Formal Methods within a Totally Functional Approach to Programming

Paul A. Bailes, Colin J.M. Kemp

School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072 AUSTRALIA
`{paul, ck}@itee.uq.edu.au`

Abstract. Taking functional programming to its extremities in search of simplicity still requires integration with other development (e.g. formal) methods. Induction is the key to deriving and verifying functional programs, but can be simplified through packaging proofs with functions, particularly "folds", on data (structures). "Totally Functional Programming" avoids the complexities of interpretation by directly representing data (structures) as "platonic combinators" - the functions characteristic to the data. The link between the two simplifications is that platonic combinators are a kind of partially-applied fold, which means that platonic combinators inherit fold-theoretic properties, but with some apparent simplifications due to the platonic combinator representation. However, despite observable behaviour within functional programming that suggests that TFP is widely-applicable, significant work remains before TFP as such could be widely adopted.

1 Programming is Too Hard

There can be little doubt that "programming" (both as metaphor for and essence of the entirety of software development activity), after a half-century of unceasing research, remains too complex. The plethora of complaints about:

- performance of software systems;
- inefficiency of software development; and
- the proposed remedies that have failed to accomplish dramatic change

all encourage the search for a new view that will yield significant improvement.

Our ultimate goal accordingly is to discover a radical simplification of software development. Our starting point in the achievement of this goal is functional programming, but with an important difference from previous treatments. Whereas functional programming has hitherto combined applicative behaviours (functions) and inert symbols (data), our approach is through replacement to the greatest extent possible of non-functional components, i.e. data and data structures, by appropriate applicative alternatives, i.e. functions. In other words, what distinguishes our approach is that it explores the functional paradigm to its limits (hence "Totally Functional" programming). This will be done by discovering functions 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 no mere accident or preference, but is rather objectively suggested for this purpose in formal terms because it inherently supplies most *convenient* access to the richest range of computable functions compared to other paradigms (see discussion of “expressive completeness” in section 7.5 below).

However, radical as the new proposal may ultimately be proven to be, there is an obligation to demonstrate its integration with existing development methods, especially formal methods (which is the subject of this paper) for at least two reasons. First, a feature of our proposal is that it can co-exist with existing (functional programming) technologies. Thus, it’s important to coexist with other methods that use these technologies. Secondly, and more objectively, the our motivation for the new proposal does not contain any conflict in principle with formal methods, so a demonstration of formal-methods compatibility indicates the soundness of the derivation of the detail of our proposal from its motivation (in addition to enjoying the actual benefits of formal methods).

Finally, we would expect that if our proposal promises simplification in general, then some of this simplification should be observable from a formal methods point of view.

2 Formal Methods and Functional Programming

One of the major claims for functional programming is that the paradigm (strictly its pure applicative superset) supports formal methods in a more accessible, simple way than compared with imperative programming [1, 2]. Referential transparency permits equational reasoning on the texts of programs, using in which programmer-defined functions are accommodated naturally. Program branching is reflected by case analysis in derivations/proofs, and iteration/recursion is reflected by induction. An implicit requirement, that any new exploitation of functional programming such as we propose, should preserve the accessibility of formal methods, is met by exploiting the technique of “fusion”, which takes it place in the formal methods landscape of functional programming as follows.

2.1 Recursion and induction

For example (following Bird [3]), given the definition of a function to reverse a list[1] (equations numbered for reference)

1. *reverse [ ] = [ ]*
2. *reverse (x:xs) = reverse xs ++ [x]*

we prove that “reverse (reverse Xs) = Xs” for all finite lists Xs.

Case [ ]:

reverse (reverse [ ])

[1] generally, example code fragments will be rendered in Haskell [4] notation, which has become the *lingua franca* of the functional programming community.

= reverse.1
reverse [ ]
= reverse.1
[ ]

Case X:Xs

reverse (reverse (X:Xs))
= reverse.2
reverse (reverse Xs ++ [X])
= assuming "reverse (Ys ++ [Y])" = "Y : reverse Ys"
X : reverse (reverse Xs)
= induction hypothesis
X : Xs

The assumption is a typical example of a necessary lemma or "auxiliary result", and can be proved similarly.

Because:

- the same techniques are used in proving the equivalence of non-executable definitions with executable definitions of functions as are used in proving the equivalence of executable with executable definitions;
- the techniques for deriving equivalent definitions are the same as for proving equivalence of definitions

the notion of proof of equivalence of executable function definitions serves as a metaphor as well for

- proof of correctness against specifications
- synthesis of programs from proofs
- optimization of implementations.

2.2 Packaging list recursion & induction with "fold"

However, it's arguable that recursion, case analysis and induction place too-great demands upon programmers' intellectual resources:

- an over-emphasis on "unnatural" recursion is sometimes raised as an obstacle the wider adoption of applicative/functional programming even in contexts (such as learning introductory programming), even when the relative familiarity of functional programming's equational style of definition and rewriting model of evaluation would appear to be the solution to the complexity of imperative programming;
- in any case, programmers have better things to do (i.e. developing applications-oriented solutions) if the basic processes of program derivation and verification can be further simplified.

