Programming Without Data - Towards a Totally Functional Programming Style

Paul A. Bailes
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072 AUSTRALIA
paul@itee.uq.edu.au

Abstract

The identity of programming with language extension has interesting consequences. If language extension implies programming, then language extension should be constrained by the requirements of what can be expected of a programmer, in particular the avoidance of the need to write interpreters, which can be satisfied by an (augmented) functional basis. If programming implies language extension, then programming should explicitly be conducted so, in particular avoiding illegitimate extension techniques, specifically the interpretation of inert data. Comprehensive replacement of data by functions shows potential for achieving this outcome. While numerous technical objectives remain to be met, the implications of the effort have wide ramifications, not just for software development.

1. Introducing Totally Functional Programming

The ultimate goal of this research is to discover a radical simplification of software development. Our approach is to achieve this by exploring the functional paradigm to its limits, distinctively in our case through the complete replacement of all non-functional components, i.e. data and data structures, by appropriate applicative alternatives, in which functions are chosen that implement the essential applicative behaviour that is hypothesised to exist when processing each data (structure) type. The choice of functional programming as a basis is objectively suggested by the paradigm's demonstrable promise as a vehicle for simple, verifiable solutions to complex programming problems.

"Functional" programming [1] is about the exploitation of function definition and application in software development. It traditionally therefore includes "applicative" programming, but extends that paradigm to programmer-definable higher-order functions, i.e. functions with functions as their arguments and distinctively with functions as their results. All of the apparent characteristics of applicative/functional programming (e.g. recursion, lazy evaluation) can indeed be explained as particular cases of higher-order functions. Use of higher-order functions comprehensively (or "totally") to replace data structures is logical fulfillment of this scheme of things.

A further consequence of this approach appears to be a lessening, if not abolition, of the need for general recursive capabilities in our totally functional language. Hence the term "Totally Functional Programming" (TFP) refers both to the total dependence upon functions as basis for programming, as well as the increased possibility of identifying a subrecursive functional programming language subset, in which all programs terminate (i.e. are total functions).

2. Programming = Language Extension

The mutual implications of programming and language extension both justify their identity as well as shedding important light on the nature of each, including a mandate for functional programming.

2.1. Programming as language extension

There are some compelling pragmatic correspondences between programming and language design. Moreover, as section 2.2 below demonstrates, language extension serves as a ubiquitous metaphor for language design. Finally, there is a basis in modified denotational semantics for the view that programmer products, in the form of declarations, are formally language extensions. Consequently, the conclusion is hard to escape that the semantic constructs created and identified by a programmer are as much linguistic entities as those produced by a language designer, and that the programming process can be viewed as one of language extension.

2.1.1. Pragmatic correspondences. Summarising [2], there's an interesting correspondence between criteria for assessing the quality of programs vs language designs:

- program correctness against requirements and specifications corresponds to language adequacy;
- a program's loosely-coupled architecture corresponds to the orthogonality of a language's components;
- a program's cohesive architecture corresponds to a language's simplicity;

- selection of meaningful identifiers in a program corresponds to well-designed language concrete syntax.

2.1.2. Formal basis. Moreover, in terms of denotational semantics [3] it's demonstrable [4] that declarations, the essence of any modular programming discipline, are in a formal sense 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 ⟧ ρ = ρ ⟦ 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 ⟦ **let** I = E ⟧ ρ =
(λ i . **if** i = ⟦ I ⟧ **then** M ⟦ E ⟧ ρ
else ρ ⟦ 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 ⟧ ρ = M ⟦ E ⟧ (M ⟦ D ⟧ ρ)

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 ρ ⟦ I ⟧ = ρ ⟦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 ρ ⟦ **let** I = E ⟧ =
M
(λi. **if** i = ⟦ I ⟧ **then** M ρ ⟦ E ⟧
else ρ ⟦ I ⟧
)

and semantics for a program rewritten consistently:

M ρ ⟦ D ; E ⟧ = M ρ ⟦ D ⟧ ⟦ E ⟧

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

M ρ = MM
where
MM ⟦ **let** I = E ⟧ =
M
(λi. **if** i = ⟦ I ⟧ **then** MM ⟦ E ⟧
else ρ ⟦ I ⟧
)
MM ⟦ D ; E ⟧ = MM ⟦ D ⟧ ⟦ E ⟧

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.

