

Hosting Object-Oriented Programming in Totally Functional Programming with “Characteristic Methods”

Paul Bailes (paul@itee.uq.edu.au)

Colin Kemp (ck@itee.uq.edu.au)

School of Information Technology and Electrical Engineering

The University of Queensland QLD 4072 AUSTRALIA

Sean Seefried (sseefried@cse.unsw.edu.au)

School of Computer Science and Engineering

The University of New South Wales NSW 2052 AUSTRALIA

ABSTRACT

A functional representation for objects seems to offer an inherent cohesiveness for class methods. Functional representations for data are indicated by the affinity between programming and language extension: if language extension should exclude interpretation of symbolic data in favour of direct definition of functions, then in programming it should be preferable to discover and use directly through functional representations the applicative behaviours inherent in symbolic data. This approach is echoed in a fundamental affinity between object-oriented and functional programming, but the idea that every object has one characteristic applicative behaviour needs to be reconciled with the tradition of multiple methods in OOP. The higher-order methods to which we have recourse not only show promise in affecting this reconciliation, but also have potential to measure the semantic cohesion present in a multi-method class.

1. PRIMITIVE RELATIONSHIP BETWEEN FP AND OOP

Object-Oriented and Functional programming are liable to thought of as competing rather than collaborating technologies. This is because, for example:

- OOP tends to make free use of assignable variables, whereas FP emphasises referential transparency [1];
- FP tends to exploit implicit parametric polymorphism in the style of Milner [2], but which seems only capable of integration with inheritance to a limited degree (e.g. [3]).

However, neither of these clashes seems to be about essentials, e.g. respectively:

- OOP can just as well deal with abstract data types and instances, as it can directly with objects (i.e. assignable variables are not intrinsic);
- convenient as Milner-style polymorphism may be, it has limitations (e.g. see below) that indicate in favour of alternatives that might not be so incompatible with inheritance.

Rather, there is a strong coincidence in fundamentals, which we think can only be deepened, as follows.

1.1. Message Passing in FP

A primitive idea underlying OOP is that objects react to messages. In contemporary parlance, methods may be thought of as message channels, and their arguments as message contents.

In FP, it's relatively straightforward to represent data (structures) in terms of functions, that is transparently in message-passing style, as follows. Consider for example the data structure of ordered or “cons” pairs, definable axiomatically as follows:

$$car (cons X Y) = X$$

$$cdr (cons X Y) = Y$$

Message-passing is evident when we further define

$$cons x y m = \\ \text{if } m == \text{“car-message” then } x$$

*else if m == "cdr-message" then y
else error*

*car p = p "car-message"
cdr p = p "cdr-message"*

That is, a pair “cons X Y” is actually a function that applies to a message parameter ‘m’. Depending on the message received, the pair yields up either its first (“car”) or second (“cdr”) element. Selector functions “car” and “cdr” send the appropriate message to their pair-operand ‘p’.

1.2. Alternate Message-Passing Style

The beauty of this sort of synthesis (OOP in FP as above exemplifies numerous similar demonstrations, e.g. [4]) is as a demonstration of the usefulness of FP as a kind of “grand unified theory” of programming and languages, i.e. everything can be explained from a minimal (functional) basis. However, the specific synthesis remains only imperfectly satisfactory, at least in that it has not been limited to a minimal functional basis: the messages are symbolic data, whereas in the minimal but still universal functional language, the lambda-calculus [5], symbolic data are not core constructs.

Instead, we can re-cast our functional rendition of message passing using only functions, and not symbolic data, as follows:

*cons x y m = m x y
car p = p first where first a b = a
cdr p = p second where second a b = b*

1.3. Towards Generalisation

The purposes of this paper are thus to consider the “alternate” message-passing style immediately above:

- to establish authoritative motivation for a programming style based purely on functions to the exclusion symbolic data;
- to elaborate this “totally functional” programming (TFP) style, including its development from “traditional” FP;
- most importantly, to consider how it appears to help or hinder OOP, and in particular to derive from the appearance of hindering OOP, insights into a potential method for developing classes with guaranteed or at least measurable cohesion on a semantic rather than a merely syntactic level.