However, it's possible to package sub-proofs as general laws, and thereby obviate the need for repeated consideration of inductive proofs. Thus, the simplification of program structures that results from packaging code fragments into a hierarchy of components (e.g. procedures, functions, types, etc.) is paralleled by a simplification of the processes of program derivation and verification that result from the availability of packaged laws/theorems about these components. The above auxiliary result for "reverse" is an example, albeit with limited application.

A particular advantage of functional (as opposed to mere applicative) programming is that this process is applicable not just to first-order operations on data, but also to the higher-order constructs that combine and produce functional program components. That is, programmers may define their own control constructs, and derived laws about the behaviours of these constructs can be used in deriving and verifying programs that use them.

An outstanding example of this approach is how laws about the "fold" function can be used to derive and verify functional programs that operate on list structures, and indeed this approach is immediately extendable to all regular recursive datatypes, for which analogies to the list "fold" exist.

For example, the list "fold" function (earlier known as "reduce" in APL):

1. *fold op b [ ] = b*
2. *fold op b (x:xs) = op x (fold op b xs)*

is well-known (e.g. [3]) to simplify derivation or verification of functions on lists. Also for example, the definitions:

sum = fold (+) 0
map f = fold (\ x xs -> f x : xs) [ ]
reverse = fold append [ ] ***where*** *append x xs = xs ++ [x]*

(n.b. '\' is Haskell for 'λ') respectively define the functions:

- to sum elements of a list (of numbers);
- to apply a function f to each element of a list
- "reverse" but more elegantly than before.

The explicit recursion that would otherwise be required is encapsulated within "fold".

Consequently, formal proofs of list-processing functions need not depend upon induction, but rather depend upon (ultimately inductively-defined) laws about "fold".

Universal property of fold. The universal property of fold [5] is that

G = fold F V
iff
G [ ] = V and G (X : Xs) = F X (G Xs)

This property follows from the definition of "fold", and can be used in proofs about functions defined using "fold" without recourse to explicit induction. For example, using universality we reconsider the proof of "reverse (reverse Xs) = Xs" for all finite lists Xs. First, express reverse in terms of a self-inverse:

reverse . reverse = id

where "id" is identity, for lists Xs thus "id Xs = Xs". Then, expand

reverse . reverse = reverse . fold append [ ]

Second, observe that application of "fold" to the list constructors simply reconstructs the list, i.e.

id = fold (:) [ ]

Thus, we require that

reverse . fold append [ ] = fold (:) [ ]

To do so by universality, prove

1. *(reverse . fold append [ ]) = [ ]*
   iff (.)
   reverse (fold append [ ] [ ]) = [ ]

iff (fold.1)
reverse [ ] = [ ]
etc. by "reverse"
2. *(reverse . fold append [ ]) (X:Xs) = X : ((reverse . fold append [ ]) Xs)*
iff (.)
reverse (fold append [ ] (X:Xs)) = X : reverse (fold append [ ] Xs)
iff (fold.2)
reverse (append X (fold append [ ] Xs)) = X : reverse (fold append [ ] Xs)
iff (append)
reverse (fold append [ ] Xs ++ [X]) = X : reverse (fold append [ ] Xs)
which is true by the auxiliary result "reverse (Ys ++ [Y])" = "Y : reverse Ys"

Fusion property of fold. Another, which is simpler to use than the universal property when applicable, is the "fusion" property [5]:
H . fold G W = fold F V
or equivalently
H (fold G W Xs) = fold F V Xs
provided that
H W = V
and
H (G Y Ys) = F Y (H Ys)
Thus, to prove "reverse . reverse = id" as above, proceed first to
reverse . fold append [ ] = fold (:) [ ]
Then proceed by fusion, which proves the above equation provided that
1. *reverse [ ] = [ ]*
trivially
2. *reverse (append Y Ys) = Y : (reverse Ys)*
iff (expanding "append")
reverse (Ys ++ [Y]) = Y : (reverse Ys)
which is the auxiliary result assumed true earlier, and which is subject to proof via the fusion law.

Fusion demonstrably makes for simpler proofs, but furthermore its connection to "fold" makes it particularly useful in our new approach to functional programming.

3 Interpretation vs Definition in Programming

The relatively greater expressiveness of functional languages is used to differentiate between two styles of programming: "interpretational" vs "definitional". The former corresponds to typical imperative programming in which algorithmic and data components play complementary roles; the latter depends upon programmer-definable function-valued functions in order to represent data by functions. This "definitional" style is exactly the new approach we seek to elaborate. (Again, "Totally Functional Programming" (TFP) refers to this subset of functional programming in which *total* dependence upon functions exclusive of data to the greatest extent is the goal.)

