constructs are achieved through extending the implementation of the "host"
language.

2. *Non-penultimate* guest constructs are modifications of host constructs achieved through corresponding modification of the host's implementation;

3. *Penultimate* guest constructs are defined in terms of existing constructs on the host.

Of these, it is clear that the third, "penultimate," style is preferable on the ground that it precisely identifies the kind of language extension feasible performed by programmers through declarations in the course of "normal" programming by normal programs—i.e., 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 translation), they threaten the integrity of the resulting software, in that unintended changes in behavior of host constructs may be effected.

2.2. *Flexible Extension: Definition (vs Interpretation*)

The essence of our difficulty with non-penultimate extensions is that they are 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 compatible) guest, but this allows for indirect paraphrasial 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 unparaphrase or rephrasing, it is similarly rejected.

That is, the effective principle of flexible language extension is that a new guest should be represented by direct paraphrasial definition (hence "Definitional") rather than by data that need somehow to be interpreted through modification or recreation of an interpreter (hence "Interpretative").

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 is that a host language directly express all elements in its semantic domain to the degree that it be expressively complete. Expressive completeness thus means that all entities in a language's semantic domain are directly corexpressible (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 [1], but the latter is in some sense more expressive/expressive (formally: more expressively complete). Whereas Haskell can directly represent and operate on non-finite sets (as characterized predicates):

\[
\emptyset \text{ elt } = \text{False} \\
\text{singleton } x \text{ elt } = x = \text{elt} \\
\text{union } x1 \text{ elt } = \text{if elt } \text{ then } x1 \text{ else elt}
\]

Pascal can't, because set-theoretic operators 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 expressiveness completeness, i.e., for flexible language extendibility by definition rather than interpreta-

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 and extension artefacts. The correspondence between language design and programming is initially discernible in correspondences between the names of the outputs of these processes. Language extension is implicitly included in the correspondence, 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 its life cycle its components are studied and learned by its developers and maintainers, 

920
unquestionable significance (how many new rare practically about Algo? after less than 25 years?)

3.1.2. Language complexity vs. software component complexity: A significant application of programming language typically provides at least 10^2 basic constructs. The number of components at the module-let show 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 programming organisation in the development and maintenance of a software system during its lifetime, the phenomenon of numerous waiting periods to make local adaptations or emergency repairs is widespread, thus providing another increase in the magnitude of the user population for a program component. Moreover, the aforementioned 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 design 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 therefore transparent as adequacy-enhancing language extension exercise.

3.2.2. Orthogonality vs. loose coupling: orthogonal language constructs of the right type constitute, etc. can be freely compared without exceptional behaviours in specific cases. Loose-coupling of modules is a means of reducing the accidental erroneous interactions in programs.

3.2.3. Simplicity vs. cohesiveness: cohesive program components are easy to understand (because the relevant information is contextually-located, and predictable in their use (at least partly because they will tend to be loose-coupled). Simple language constructs are (therefore so because they are easily-described and tend to wards orthogonality as a means of dealing with inherent complexities.

3.2.4. Resalabiity vs. non-reusability: the derivation of correct semantic notes from lexical appearance is the essence of a language's readability (eq. 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/data-type.

3.3. Basis in denotational semantics

3.3.1. Denotation of statements that declarations, the essence of any modular programming discipline, effect language extensions. First, consider the typical denotational meaning function M with signature M :: Sig -> Den where Sig is the domain of representation/syntax of programs; Dom is the domain of meaning/semantics of programs; and Env is the domain of environments, i.e, mappings identities to elements in Dom to which they are bound by prevailing declarations. Thus Env = id -> Dom. For example, the semantics of an identifier occurrence i is determined by accessing the environment p:

\[ M[i] p = \text{Env}(i) p \]

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

\[ M[i \leftarrow E] p = \text{Env}(i) p \]

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

\[ M[D ; i \leftarrow E] = M[D \cup \{i \rightarrow E\}] \]

Now, consider simply interchanging the order of M, i.e. changing the signature to M :: Env -> Den. Semantics of expressions are unchanged, except that operands of M are reversed, e.g.

\[ M[i \leftarrow E] p = \text{Env}(i) \]

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

\[ M[i \leftarrow E] m = \text{Env}(i) \]

and semantics for a program written conditionally:

\[ M[p ; i \leftarrow E] = \begin{cases} M[i \leftarrow E] & \text{if } p \text{ is true} \\ M[i \leftarrow E] \cup \{i \rightarrow M[M[i \leftarrow E]]\} & \text{otherwise} \end{cases} \]