2. “INTERPRETATION” PERVADES PROGRAMMING

The key to our unifying approach to OOP and FP is to avoid symbolic data and instead to use functions as the preferred basis for conceptual modelling. The motivation for this approach originates in drawing a comparison between language extension and programming: if programming is a kind of language extension, then programming should be subject to the same constraints as language extension. By distinguishing between different kinds of language extension, and in particular avoiding those involving interpretation in favour of direct definition of functions, it follows that direct definitional programming (however conceived) should be preferred over programming that involves interpretation of symbolic data.

2.1. Programming vs Language Design/Extension

While it’s obvious (by definition) that language extension is a kind of programming, further reflection reveals programming as a kind of language design and thereby a kind of language extension.

2.1.1. “Program” = “Language”

Programming artefacts directly correspond to language design & extension artefacts. The correspondence between programming and language design (and thus language extension, as its products are the result of both programming and language design) is discernable in correspondences between the natures of the outputs of these processes, in summary as follows:

- software component libraries can be at least as long-lived as programming languages;

- software component libraries can have as many components (and be as difficult to learn) as programming languages;
- software component libraries can be as widely-distributed as programming languages.

2.1.2. “Programming” = “Language Design”

Further substantiation comes from the correspondence in the respective approaches taken to them by their practitioners. Our key observation [16] is that there is a correspondence between programming language design criteria and program quality criteria, in summary:

- adequacy of language constructs vs. application-focus of program components;
- orthogonality of language constructs vs. loose coupling of program components;
- simplicity of language constructs vs. cohesiveness of program components;
- readability of languages vs. choice of naming convention, layout etc. in programs.

2.1.3. Basis in denotational semantics

The ultimate unity of programming and language extension is objectively demonstrable via denotational semantics, which shows that declarations, the essence of any modular programming discipline, effect language extensions.

First simply interchange the operands of the usual denotational meaning function M , i.e. changing the signature to “ $M :: \text{Env} \rightarrow \text{Rep} \rightarrow \text{Dom}$ ” where: Rep is the domain of representations/syntax of programs; Dom is the domain of meanings/semantics of programs; and Env is the domain of environments, i.e. mappings from identifiers Id to elements in Dom to which they are bound by prevailing declarations and thus $\text{Env} = \text{Id} \rightarrow \text{Dom}$. Semantics of expressions are unchanged, save that operands of M are reordered from the usual, e.g.

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

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

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

and semantics for a program rewritten consistently:

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

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

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

In a real sense, MM (or “ $M \rho$ ”) effectively defines the prevailing programming language, ascribing as it does meanings to all symbols, both built-in syntax and identifiers defined so far by declaration. Correspondingly, “ $MM \llbracket D \rrbracket$ ” defines MM extended by D . That is, declaration D truly extends language MM .

2.1.4. Conclusion: Programming as Language Extension

We’ve now demonstrated in a number of ways that programs, or their components, can be viewed as if components of programming languages. Hence, programming can be viewed as a language development process. Because of the limited means by which programmers can effect language development (typically by definition/declaration) this means programming can be viewed as language extension, which contention is cemented from a formal semantic perspective.

2.2. “Definition” vs “Interpretation” in Language Extension

Language extension remains an implicitly-preferred method of developing and presenting language designs, e.g. the heavy use of libraries rather than complexes of primitive constructs, for OO, Functional and other languages alike. The key concept of practical language extension is that it’s the sort of activity that could be carried out by a programmer. Thus, within Standish’s taxonomy of extension mechanisms [6]:

- *orthophrase* – add new construct by modifying the implementation;

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

only the last of these qualifies as “practical”, the others requiring language technology skills and indeed inviting all sorts of problems with language processor integrity and reliability.