The key components of the defintional/TFP style are of course the functions that replace data and thus the need to interpret these applicative behaviours from data. We call these “platonic combinators”, of two kinds “pure” and “impure”, conceived of as follows.

3.1 Pervasivess of interpretation of data

Consider the elementary “Boolean” datatype:

data Bool = True | False

Operations on booleans are programmable by cases:

not True = False
not False = True
and True x = x
and False x = False
etc.

This example, though simplest, is also signal: the various operations individually repeat the same essential role at each stage, of selecting one of two result paths depending upon the value of an operand True vs False. We contend that this exemplifies the essential characteristics of the “interpretational” style:

- a common essential operation is performed when processing a data type, in this case selection between two alternatives;
- the operation is not explicit, but is animated from the inert data by case analysis on the symbols/data type constructors;
- the common operation has to be replicated consistently across the different operations on the type.

The same appears to apply to other types. For example, consider the implementations of some basic operations on natural numbers:

add 0 n = n
add (m+1) n = (add m n) + 1
mul 0 n = 0
mul (m+1) n = add n (mul m n)

The characteristics of interpretation are present once again:

- an essential operation, this time of iteration, is applied (to some other operation that is in turn characteristic of the specific arithmetic operator – successor “+ 1” of 0 in the case of “add”, addition of “n” to 0 in the case of “mul”);
- the iteration is not explicit but is implicit in the (in this example, recursive) pattern of cases;
- the same general pattern of cases is used for each different specific operation (“add”, “mul”, etc.)

We use the term “interpretational” because this kind of animation of inert data to generate a computation is also characteristic of how an interpreter (for all kinds of programming language) induces a computation on the representation of a program. See further below (“Basis in language extensibility”) for more on the relationship with general programming language design and implementation.

3.2 “Platonic combinators” - definition without data

We define functions to represent the entities hitherto represented by symbolic data, and accordingly the functions on data are lifted to higher-order functions. We coin the phrase “platonic combinator” to refer to the functional representations of data, in that these functions (a.k.a. “combinators”) purport to embody the “essence” (following Plato) of data in terms of the inevitable characteristic applicative behaviour that is animated from the data.

The simplest of these functional representations unsurprisingly correspond to those invented for Church’s untyped lambda-calculus [6], where the unavailability of data necessitates such representations.

“Church booleans”. Rather than interpreting symbols “True” and “False”, the computation inherent in Boolean values may be defined directly in the following terms:

```
true = \ x y -> x
false = \ x y -> y
```

That is, either Boolean value represents one of the possible choices form among two operands, and to make the choice is simply to apply the Boolean. Thus, instead of

if C then E1 else E2

which interpretationally tests C for equality to the symbols “True” or “False”, we can write simply the direct application

C E1 E2

Higher Boolean operations are redefinable accordingly:

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

To summarise, the purpose of type Boolean is to make a choice, which these functional representations for truth values directly express.

“Church numerals”. Similarly, rather than interpreting symbols “0”, “+ 1” to induce the iteration inherent in natural numbers, instead directly define the naturals as iterators:

```
zero= \ f x -> x
succ n = \ f x -> f (n f x)
one = succ zero
etc…
```

so that for any function F and any X

```
succ zero F X = F X
succ one F X = F (F X)
etc.
```

Arithmetic operations are redefinable in the direct definitional style e.g. as

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

To summarise, the purpose of type Natural is to iterate, which these functional representations for truth values directly express. Any other behaviour for Naturals represents a misuse, for which another type should be used. (NB this corresponds with good programming practice e.g. enumerated types should be used to represent types such as error codes, colurs, etc., rather than misleading numerical representations.)

3.3 Pure vs impure platonic combinators

The platonic combinators themselves can be distinguished on the basis of whether they are "pure" i.e. eschew, or are "impure" i.e. necessarily retain, some degree of interpretation. The above examples of Church booleans and numerals are evidently pure (though their operands may be interpretational).
As an example of a necessarily-impure platonic combinator consider the representation of sets by characteristic predicates, constructed and manipulated by operations as follows:

```
emptyP = \ y -> False
singleP x = \ y -> x==y
unionP s1 s2 =\ y -> memberP s1 y or memberP s2 y
memberP s y = s y
```

The "memberP" is of course superfluous, the whole point of platonic combinators being that they are directly applicable in their essential behaviour as functions.

Characteristic predicates are platonic combinators because they represent sets as their essential computations, i.e. the essence of a set is the membership test, which the characteristic predicate implements. They are impure, because the characteristic predicate for a singleton set "single e" involves interpretation: testing that the element 'e' is equal to the putative element 'x'; and because equality testing is ultimately definable only in terms of symbols or symbolic structures. That is, sets (as defined here in terms of empty, singleton and union of sets) only have meaning in terms of a domain of elements that is interpretational. As such, sets serve as a model for all kinds of structure where, while the interpretational nature of the elements enjoys no improvement, the structure itself can be improved to become definitional.