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

\[ M;p = MM[D ; i \leftarrow E] = \begin{cases} M[i \leftarrow E] & \text{if } p \text{ is true} \\ M[i \leftarrow E] \cup \{i \rightarrow M[M[M[i \leftarrow E]]] \} & \text{otherwise} \end{cases} \]
In a real sense, MM (or "M p") effectively defines the prevailing programming language, assuming as it does meanings to all symbols, both built-in syntax and identifiers defined so far by declaration. Correspondingly, "MM [ or D F ]" 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 arguments for the Committee:

1. the pragmatic correspondence 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, i.e., independent establishment of which above serves to corroborate 1.;

3. from 1., thus corroborated, the converse should follow, i.e., that programming is a kind of language extension;

4. independent corroborations of 3. derive 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 in language extension should be processed (executed, evaluated and constrained) as a kind of programming (i.e., definitionally not interpretationally), so should programming be processed as a kind of language extension.

4. FUNCTIONAL PROGRAMMING ENABLES DEFINITONAL 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 processed definitionally and 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 that revealed in terms of the opportunity it offers to pursue programming subject to these language extension conditions, i.e., definitionally not interpretationally.

4.1. Preservation of Interpreters

To date however interpretation is still throughout program. Consider the following simple subtypes and operations thereon:

```plaintext
not False = True
and True x = x
and False x = False
data Nat = Zero | Succ Nat
add Zero m = m
add (Succ n) m = Succ (add m n)
mul Zero x = Zero
mul (Succ n) x = 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 "Mini-language" that each data type comprises: "not", "and" for "bool"; "add", "mul" for "Nat".

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

```plaintext
if_thenElse True x y = x
if_thenElse False x y = y
not x = if_thenElse True False x y