2.2. Language Extension as Programming

Programming is correspondingly a legitimate language development paradigm with specific demands of language adequacy. There are complementary cues that suggest equally-strongly that languages are advantageously developed as extensions of a simple base language:

- it's a long-standing practice for many languages [5, 6] to define core infrastructural components through libraries (so-called "standard preludes");
- increasingly expressive programming languages allow increasingly "deeper" infrastructure to be provided in this way.

Possible reasons for this are greater transparency of language definitions, and more flexible access to development resources (library programmers are more readily-available than language implementors).

2.2.1. Modes of language extension - "definitional" vs "interpretational". "Definitional" extensions are very much preferable to "interpretational" extensions because the former make reasonable demands of programmers, unlike the latter.

The association of language extension with programming further implies that language extensions should be as much as possible achieved by the sorts of means that a programmer would normally use.

Standish [7] 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" means is preferable on the ground that it precisely identifies the kind of language extension performed by programmers through declarations in the course of "normal"

programming – it will be observed that this was one of the pragmatic bases for associating programming with language extension as exposed above.

However, we may further distinguish within paraphrase between direct and indirect 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.

Direct paraphrase is therefore the desirable form of extension that is compatible with the "programming" view of language extension. The other extension forms may be discounted on grounds as follows:

- orthophrase and metaphrase require access to the language's interpreter (by which we include implementation via compiler or other translator);
- indirect paraphrase requires creation of an interpreter;
- in any case, they threaten the integrity of the resulting software, in that unintended changes in behaviour of host constructs may be effected.

That is, the effective principle of 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").

2.2.2. Language extension requires expressive completeness. Expressively-complete bases are necessary for reasonable programming of definitional language extensions.

Whatever the reasons, if language development is to proceed through programmer-extension (Standish's "paraphrase") of smallest base languages, an essential requirement is that base languages are capable of expressing all semantic entities of conceivable extensions. Excluding the option of programming an interpreter (like a Universal Turing Machine) for an extension, the expressive requirement is precisely that the base language be expressively complete [8].

Expressive completeness means that all entities in a language's semantic domain are directly expressible (without recourse to writing an interpreter) by terms in the language. A summary of distinctive technical features of an expressively-complete version of a typical modern programming language is

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

Requirements 1. and 2. are satisfied by typed lambda calculus/functional languages [1]. Item 3. is less frequently supported (but see our [9]), nevertheless is readily achievable through a parallel existential quantification operator E, such that

P ⊥	∃Xi∈ [X1,...]•P Xi	E [X1, …] P
⊥	⊥	⊥
⊥	**false**	**false**
⊥	**true**	**true**
false	**false**	**false**
true	**true**	**true**

That is, "E [X1, …, Xn] P" determines as far as possible whether or not one of the Xi satisfies P. Note especially that when "P ⊥" (alternatively "**not (**P ⊥)"), necessarily all "P Xi" (alternatively "**not** (P Xi)") because of the monotonicity of P. The cases in which quantification over an infinite list is in fact computable are thus handled.