Superificially, there is little apparent difference between pure platonic combinators (PPCs) and impure platonic combinators (IPCs), but formal derivations of the latter are somewhat more involved than the former.

3.4 Practicality of platonic combinators

Examples of platonic combinators, especially of the impure variety, abound. Indeed, we speculate that much of the appeal of impressive examples of higher-order functional programming derives from how they replace symbolic data with functional representations, exactly in as platonic combinators. Consider the following examples.

Combinator parsers. 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 [7], 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 [8] seems to provide powerful independent support for TFP.

Exact real arithmetic. Boehm & Cartwright [9] 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.

Programmed graph reduction. Just as combinator parsers replace a static structure (the grammar) and its interpreter (parsing engine) with a single applicative entity, so does programmed graph reduction of lambda-expressions [14]. It would seem that this is a further instance of the platonic combinator idea, perhaps requiring further investigation of the links in detail.

4 “Fold” and Pure Platonic Combinators

PPCs emerge exactly as partial applications of folds to data, which enables very simple formal derivations.

4.1 Generalising “fold”

Analogies to “fold” exist for all “regular” datatypes [10], as do corresponding analogies for laws such as fusion. Consider an ADT T with constructors C1 … Cn where each Ci has arity m and the jth operand is of type Tj (without loss of generality, this also applies for polymorphic T):

data T = … | Ci Ti1 … Tm | …

Generally then, the definition of foldT consists of a set of equations, one for each different pattern of construction (i.e. depending on each different constructor Ci) in T:

foldT c1 … cn (Ci a1 … am) = ci A1 … Am

where

- each equation includes formal parameters ci corresponding to constructors Ci
- the body of the equation for Ci applies corresponding parameter ci to operands Aj, each in turn corresponding to the operands of Ci
- each operand Aj of ci is derived from operands aj of the prevailing Ci: if aj corresponds to a nested element of T, then Aj is aj “folded”, i.e. of the form “foldT c1 … cn aj”; otherwise Aj is just aj

This pattern can be observed for “fold” as defined on lists above. Other simple folds are for “Cons”-pairs:

foldP m (Cons a b) = m a b

for Booleans:

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

(formal parameters named ‘t’ and ‘f’ are suggestive of their function i.e. of what is to be returned when the other operand is either “True” or “False”) and for Naturals, with constructors 0, and successor denoted (+1):

foldN s z 0 = z
foldN s z (m+1) =s (foldN s z m)

(formal parameters named ‘s’ and ‘z’ are suggestive of their function i.e. of what to do when the other operand is either an application of the successor function or zero).

Corresponding generalisations of the fusion (and thus universality law) suggest themselves, but we will defer treatment to their context in terms of PPCs below.

4.2 From folds to PPCs

As generalised above, it emerges that folds are somewhat more than packaged recursion: they are a means of representing data structures that is abstract in terms of constructors, as effected by the general schema above. The “recursion packaging” capability exists precisely because the constructors are replaced by the operands of the fold.

Things become clearer in this regard if we invert the order of the parameters to fold in order to place the data (structure) over which the fold operates first. Thus the schema above becomes

ifoldT (Ci a1 … ain) c1 … cn = ci A1 … Ain

with applications of fold to ak inside Ai rewritten accordingly. For example

ifold [ ] op b = [ ]
ifold (x:xs) op b = op x (ifold xs op b)

It may be observed now that our PPCs (Church booleans and naturals) are simply the results of the partial application of these inverted folds to the corresponding data:

ifoldB True = \t f -> t = true
ifoldB False = \t f -> f = false

ifoldN 0 = \s z -> z = \f x -> x = zero
ifoldN (m+1)
= \ s z -> s (ifoldN m s z)
= …
= \s z -> s $^{m+1}$ *z*
= \f x -> f $^{m+1}$*x*
= succ^{m+1} *zero*

Likewise, we can synthesise what we might call “Church lists”:

ifold [ ] = \op b -> b
ifold [X1…Xn]
= \op b -> op X1 (ifold [X2:…:Xn] op b)
= …
= \op b -> op X1 (op X2 … (op Xn b)…)

In this example, note how Church lists are the computational embodiment of the essence of lists: the order of operands to “op” embodies the ordering of elements in a list, and the role of ‘b’ signifies the list’s finiteness.

It’s thus clear as claimed above that PPCs are data structures from which the constructors have been abstracted. The relationships between “ifoldT” for some type T, PPCs corresponding to T, data D of type T and the constructors C1 … Cn that generate values of T, can be summarised as follows:

ifoldT D = PPC

PPC C1 … Cn = D

A fixed-point property is more apparent when presented in terms the "original" fold:

foldT C1 … Cn D = D

4.3 Generating PPCs from specifications

The above generation of PPCs by folding over data (structures) is of course satisfactory to a limited degree – to achieve totally functional programming's goal of "datalessness", it's essential to generate PPCs directly. In other words, how to synthesise the functions creating PPCs (e.g. "succ" on Church numerals), and even the "atomic" PPCs (e.g. "zero"), directly from specifications?