if m = f zero m f x then f else f 0 m x
mul m n = f zero m f n
```

Saliency points:

- note how the characterisation of interpretation (branching, recursion/cycling) are absent from the other-side direct definition of "not", "and", "add", "mul"; rather these characterisations have been localised in the respective interpreters "if_then_else" and "f";
- each interpreter embeds a proposed universal or objective behaviour for its respective type;
- the proposed objective behaviour for boolean is branching; and that for natural is iteration;
- other types are expected to have their own such behaviours and interpreters.

To summarise:

- operations on symbolic data take the essential structure of interpreters, thus symbolic data types can be thought of as "mini-languages" interpreted by their operations
- alternatively, the interpreter for each mini-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-
4.3. Pure vs. Impure Platonist Combinators

There is no reason to suppose platonist combinator is restricted to such simple examples. For example, platonist combinator exists for data structures. The platonist version of a list is order of elements X1...Xn for processing by some operation as list. Accordingly, such a list takes the (platonist) form
\[ \text{lsp} \, b \, \text{op} \, \text{X1} \, \text{op} \, \text{X2} \, \ldots, \text{op} \, \text{Xn} \, \text{b} \, \ldots \] where b is the value for the empty case. Platonist lists have generators "nil" and "cons" according:

- \[ \text{nil} := \text{b} \]
- \[ \text{cons} := \text{xx} \, \text{op} \, \text{b} := \text{xx} \, \text{op} \, \text{b} \]

Operations on platonist lists may be programmed definitional (i.e., straightforwardly, e.g.,

- \[ \text{total} \, \text{xx} := \text{xx} \, \text{op} \, \text{b} \, \text{xx} \, \text{op} \, \text{b} \, \ldots \]
- \[ \text{length} \, \text{xx} := \text{xx} \, \text{op} \, \text{nil} \, \text{nil} \, \text{op} \, \text{b} \, \text{xx} \, \text{op} \, \text{b} \, \ldots \]
- \[ \text{concat} \, \text{xx} \, \text{yy} := \text{xx} \, \text{op} \, \text{yy} \, \text{op} \, \text{b} \, \text{xx} \, \text{op} \, \text{yy} \, \text{op} \, \text{b} \, \ldots \]

Structures such as lists can be regarded as definitional "pure" platonist combinator (PCCs). In fact, they are nothing inherent to the structure that compels interpretation. However, some structures are not to pure, for example sets. The internal realism is realized in terms of a function that embodies its essential objective behavior (membership testing). However, the equality test on partial elements in a "singleton" necessarily involves ultimate interpretation, i.e., symbolic comparison of data in the predicate for "singleton.

Such "pure platonist combinator" (PCC) thus provides a potential bridge between our somewhat rigorous database ideal and a scenario where the totally functional programming style can be used in hybrid mode, if not to expose entirely the evil of interpretation, at least to mitigate them in practical settings.

4.4. Practicality of Platonist Combinators

Indeed, PCCs can be discovered widely among some of the more interesting applications of higher-order functional programming. As well as indicating the pragmatic utility of PCCs, these examples tend to speculate that as the extension of many more functional programming applications may be attributable to their replacement of symbolic data by platonist combinator.

4.4.1. Combinator parser: a good example of the rewrite of interpretation of data structures by direct functional representation is provided by the unification of grammars and parsers in the "normal form" power [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 parser.

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

4.4.3. Programmed graph reduction: either than represent a section as a graph into which an interpreter substitutes "open" rewrites, represent as the program which builds the resulting graph for its functional representation [8].

4.4.4. Subsemmata: because the definitional style eschews recursion, there would seem to be a link with the entire "subprogram" programming style [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 exposing a particular paradigm within functional programming — structuring programs around "fold" operators. Thus, the discovery of plasmonic combinatorics derives from the specification of the datatypes for which PCs and PPs provide alternative representations.

5.1. PCs

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

\[
\text{fold}_b \circ \phi \circ \beta = b
\]

\[
\text{fold}_b \circ \beta (c x) = c \circ \phi (\text{fold}_b \circ \beta x)
\]

The differences may be reconciled by unfolding 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 variates.

\[
\text{fold}_b \text{True} = b
\]

\[
\text{fold}_b \text{False} = b
\]

\[
\text{fold}_b \text{Zero} = b
\]

\[
\text{fold}_b \text{Succ} \circ a = \text{Succ} \circ (\text{fold}_b \circ a)
\]

(It's instructive to view the earlier-proposed "universal behaviour" for booleans and naturals — "if then else" and "iter" — as the intermediate stage between "fold" and PPC generators.)

A highly relevant aspect of functional programming with fold is the encapsulation of proof rules, to match the packaging of recursion in fold. One of the most significant is "Reader", which is replaceable in modified form to the PPC-variants of fold e.g. for plasmonic lists L:

\[
\text{L}_F \text{F}_2 \sim \text{L}_E \text{E}_2
\]

provided that

\[
\text{E}_2 (\text{E}_1 \text{X}) = \text{E}_1 \text{X} (\text{H}_2 \text{X})
\]

and similarly for all other PPs.

5.2. PCs

In the impure case the situation is a little more complicated. Whereas PPCs' generators are characterized (like folds) by their type-signature on the datatype constructor, PCs need to account additionally for the specification of a selector, in which any interpretation will be contained. Fortunately, it's implied in the hypothesis of a universal behaviour for a datatype that there be exactly one selector for each datatype. Derivation of PPC generators from the specifications (constructors & selectors) is perhaps clever but nonetheless straightforward. We use the earlier example of an PPC version of size (represented by characteristic predicates) to illustrate by steps as follows:

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

\[
\text{data Set} = \text{Empty} \cdot \text{Singleton} \circ \text{Union} \circ \text{Set}
\]

the PPCs corresponding to sets S are just the characteristic predicates definable by the partial application "\text{member \ S}"