This E can be used to define other parallel logical connectives. For example, symmetric "parallel" conjunction and disjunction as specified:

x	y	x ∧ y	x ∨ y
⊥	⊥	⊥	⊥
⊥	**false**	**false**	⊥
⊥	**true**	⊥	**true**
false	⊥	**false**	⊥
false	**false**	**false**	**false**
false	**true**	**false**	**true**
true	⊥	⊥	**true**
true	**false**	**false**	**true**
true	**true**	**true**	**true**

may be defined in terms of E above:

let x ∨ y = E [x, y] (λ x . x)
let x ∧ y = **not** (E [x, y] (λ x . **not** x))

3. Implication: Data-less Programming

Now that we see that programming should be perceived and pursued as a kind of language extension, then the

disciplines that apply to language extension (likewise perceived as a kind of programming) necessarily apply. In particular, because paraphrastic extension embraces both desirable direct and undesirable indirect extension modes (direct = Definitional vs indirect = Interpretational), vigilance against such Interpretational extensions is important.

3.1. Data implies interpretation

The characteristic sign of an undesirable Interpretational extension is the presence of symbolic representations of operations that thus require interpretation. We now go so far as to hypothesise that the presence of data is a complementary sign of an undesirable Interpretational extension!

For example, consider the representation of natural numbers as symbols

```
data Nat = Zero | Succ Nat
```

Completion of this language extension requires programming of a collection of mutually-consistent interpreters for the symbols Zero, Succ (Zero), etc. For example:

- iteration:

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

- arithmetic:

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

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

etc.
```

Our hypothesis would be supported, and at the same time the problems inherent in programming being pervaded by Interpretational extension solved, if we could identify alternative Definitional extension techniques to replace the use of symbolic data in programming.

3.2. Eschewing interpretation

In the search for a Definitional programming style as a substitute for Interpretation, we look to the kind of software technology that is required as a basis for Definition, as exposed above – functional programming. From an extension point of view, this means that instead of representing new "data" types by symbols, we instead represent them by the functions that their interpreters would have reflected.

For example, it's arguable that the purpose of natural numbers as numbers *per se* (as opposed e.g. to serial labels) is to control iterative processes. Thus for example, instead of the "iter" function above, instead directly represent natural numbers as iterators:

```
zero f x = x
succ zero f x = f  x
succ (succ zero) f x = f (f x)
etc.
```

This is well-known as the "Church numeral" [10] representation, where a natural number N is represented as N-fold composition. The viability of this arrangement as a Definitional extension is proven by the existence of a definition for successor:

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

so that only the two equations for "zero" and "succ" are needed to define all naturals.

Consistent definitions defining "higher" operations are achievable, e.g.

```
add m n = m succ n
mul m n = m (add n) zero
etc.
```

Note how simple these equations are compared to the corresponding definitions of section 3.1; in particular, there is no recursion and no pattern-matching/testing of argument values.

3.3. "Platonic Combinators"

The principle being followed in the above presentation is to replace symbols + interpretations by functions that directly effect the desired interpretations. The underlying hypothesis is that for every data type, there exists an inherent operation (function, or combinator) that embodies the essence (hence "platonic combinator") of that type. For example, Church numerals are the platonic combinators for natural numbers, because the essence of a natural number is to iterate.

In other words, echoing the general advantage of Definition over Interpretation, modelling a new type by inherent functions means that the semantics of the type is directly constituted in the type's elements, without requiring separate interpretation of the symbols that would instead represent these elements.

4. Pure vs. Impure Platonic Combinators

We're not ready to demand that data be avoided entirely, but that its avoidance where possible results in simplifications. Thus, platonic combinators can be partitioned into "pure" and "impure" classes: the former in which no symbolic data are necessarily involved; the latter in which data may be so.

4.1. Pure platonic combinators – fold functions

Besides Church numerals, some other simple platonic combinators are those for boolean and list types.

4.1.1. Booleans. The two truth values are defined by combinators:

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

exposing the platonic essence of the boolean type as making a choice. Higher boolean operators are simple:

```
and x y = x y false
or x y = x true y
not x = x false true
etc.
```

4.1.2. Lists. The platonic purpose of a list is to order its elements for processing by some function:

```
nil c n = n
cons x xs c n = c x (xs c n)
```

That is, any list "nil" or "cons x xs" is a function that takes as arguments:

- 'c' – a function that will compose the head element of a list with the result of applying the function that is the tail of the list;
- 'n' – the "base value" for the empty list.

Higher list operators are definable, e.g.:

```
length list =
        list increment 0
        where
        increment x n = n+1
```

4.1.3. Folds. Generally, for any regular type, a pure platonic combinator 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, and the equations for the usual functional "fold":

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

(reconcile differences by reordering operands and fusing "fold" and the list itself). Earlier platonic combinators for booleans and natural numbers (Church numerals) are likewise simplifications of the relevant fold-variants.

4.2. Impure platonic combinators – significant examples

The pure combinators above both notably eschew data entirely, and are useful to a significant degree. However, there are other, precedented examples which at least partially fulfill the ideal of replacing data by functions, but which still make some use of symbolic data, and which serve to validate the potential viability of TFP.

4.2.1. Sets. Non-cofinite sets (of which neither they nor their complements are finite) cannot be defined by enumeration. The only finite representation of such an infinite amount of information is as a function, in this case specifically by characteristic predicates. Platonically, a set *is* its membership test. Operations to construct sets therefore create or combine characteristic predicates:

```
empty x = false
singleton e x = x==e
union s1 s2 x = s1 x or s2 x
complement s x = not s x
etc.
```

Other "sets" with additional operations (e.g. remove member, count members, etc.) are actually more complicated types with undoubtedly more complicated platonic combinators awaiting discovery.

4.2.2. Combinator parsers. Recall the basis for the TFP approach is the replacement of interpretation of data structures by direct functional representation, inspired by the different Definitional vs Interpretational approaches to language extension. An example where the linguistic parallel perhaps becomes clearer is in the case of context-free parsing. Typically, a parser is realised in terms of a representation of a metalanguage or grammar (often in optimised terms, e.g. LR parse table) animated by a parsing engine [12], which is rather obviously a case of interpretation. However, the alternative TFP-compatible approach is to represent grammar components (terminal & nonterminal symbols) by their parsers, and to implement context-free compositions of concatenation and alternation by higher-order functions that operate on parsers to produce "larger" parsers. The independent existence of such so-called "combinator" parsers [13] seems to provide powerful independent support for TFP.

4.2.3. Exact real arithmetic. Finally, Boehm & Cartwright [14] identify a class of impure platonic combinators for exact real arithmetic. Basically, a real number is represented by a function which computes a real to any required rational precision.

5. Technical Challenges for TFP

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 (though it has to be emphasised that any higher-order functional programming implicitly recognises the value of replacing data by functions).

5.1. Objectivity of TFP

Some of our hypotheses in particular demand validation, e.g.:

- is there a characterisation of a total subset of functional programming adequate to practical TFP?
- 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?

Our preferred approach to these issues is first to study the existing corpus of functional programming experience to attempt to find evidence to support these contentions.

5.2. Type-theoretic issues

First up, the Hindley-Milner [15] -derived 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 nunerals

```
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 to support TFP. Many such exist, but the question of our ultimate quest for TFP as a subrecursive programming language raises the question of a convergence between TFP, a suitable type system for it, and other programs-as-types convergences such as Martin-Lof’s [16].

5.3. Programming methodology

Pure platonic combinators – folds over regular datatypes – are uniformly derivable from type signatures. However, it appears that interesting types (e.g. as simple as sets represented by characteristic predicates) require more sophisticated derivations. If TFP is to support non-trivial applications, then there need to be suitable calculi of program derivation, corresponding to those for functional programming in general [1].

Also, there’s an interesting contrast between TFP and OOP: a platonic combinator purports to be the single method applicable to an object. No other methods/attributes exist. Our abovementioned quest for unique platonic combinators classes for interesting types (e.g. grammars) is driven by the need to effect a reconciliation in this regard.