Consider the difference between the generation of PPCs, e.g. of Church numerals by "succ" and "zero", and the corresponding partial applications of "ifoldN":

zero = \ f x -> x

succ n = \ f x -> f (n f x)

vs

ifoldN 0 = \s z -> z

ifoldN (M+1)= \ s z -> s (ifoldN M s z)

The differences with PPCs are that

- the "fold" and the data constructor are absorbed together into the PPC generator
- the PPC generator is directly applicable to the data to which the constructor was applicable
- PPCs themselves (e.g. as represented by parameter 'n' to "succ") are applicable to the remaining operand of the (inverted) fold.

Consequently, the derivation of PPCs, or more specifically the derivation of generators that yield PPCs, mirrors the derivation of the fold-function for the ADT but accounting for these differences. Thus in general, consider again the above ADT T with constructors C1 … Cn where each Ci has arity m and the jth operand is of type Tj:

data T = … | Ci Ti1 … Tm | …

Thus, the generator functions Gi of PPCs for T are defined by a set of equations, one for each different pattern of construction (i.e. depending on each different constructor Ci) in T:

Gi a1… am = \c1 … cn -> ci A1 … Am

where :

- the construction pattern "(Ci a1 … am)" formerly present in the definition for the various folds is replaced by direct citation of the aj as formal parameters to Gi, since the discriminating role of Ci is discharged through its absorption with the folds to form distinct PPC generators Gi;
- each equation still includes formal parameters ci corresponding to constructors Ci
- as before, the body of the equation for Gi (corresponding to constructor Ci) applies corresponding parameter ci to operands Aj, each in turn corresponding

to the operands of Ci

- as before, each operand Aj of ci is derived from operands aj of the prevailing Ci but slightly differently in order to account for the absorption of “fold” into the PPC at the same time as the aj are PPCs, not data (structures): if aj corresponds to a nested element of T, then Aj is aj “folded”, but now of the form “aj ci … cn”; otherwise Aj is just aj

As a further example, consider binary trees:

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

“Church tree” generators are therefore

null = \n l b -> n
leaf e = \n l b -> l e
branch b1 b2 = \n l b -> b (b1 n l b) (b2 n l b)

Continuing/elaborating the summary at the end of the last section, expressing data D as a construction of constructor Ci of arity m applied to operands Xj, and introducing Gi as the PPC generator corresponding to the absorption of “ifoldT” and Ci, gives:

ifoldT (Ci X1 … Xm) = PPC
Gi X1 … Xm = PPC

There is a further property:

PPC G1 … Gn = PPC

which can be expressed as a fixed-point

(\p -> p G1 … Gn) PPC = PPC

and which can be expressed “old style” as

ifoldT D G1 … Gn = ifoldT D = PPC

4.4 Generalised fusion for PPCs

A fusion law for PPCs would seem to be able to be generalised and adapted from that presented for list fold above. In general, for the usual ADT T

data T = … | Ci Ti1 … Tm | …

we envisage that (for any applicable Gi, Fi, not necessarily restricted to the specific Gi from which the PPC was generated)

H (PPC G1 … Gn) = PPC F1 … Fn

provided that

H (Gi A1… Am) = Fi A1’ … Am’

where

Aj’ =H Aj when the jth operand of Ci/Gi is a nested occurrence of T
Aj’ = Aj otherwise

For example, the fusion law for “Church trees” CT above would be

H (CT G1 G2 G3) = CT F1 F2 F3

provided that

H G1 = F1
H (G2 X) = F2 X
H (G3 X1 X2) = F3 (H X1) (H X2)

An example application of generalised fusion is presented in the context of IPCs further below.

5 Formal Derivation of Impure Platonic Combinators

While derivation of PPCs from folds follows the derivation of folds from ADT signature in a straightforward fashion, derivation of the IPCs that account for other ADT operations is a little more challenging. However, as with PPCs, "fold" functions play a key role.

Recall that the idea of an IPC (compared to a PPC) is that the IPC corresponds to a function that extracts data from a structure and processes it interpretively, whereas a PPC (from the above laws) is a partially-applied fold that further applies to contructors/generators. Application of a PPC to the generators regenerates the PPC; however application of the PPC to other functions yields different behaviours. "Normally" these behaviours are instances of the essential computation for the PPC, e.g. applying a church numeral N to some F and some X executes N-fold composition of F over X. However, with care the operands to a PPC (derived from an ADT signature) can be chosen to generate the IPC corresponding to the ADT signature along with the behavioural specification for the extractor function that the IPC models.