However, even with paraphrase, further distinction is necessary, highlighted by differences between the practical expressive power of different programming languages. While from a certain theoretical standpoint all “real” programming languages are Turing-complete and thus of equivalent expressive power, not all languages can express the same functions/programs in the same qualitative way. That is, whereas a more practically-expressive language can express a function directly as a term, a less practically-powerful language needs to resort to interpretation. Consider for example the abstraction of mathematical sets in terms of characteristic predicates. Because each set is a function, then operations that combine sets (union etc.) are higher-order (function-valued) functions. While these functions can be expressed directly within functional languages (such as Miranda [7], Haskell [8], etc.), in other languages (FORTRAN, Algol, C, Java, etc etc etc) they can only be achieved by interpretation of symbolic representations; in the most comprehensive solution, a FORTRAN/Algol/C/Java/etc programmer who wants to use sets as characteristic predicates should write a Miranda/Haskell interpreter!

It’s clear of the two extension styles within paraphrase – “interpretational” and “definitional” – that the former is just as problematic as the rejected alternatives to paraphrase, and that the latter is the sort of extension that is practicable by programmers.

(Note that the concept of “expressive completeness” has been developed [9] to characterise the direct expressiveness of a programming language without recourse to interpretation; and that it’s the greater expressive completeness of functional languages that makes them more practically extensible as a result.)

2.3. Interpretation Pervades Programming

If programming (as language extension) is to be subject to extensibility criteria, it now follows that programming should eschew interpretation instead of direct definition. It’s therefore provocative that interpretation can be discerned throughout programming. Consider for example the following standard definitions of elementary arithmetic operations: given successor “Succ” and zero ‘Zero’, we have

$$\begin{aligned} \text{add } m \text{ Zero} &= m \\ \text{add } m \text{ (Succ } n) &= \text{Succ (add } m \text{ } n) \end{aligned}$$

$$\begin{aligned} \text{mul } m \text{ Zero} &= \text{Zero} \\ \text{mul } m \text{ (Succ } n) &= \text{add } m \text{ (mul } m \text{ } n) \end{aligned}$$

$$\begin{aligned} \text{exp } m \text{ Zero} &= \text{Succ Zero} \\ \text{exp } m \text{ (Succ } n) &= \text{mul } m \text{ (exp } m \text{ } n) \end{aligned}$$

In each of these cases, we see patterns typical of interpreters:

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

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

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

Because this interpretational treatment of naturals is typical of the treatment of an approach to structured data which pervades programming, interpretation and its problems pervade programming.

3. EXPUNGING INTERPRETATION WITH “PLATONIC COMBINATORS”

Because programming is a kind of language extension, then just as there is a definitional style of language extension that avoids having to represent functions by symbols and then animating the intended behaviours by interpreting the symbols, there is a corresponding programming style that eschews symbolic data in favour of functional representations.

3.1. Removing Interpretation

A first step in reducing the problems of pervasive interpretation would appear to be the isolation of the multiple occurrences of the mini-language interpreter that has to be given each time data is used and interpreted. For our natural number example, what is the common interpretive pattern to each arithmetic definition? It’s obvious that each recurses on the ‘n’ parameter, i.e. n-fold iteration is taking place. We can make this explicit by defining

$$\begin{aligned} \text{iter Zero } f x &= x \\ \text{iter (Succ } n) f x &= f (\text{iter } n f x) \end{aligned}$$

and rewriting the other definitions:

$$\begin{aligned} \text{add } m n &= \text{iter } n \text{ Succ } m \\ \text{mul } m n &= n (\text{add } m) \text{ Zero} \\ \text{exp } m n &= n (\text{mul } m) (\text{Succ Zero}) \end{aligned}$$

As part of uniform application to natural numbers, what this means is

- we have identified iteration as the semantics of natural numbers (it’s understood that programmers may use natural numbers for purposes other than iteration, but we regard this as improper use of the type);
- use of natural numbers should involve application of the interpreter “iter”.

From the last point, it’s a short step to

- requiring that the interpreter might as well be fused with the interpreted data, which we call “animations”; and
- replacing data structure constructors with “generators” that produce these animations.