2. The PPC 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 "\text{member \ S}" above to a set data structure, we seek to construct the characteristic predicate directly by means of applying PPC generators:

\[
\text{empty member \ Empty}
\]

\[
\text{singleton X \ member \ (\text{Singleton} \ X)}
\]

\[
\text{union P F T \ member \ (\text{Union} \ S \ T)}
\]

3. The solution involves reconsideration of PPCs. As inclusion of the adjective is what differentiates an PPC datatype from a PP datatype, setting aside the selector allows an underlying PPC datatype to be discerned. For these acts the PPC generators would be "\text{null} \circ \text{union}" and "\text{member} \circ \text{singleton}"

\[
\text{null} \text{\times} \text{\times} \text{\times} = \text{\times}
\]

\[
\text{null} \circ \text{\times} \text{\times} = \text{\times} (\text{null} \circ \text{\times} \text{\times}) (\text{\times} \text{\times} \text{\times})
\]

Note that whereas SI are conventional data structures, PI are corresponding PPCs (characteristic predicates).

em \ s \ e \ c \ r \ y = \ r \ e \ c \ n \ e\ r\ s e\ r \ c \ l \ e \ t s

\[
\text{null} \text{\times} \text{\times} \text{\times} = \text{\times}
\]

\[
\text{null} \circ \text{\times} \text{\times} = \text{\times} (\text{null} \circ \text{\times} \text{\times}) (\text{\times} \text{\times} \text{\times})
\]

924
The union law for the "set" PPCs is
\[ S X \cup F X = H (S E X E) \]
provided that
\[ H E = F I \]
\[ H (E E X) = F X \]
\[ (E X) S X = F (H S I) (O F S) \]

As with fusion, it also follows from the definitions of PPC generators (i.e., \( \text{set "fold" logic} \)) that there is a correspondence between datatype constructors \( C \), PPC generators \( C E \), and PPC operands \( X \) (this is apparent in the PPC generators for sets above). Moreover, if some PPC derives from application of \( C I \): PPC = GI args ...

then
\[ \text{PPC X} = \text{Gi X} \text{ args} \]

In particular,
\[ \text{PPC Ci} = \text{Ci Gi args} = \text{PPC} \]
\[ \text{PPC Ci} = \text{Ci Gi args} \]

That is, application of a PPC to the generator 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:

\[ \text{empty X Empty Set} \cup \text{Sing X} \]
\[ \text{in P1 P2 Empty Sing Union} = \text{Sing X} \]
\[ \text{in 0 Empty Sing Union} \]
\[ \text{in X Empty Sing Union} \]
\[ \text{in P1 P2 empty sing union} \]

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

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

\[ \text{empty single union} \]
\[ \text{member in empty sing union} \]
\[ \text{in X empty sing union} \]

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

\[ \text{empty single union} \]
\[ \text{member in empty sing union} \]

- which is in the form in which fusion (3. above) is applicable. Thus it reduces to

\[ \text{member Empty empty sing \cup sing X} \]
\[ \text{member (Union S1 S2)} = \text{empty (member S1) (member S2)} \]

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

\[ \text{empty elt = false} \]
\[ \text{single x elt = x} \]

For the third requires a little elaboration. First, recognize that "member SI" refers to an PPC, i.e. a set of Haskell types to "build" the right hand side on.

\[ \text{union S1 S2} \]

Returning to the left hand side,

\[ \text{member (Union S1 S2)} \]

- (following "member")

\[ \text{elt = member S1 elt or member S2 elt} \]

- (inlining "member S1" to "S" as above)

\[ \text{elt = elt or elt S2 elt} \]

Resulting body yields the definition as required:

\[ \text{union S1 S2 elt = elt or elt S2 elt} \]

6. Beyond Functional Programming

The definitional functional programming style contrasted above seems to have significance beyond mere programming as follows:

6.1 Categorical Software Design Representation

One way of conceiving of the drawbacks of conventional, interpretational programming is that the more the behavior of software is controlled by the data, the more generic and less informative the corresponding code becomes. In the limiting case, the data becomes a program and the code a generic, fully-fledged but uninformative interpreter. This situation resembles the case of the software reverse engineering/design recovery context, where as well as the need to recover the design hidden in data, it's essential that the recovered design itself retain its data-oriented design information. It would appear that our definitional programming style may serve as a candidate for a canonical representation for software designs.

6.2 Analog and Systems Design

Our situation, where data are represented in terms of objective behavior, and the observers thereof exploit these behaviors, seems to recall analog computing, where computations are composed from physical components whose behaviors model the dynamics being computed with. Thus, rather than the binary 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 pole, analog by (Definitional, Continuous), and this definitional proposal by (Interpretational, Discrete/Symbolic), as in the following diagram:

<table>
<thead>
<tr>
<th>Interpretational</th>
<th>Digital</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definitional</td>
<td>Analog</td>
</tr>
</tbody>
</table>

925
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 TOTALY 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 characterization of a total subset of functional programming adequate to practical requirements?

• Does a wide range of interesting data types exist that have unique planar combinator classes? For example, in the combinator parser case, a grammar would seem to have a pair of essential operations: "par" and "unpar" — 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 planar combinators. For example, the obvious definition of exponentiation in Church numerals

\[ \text{pow} \ m \ n = n \times (m \text{ pow} m \ n) \]

fails to type check. A more subtle definition

\[ \text{pow} \ m \ n = n \times m \]

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 optimization.

8. CONCLUSIONS

Our idea is that the significance of functional program- ming/languages is that it provides the key to a languageextension-based view of software development and entitles 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 "planar" behaviours. The ability to create and use IPCs means that both worldviews can productively coexist, to further indicate by some apparently significant examples (combinator parser etc.). A conceptual link with analog computing offers much wider potential application than just what may be regarded as "programming."

REFERENCES