The essence of our technique is as follows:

1. IPCs can result from applying PPCs to IPC generators
2. by definition, an IPC is the function resulting from (partial) application of an ADT extractor to the data structure (e.g. the characteristic predicate for a set is the partial application of "member" to the set)
3. 1. above can be expressed as a fold
4. 2. above can be expressed as an operation on a fold
5. 3. and 4. above can be related by fusion, allowing the IPC generators to be solved for.

This technique is exemplified below by the "set" ADT for which the IPC was exposed above.

5.1 "Set" ADT specification

The algebraic specification for this version of sets is as follows.

```
Empty :: Set t
Single :: t -> Set t
Union :: Set t -> Set t -> Set t

member :: Set t -> t -> Bool
member Empty y = False
member (Single x) y = x==y
member (Union s1 s2) y = member s1 y or member s2 y
```

The PPC generators for this ADT are thus

```
empty = \e s u -> e
single x = \e s u -> s x
union s1 s2 = \e s u -> u (foldS s1 e s u) (foldS s2 e s u)
```

Naturally, the PPC ignores "member", which will be accounted for in the IPC to follow. In view of the isomorphism between Sets and Church trees above, isomorphism of PPCs follows. Likewise, the above fusion law for Church trees also holds for Sets.

5.2 "Set" IPC derivation

Notation:

- ADT constructors have an initial capital, the corresponding IPC generators are all in lower case but with a capital 'I' suffix
- other ADT operations are all in lower case, the corresponding IPC operations likewise but with a capital 'I' suffix
- variables denoting members of the ADT are all in lower case, the corresponding IPC/PPC instances likewise but with a capital 'I'/'P' suffix.

We require that

- the IPC for a set is the set's characteristic predicate; in terms of the above specification:
  sI = member s
- sI be derived from the PPC corresponding to 's' by application to IPC generators:
  sI = sP emptyI singleI unionI
- that is,
  member s = sP emptyI singleI unionI

Now, recall a set 's' can be expressed as the inverted fold for Sets "ifoldS" applied to 's' then applied to the set constructors (just as a PPC applied to generators regenerates the PPC). At the same time, PPC "sP" is similarly "ifoldS" applied to "s", by definition of PPCs.

Thus rewriting each side of the above:

member ((ifoldS s) Empty Single Union) = (ifoldS s) emptyI singleI unionI

Now, according to the fusion law for Sets, this equation holds provided that

1. *emptyI = member Empty*
2. *singleI x = member (Single x)*
3. *unionI (member s1) (member s2) = member (Union* s1 s2)

In each case, we derive

1. *emptyI = member Empty = \y -> False*
2. *singleI x = member (Single x) = \y -> x==y*

(recalling that member si = siI)

```
3. unionI s1I s2I    = member (Union s1 s2)
                     = \y -> member s1 y or member s2 y
                     = \y -> (s1I y) or  (s2I y)
```

6 Proving with Platonic Combinators

We show how the platonic combinator representation can simplify proofs as well as derivations above.

6.1 Exploiting PPCs as folds

To prove (associativity of +):

add A (add B C) = add (add A B) C

iff (expanding all occurrences of “add”)

A succ (B succ C) = (A succ B) succ C

i.e.

A succ (B succ C) = (\x -> x succ C) (A succ B)

Now the fusion law for Church numerals applies:

N F’ X’ = H (N F X)

provided that

H X = X’
H (F N) = F’ (H N)

Thus, we proceed

1. *(\x -> x succ C) B = B succ C*
   trivially
2. *(\x -> x succ C) (succ A) = succ ((\x -> x succ C) A)*
   iff (expanding anonymous function applications)
   ((succ A) succ C) = succ (A succ C)
   iff (expanding “succ n = \f x -> f (n f x)”)
   succ (A succ C) = succ (A succ C)

To conclude, when all data is represented by the results of folds, then some operations on data are more apparently subject to analysis by fold laws.

6.2 Exploiting specifics of platonic combinators

Finally, specific properties of representations of platonic combinators or of operations can be exploited.

For example, further simplification of the above associativity of “add” is possible if an alternate representations for the operation is used. The equivalent definition of “add” (which can be derived by fusion)

add m n = \f x -> m f (n f x)

results in the trivialisation of the above proof:

add A (add B C)
= expanding “add”
\f x -> A f (add B C f x)
= expanding “add”
\f x -> A f (B f (C f x))
= recognising “add”
\f x -> add A B f (C f x)
= recognising “add”
add (add A B) C

For an IPC example, consider how trivially we may prove that set union is associative.

unionI A (unionI B C)
= expand union

\y -> A y or (B y or C y)
= associativity of ‘or’ - trivial
\y -> (A y or B y) or C y
= recognise union
unionI (unionI A B) C)

7 Related Technical Issues

While the focus of this paper is on the relationship between formal methods and TFP, a number of other issues are of interest, either of necessity for practical development of this field (i.e. types for TFP) or because they indicate the greater potential of TFP in a wider context of computer science and related fields.