For example:

$$\begin{aligned} \text{zero } f x &= f \\ \text{succ } n f x &= f (n f x) \end{aligned}$$

and

$$\begin{aligned} \text{add } m n &= n \text{ succ } m \\ \text{mul } m n &= n (\text{add } m) \text{ zero} \\ \text{exp } m n &= n (\text{mul } m) (\text{succ zero}) \end{aligned}$$

3.2. “Platonic Combinators”

The idea underlying the above device is that for every data type, there is a characteristic behaviour pattern that embodies the essential semantics of the type, and thus the essence of each member of the type can be represented by a function, or “Platonic Combinator” (PC). Within the platonic combinators, there seem to be two distinct kinds as follows.

3.2.1. “Pure” platonic combinators and “fold”

The motivation for platonic combinators is to avoid interpretation. PCs that do do entirely we call “Pure Platonic Combinators” (PPCs), for example natural numbers as above. PPCs are intimately-related to the “fold” functions [10] that exist for regular recursive datatypes. For example for naturals, the list fold function “foldn”

$$\begin{aligned} \text{foldn } f x \text{ Zero} &= x \\ \text{foldn } f x (\text{Succ } n) &= f (\text{foldn } n f x) \end{aligned}$$

is obviously the natural number interpreter “iter” with parameters re-ordered. The structure of folds follows directly from the signature of a type’s constructors: for naturals

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

For lists, the signature

$$\text{data List } t = \text{Prepend } t (\text{List } t) | \text{Nil}$$

leads directly to the familiar list “fold”

$$\begin{aligned} \text{fold } o \ b \ \text{Nil} &= b \\ \text{fold } o \ b \ (\text{Prepend } x \ xs) &= o \ x \ (\text{fold } o \ b \ xs) \end{aligned}$$

from which we can progress do the direct generation of animated lists/platonic combinators:

$$\begin{aligned} \text{nil } o \ b &= b \\ \text{prepend } x \ xs \ o \ b &= o \ x \ (xs \ o \ b) \end{aligned}$$

That is, the applicative semantics of a list of elements is essentially to insert a given binary operation between the elements.

3.2.2. “Impure” platonic combinators

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

$$\text{data Set } t = \text{Empty} | \text{Singleton } t | \text{Union } (\text{Set } t) (\text{Set } t)$$

The membership operation is specified:

$$\begin{aligned} \text{member Empty } x &= \text{false} \\ \text{member (Singleton } e) \ x &= x == e \\ \text{member (Union } s1 \ s2) \ x &= \text{member } s1 \ x \ \text{or} \ \text{member } s2 \ x \end{aligned}$$

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

$$\text{member } s \ x = s \ x$$

and by which predicates may be generated

$$\begin{aligned} \text{empty } x &= \text{false} \\ \text{singleton } e \ x &= x == e \\ \text{union } s1 \ s2 \ x &= s1 \ x \ \text{or} \ s2 \ x \ \text{-- equivalently: member } s1 \ x \ \text{or} \ \text{member } s2 \ x \end{aligned}$$

Obviously:

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

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

Generators for IPCs can’t be derived from type-signatures as simply as for PPCs via fold; however fold-theory can be employed for derivation of IPCs [17].

3.3. Practicality of Platonic Combinators and a “Totally Functional” Programming Style

Platonic combinators thus form the basis for a “Totally Functional Programming” (TFP) style, alternative to “traditional” digital computing which is based on what we may reasonably attribute as the Turing-Von Neumann model inherent to which is the animation of symbolic data through interpretation. TFP on the other hand can be thought of as following the Church (lambda-calculus) model, which while hitherto perhaps regarded only as a theoretical curiosity, has a practical significance that can be validated by observing the significant role played by platonic combinators in computer science.

PPCs and IPCs exist for numerous basic programming language structures:

- PPCs for ordered pairs are as in the “alternate message-passing style” above

- PPCs for Booleans are the ordered pair auxiliary functions “first” and “second” (corresponding to true and false respectively)
- PPCs for natural number and list (as above)
- IPCs for sets (as above).