5.4. Implementation

Finally, functional programs are notorious for poor performance. While great strides have been made in improving the efficiency of implementations [17], it is likely that, as with type-checking, some of the hitherto uncommon patterns of functional definition and application will exceed the bounds of optimisations anticipated to date. However, if TFP can be found to be associated with a subset of functional programming, subrecursive or otherwise, it may be possible to use this subset as the key to TFP-specific implementation performance optimisations.

6. Implications beyond Programming

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

6.1. 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 [18], 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.

6.2. 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 hitehrto 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 (Interpretational, 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/prorotyping 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. 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. Conclusions

We have shown how a new programming paradigm derived from functional programming is motivated by the correspondence between programming and language design/extension. The potential validity of the paradigm derives directly from this motivation, as well as from a number of external factors:

- it provides a more thorough explanation for the benefits of functional programming than before [19];
- it is supported by the use of functional representations for non-trivial data, such as generalised "folds, combinator parsers and exact reals;
- ours is but one of a number of approaches entailing subrecursive languages (defining only total functions) [20, 21];

even though it has to be acknowledged that there is a non-negligible opinion [22] that would eschew the higher-order functions that are inherent in our approach.

8. Acknowledgements

This work benefitted from a conversation in its early stages with J.C Reynolds. I am grateful to my colleagues Ian Peake, Colin Kemp and Sean Seefried for their contributions to research reflected in this report, and in advance for their contributions to the further research foreshadowed.

9. References

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

[2] Bailes, P.A., "The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design)", *Proceedings of the 1986 Australian Software Engineering Conference*, Canberra, 1986, pp. 14-18.

[3] Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M.I.T. Press, Cambridge, 1977.

[4] Bailes, P.A., Chorvat, T. and Peake, I., "A Formal Basis for the Perception of Programming as a Language Design Activity", *Journal of Computing and Information, vol. 1, no. 1, Special Issue: Proceedings of the 6th International Conference on Computing and Information*, Trent University, 1994, pp. 1279-1294.

[5] Van Wijngaarden, A. (Ed.), Mailloux, B., Peck, J. and Koster, C., "Revised report on the Algorithmic Language ALGOL68", *Acta Informatica, vol. 5*, 1975, pp. 1-236.

[6] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, 1978.

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

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

[9] Bailes, P.A., Chapman, M., Gong, M. and Peake, I., "GRIT: an Extended Refine for More Executable Specifications", *Proceedings 8th Knowledge-Based Software Engineering Conference*, Chicago, IEEE, 1993, pp. 123-132.

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

[11] Sheard, T. and Fougaras, L., "A fold for all seasons", *Proc. ACM Conference on Functional Programming and Computer Architecture*, Springer, 1993.

[12] Aho, A.V. and Ullman, J.D., *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, 1972.

[13] Okasaki, C., "Functional Pearl: Even higher-order functions for parsing", *Journal of Functional Programming*, vol. 8, no.2, 1998, pp. 195-199.

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

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

[16] Martin-Löf, P., "Constructive mathematics and computer programming", in Cohen, L.J. et al. (eds.), *Logic, Methodology and Philosophy of Science VI*, North-Holland, Amsterdan, 1982.

[17] Asperti, A.,and S. Guerrini, S., "*The Optimal Implementation of Functional Programming Languages"***,** CUP, 1998

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

[19] Hughes, J., "Why Functional Programming Matters", *The Computer Journal, vol. 32, no. 2*, 1989, pp. 98-107.

[20] Royer, J.S., & J.Case, J., *Subrecursive Programming Systems: Complexity & Succintness*", Birkhauser, 1994.

[21] Turner, D.A., "Elementary Strong Functional Programming", *Proceedings of the first international symposium on Functional Programming Languages in Education*, Springer LNCS vol. 1022, 1995, pp. 1-13

[22] Goguen, J., "Higher-Order Functions Considered Unnecessry for Higher-Order Programming", in Turner, D.A. (ed.), *Research Topics in Functional Programming*, Addison-Wesley, 1990.