7.1 Types for TFP

The inadequacy of Hindley-Milner-based typing [11] (as implemented in current functional languages, e.g. Haskell) for TFP is apparent: for example, reconsider operations on Church numerals

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

The complementary rendition of exponentiation is

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

which however leads to a type error: given further definitions

one = succ zero
two = succ one

then for the application

exp one two

the HUGS implementation of Haskell gives the daunting response

```
ERROR: Type error in application
*** Expression     : exp one two
*** Term           : one
*** Type           : ((d -> e -> e) -> (((a -> b) -> c -> a) ->
(a -> b) -> c -> b) -> ((a -> b) -> c -> a) -> (a -> b) -> c ->
b) -> (d -> e -> e) -> (((a -> b) -> c -> a) -> (a -> b) -> c ->
b) -> ((a -> b) -> c -> a) -> (a -> b) -> c -> b
*** Does not match : (((a -> b) -> c -> a) -> (a -> b) -> c ->
b) -> (d -> e -> e) -> (((a -> b) -> c -> a) -> (a -> b) -> c ->
b) -> ((a -> b) -> c -> a) -> (a -> b) -> c -> b
*** Because        : unification would give infinite type
```

Other “surprises” are only to be expected as more complex platonic combinators are discovered in the course of development of TFP.

This situation can be recovered from in different ways. First, we can re-express our definitions. For example, redefining either as

exp m n = n m

or

mul m n = m . n

will allow “exp one two” to type-check. The new “mul” and “exp” can be derived (in turn) from the originals, but it seems doubtful that programmers could reasonably be expected to perform these derivations (but there is scope for this to be automated).

Second (and arguably preferably in that it lets us express what we want!), is to adopt a more powerful type system. Second-order polymorphic typed-lambda-calculus [12] 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. A compromise by which polymorphic values are represented as datatype components [13] allows for a combination of greater type-expressiveness and effective type inference. However, this representation conflicts with the anti-interpretational goal of TFP. Clearly, more research is needed before a pure outcome for TFP is available.

7.2 Efficiency?

While the goal of TFP is to provide simplicity for programmers, it appears there may be some opportunity to observe performance *improvement*. This is despite TFP’s extreme dependence upon functions and higher-order functions in particular, which are the essential source of the celebrated inefficiency of functional languages [14]. In particular, some of the functional representations appear extraordinarily inefficient, e.g. Church numerals are in effect a unary representation.

However, definitional TFP has an advantage over interpretational programming (functional or imperative) in that the costly process of interpreting/animating inert data into a computation is avoided. Moreover, the inevitable functional representations for various platonic combinators reflect the computation that will take place. For example, natural numbers give rise to a sequence of operations as the inherent iteration executes, which is essentially a unary representation of the number. In other words, unary representation is not always inconvenient, especially in a pure TFP setting.

Of course, when integrating TFP in impure contexts, it may be necessary to provide efficient built-in representations for key platonic combinators, such as naturals. We envisage possibly that an implementation could represent them in traditional binary form (and basic operations accordingly), but this representation could be recognised by the implementation as an iteration operator as well.

7.3 Integration with OOP?

By their nature, pure/impure platonic combinators can be thought of as objects with just one method (other than generators) – the PPC/IPC. This varies with universal object-oriented practice. However, our restriction is not necessarily nonsense: surely a well-designed component (function) should have a single cohesive behaviour, so why should multiple behaviours be permitted?

Of course, we entertain an apparent counter-example of our own policy: different kinds of IPC can be generated from the one PPC by supplying different seeds, so we do seem to permit multiple behaviours. However, it is the relationship of these behaviours though a common "mother behaviour" in the PPC that makes our approach different and supports our claim of "one object, one method". It will however be necessary to demonstrate a "mother" for apparently different operations on the one class. We are currently working on a demonstration that the multiple behaviours that can be associated with a context-free grammar (recognise, parse, unparse) all stem from a common platonic combinator.

7.4 Subrecursive programming and TFP?

A by-product of eschewing interpretation is that the full expressive power needed to write interpreters, and associated complications (i.e. possibility of nontermination), is not necessary. While a feature of TFP is that it integrates smoothly into contexts where at least a residue of essential data/interpretation prevails, at least some TFP components would be able to be written in a subrecursive functional language subset.

It's surprising, but questionable, as to why Turing-completeness, and the language constructs the enable it such as recursion, should be an essential desideratum for programming language expressiveness. Without it, it becomes impossible to write an interpreter for a universal Turing machine, or any other universal interpreter, but what else that is moreover pragmatically useful is unavailable? Indeed, within second-order polymorphic typed lambda-calculus which lacks recursion, it is possible to express any function that is provably terminating in second-order arithmetic. An implication of this is that Ackerman's function is expressible! Is the only point of having the expressive power of a Turing Machine so that a Universal Turing Machine can be programmed? Our TFP can be thought of as an hybrid version of second-order polymorphic typed lambda-calculus, hence the "total" in TFP also refers to this subrecursive component.