However, there are some interesting examples of advanced uses of higher-order functions that appear to be in fact platonic combinators:

- Hutton’s “combinator parsers” [12] in which grammars are not interpreted by some parsing engine, but are directly represented as their parsers, and context-free grammar combining forms of concatenation and alternation are rendered by appropriate (higher-order) functions on parsers;
- Boehm & Cartwright’s “exact real arithmetic” [13] in which a real number is represented by a function that computes an arbitrary approximation;
- implementation of functional languages by “programmed” graph reduction [14] in which rather than representing a lambda-term by a graph which is interpreted by a rewriting machine, the effect e.g. of performing substitutions in a graph is directly achieved by executable code.

We hypothesise that further examination of advanced applications of higher-order functional programming will yield further examples of platonic combinators.

4. “CHARACTERISTIC METHODS” FOR CLASSES

Further validation of our approach should derive from its compatibility with other, successful programming methodologies, in particular OOP. Obviously, PCs correspond to objects (and the function types of PCs correspond to classes). The problem to be overcome is that our objects appear to have serious limitations on the methods definable on them (i.e. one method per class/object). However, the means by which this limitation may be overcome has potential to contribute to the cohesion of methods on classes in OOP in general.

4.1. Methods and PCs

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

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

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

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

can be rendered by (Haskell) definitions

```
sentence = subject `cat` predicate
subject = tok "paul" `alt` tok "colin"
predicate = tok "eats" `alt` tok "sleeps" `alt` tok "works"
```

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

$$(p1 \text{ `alt` } p2) \text{ } xs = p1 \text{ } xs ++ p2 \text{ } xs$$

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

$$\text{split } xs = \text{take } (\text{length } xs) (\text{split}' \text{ } xs)$$

```

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

```

```

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

```

```

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

```

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

```

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

```

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

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

4.2. Generalised “Characteristic Methods”

Our solution exploits functional languages’ adherence to Tennent’s principles of abstraction and correspondence [15]. In particular, any component can be parameterised in terms of any of its sub-components. This means in particular that multiple methods on a class/object can be instantiated from a single “mother” characteristic method by suitable selection of formal and actual parameters, and that is this “mother” that is the single characteristic method of the class.

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

```

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

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

tok ts xs = ts == xs

```

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

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

```

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

```

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

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

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

For a grammar ‘g’ generated with these generic “alt”, “cat” and “tok”, we can instantiate a recogniser:

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

and a likewise a parser:

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

4.3. Characteristic Methods and Cohesion

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

```
(x.x)  $M_i$  =  $M_i$ 
```

This is indeed so, but rather than invalidating our approach, it suggests that the approach has wider applicability as a measure of the cohesion of multi-method classes, as follows. “Cohesion” is a valued attribute of software components, classes and objects not excepted. Measures of cohesion as developed to date (e.g. [18]) can be regarded as somewhat “syntactic” in that they don’t depend on knowledge of the functions of various methods. We’re optimistic that “mother” classes could serve as the basis of a more semantic approach to measuring cohesion, in that they at least embody abstractions of the behaviours of their instances/descendants, as follows. The arbitrarily-distinct methods M_i that would require the identity function as their “mother” should be regarded as not cohesive. On the other hand, methods that admitted a “mother” that embodied substantial common structure, with relatively minor differences expressible in terms of a few parameters, would be regarded as semantically cohesive. For example, the above treatment of parsers, recognisers and generic grammars suggests that parsing and recognition are cohesive.

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

5. CONCLUSIONS

A data-less “totally functional” programming style that served to unify FP with the message-passing foundations of OOP has been motivated and elaborated, and finally its relationship to OOP exposed and analysed. An apparent discrepancy, that functional representations for objects are restricted to single “characteristic method” method classes, is overcome by considering higher-order “mother” characteristic methods from which the other specific methods might be instantiated. These “mother” methods are not a device to escape from a contradictory reality, but in fact suggest a means by which the important cohesion of class and object methods may be measured in much more of a semantic way than previously.

5.1. Specific Research Direction

For the purposes of validating the relationship between FP/TFP and OOP, we would need first to exhibit some further practical examples of (i) how a practical application such as context-free

grammars could be rendered in TFP, and (ii) what sorts of “mothers” would need to be created to suit a variety of practical methods? Then, how could the specifications of these “mothers” be derived directly from requirements, rather than by abstraction from all the conventional methods? How could “mothers” be exploited to deduce methods hitherto unconceived-of?

Also, it would be necessary to develop the theoretical underpinnings of “mothers” as a measure of cohesion, in particular the “specificity ordering” contemplated above.

5.2. General Research Directions

Additionally, in order to make TFP viable, many other questions need to be resolved, most importantly:

1. How much can we learn from existing FP about TFP?
2. What is the precise distinction between Definition and Interpretation?
3. How may functional alternatives for data be derived systematically?
4. What is the semantic model for TFP?
5. How may TFP be type-checked?
6. May functional language infrastructure (implementations, validation regimes) be optimized for TFP?

6. ACKNOWLEDGEMENTS

We’re particularly grateful Ian Peake for his contributions to our early research on this topic.

7. REFERENCES

1. Turner, D.A., “Functional programs as executable specifications”, in Hoare, C.A.R. and Shepherdson, J.C. (eds.), *Mathematical Logic and Programming Languages*, pp. 29-54, Prentice-Hall, London (1985).
2. Milner, R., “A theory of type polymorphism in programming”, *J. Computer and System Sciences*, vol. 17, no. 3, pp. 348-375 (1978).
3. Bracha, G., Odersky, M., Stoutmaire, D. and Wadler, P., “Making the future safe for the past: Adding Genericity to the Java™ Programming Language”, *Proc. OOPSLA 98*, pp. 183-200 (1998).
4. Abelson, H., Sussman, G.J. and Sussman, J., “Structure and Interpretation of Computer Programs”, M.I.T. Press, Cambridge (1985).
5. Barendregt, H.P., “The Lambda Calculus - Its Syntax and Semantics”, North-Holland, Amsterdam (1984).
6. Standish, T.A., “Extensibility in Programming Language Design”, *Proc. Natnl. Comp. Conf.*, pp. 287-290 (1975).
7. Turner, D.A., “Miranda - a non-strict functional language with polymorphic types”, in Jouannaud (ed.), *Conference of Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, vol. 201, pp. 1-16, Springer, Berlin (1985).
8. www.haskell.org
9. Plotkin, G.D., “PCF Considered as a Programming Language”, *Theoretical Computer Science*, vol. 5, pp. 223-255 (1977).
10. Hutton, G., “A Tutorial on the Universality and Expressiveness of Fold”, *Journal of Functional Programming*, vol. 9, no. 4, pp. 355-372 (1999).
11. Bailes, P.A., Kemp, C.J.M., Peake, I.D. and Seefried, S., “Why Functional Programming *Really* Matters”, *Proceedings 21st IASTED International Multi-Conference on Applied Informatics (AI 2003)*, pp 919-926, Acta Press (2003).
12. Hutton, G., “Parsing Using Combinators”, *Proc. Glasgow Workshop on Functional Programming*, Springer (1989).
13. Boehm, H. and Cartwright, R., “Exact Real Arithmetic: Formulating Real Numbers as Functions”, in D.A. Turner (ed.), *Research Topics in Functional Programming*, pp. 43-64, Addison-Wesley (1990).
14. Peyton Jones, S., “The Implementation of Functional Programming Languages”, Prentice-Hall International, Hemel Hempstead (1987).
15. Tennent, R.D., “Language Design Methods Based on Semantic Principles”, *Acta Informatica*, vol. 8, pp. 97-112 (1977).
16. Bailes, P.A., “The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design)”, *Proc. 1986 Austrln. Software Eng. Conf.*, pp. 14-18 Canberra (1986).
17. Bailes, P.A. and Kemp, C., “Formal Methods within A Totally Functional Approach to Programming”, to appear in Aichernig, B.K. and Maibaum, T. (eds.), “Formal Methods at the Crossroads: from Panacea to Foundational Support”, *Proc. 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, March 18-21, 2002*, Springer-Verlag (2003).
18. Chidamber, S.R. and Kemerer, C.F., “Towards a Metrics Suite for Object Oriented Design”, *Proceedings of OOPSLA’91, ACM SIGPLAN Notices*, vol. 26, no. 11, p. 197 (Nov. 1991).