Besides, there has been a long history of other research-cum-speculation about the theory and pragmatics of "subrecursive" programming [15]. Of most apparent relevance to TFP is Turner's proposal [16] for "elementary strong functional programming", which restricts functional programming to total functions.

Turner's approach has the following salient points:

- the simplicity of equational reasoning for functional programs is not quite as attractive as promoted, in view of the need to handle nonterminating computations;
- type-theoretic approaches to the problem (e.g. constructive type theory) have poor pragmatics;
- essentially syntactic restrictions on a pragmatic functional language give a computationally-equivalent result;
- even operating systems can be programmed in this style;
- in the context of all the above, the inability of the language to program its own interpreter is no loss, especially as compilers can still be programmed;
- the appearance at least of data is retained (in contradistinction to TFP).

We might also recall that Backus' seminal Turing Award paper [17] promoted a language that eschewed programmer-defined recursion/iteration, relying instead upon a fixed set of specific iterators (including a version of fold).

The import of these observations is that the subrecursive aspect of TFP, far from being an eccentricity, locates it in a significant stream of programming language development. Likewise, TFP's links to the mainstream of functional programming may offer the Subrecursion theme fulfilment.

7.5 Basis in language extensibility

The origins of this project lie in the vision of projecting the idea of language extensibility into mainstream software development. It's demonstrable that programming is a language design/extension activity: pragmatically, standard criteria for program quality assessment parallel those for assessing language designs [18]; formally, a straightforward reordering of the parameters to the denotational semantic meaning function exposes declarations as explicit language-extending constructs [19].

At the same time however, language design/extension is a programming activity, and language extension should eschew undesirable programming practices. We consider that recourse to writing an interpreter (anew, or by modification to the host language's existing interpreter), whereby the syntax identifying some new semantic entity is provided with those semantics through a comprehensive translation of the extended language into the original base language, is the hallmark of bad language extension practice:

- writing of interpreters requires particular skills;
- there is no guarantee that the semantics of the base language will be preserved in the extension, unless proven so;
- in all, interpretation is a complex undertaking which does little to simplify things.

The complexity of interpretation can also be thought of in terms of the impossibility of providing a simple, localized translation of the extension's terms into base language semantics. Far simpler and thus preferable is direct definition, whereby the association of new syntax with new semantics is provided by a local replacement rule, such as macro expansion or identifier declaration. Thus, language extension should be carried out on an expressively-complete [20] base language, in which all conceivable semantic entities can be expressed/defined directly without interpretation of symbolic representations.

Reverting to programming as language extension, it follows that undesirable language extension practices, especially interpretation, should be eliminated or at least minimised in programming. How then is interpretation manifested in "normal programming", and how may it be avoided? We contend that the need to animate inert symbolic data by a combination of branching and iterative processing is what mirrors the complexities of interpretation. Programming could be greatly simplified if data could be replaced by the functions that these interpretations eventually achieve, which proposition leads directly into TFP as presented above.

7.6 Beyond programming?

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

Canonical software design representation. One of the drawbacks of Interpretational 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 [21], 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. The more a software system depends upon data for its behaviour, the less the program structure indicates what's really happening. 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.

Analog systems specifications and prototyping. There appears to be an interesting connection between analog computing, where computations are composed from physical components whose behaviours model the domains being computed with, and TFP where data are represented by (the behaviours of) platonic combinators. Indeed, the existence of TFP suggests that the hitherto one-dimensional division of computation into the poles Analog vs. Digital should be replaced by a two-dimensional structure: (Interpretational vs Definitional, Discrete/Symbolic vs Continuous). Conventional "digital" computing is identified by the (Interpretational, Discrete/Symbolic) point, analog by (Definitional, Continuous), and TFP by (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/prototyping language for analog systems: functionality can be developed in the relatively relaxed Discrete/Symbolic domain before being built in the exacting electronic manifestation of the Continuous domain.

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.

8 Conclusions

We have covered related issues as follows.

First, we provided the reader with an introduction to "Totally Functional Programming", and referred to its varied roots in language extensibility, formal methods and type theory. TFP is an approach to functional programming that eschews data (inert symbolic representations) for functions (dynamic applicative representations) that empody what we hypothesise to be the distinctive applicative behaviour inherent in every datum.

Second, and as the essence of this paper, we have shown how TFP is compatible with advanced formal methods for functional programming, in particular formal derivation of platonic combinators using laws about "fold", as well as examples of how reasoning is sometimes simpler in "dataless" TFP than in usual "dataful" functional programming.

Finally, we indicated some aspects of the research agenda to be followed in order to make TFP practicable.

9 Acknowledgements

We are grateful to our numerous students/colleagues whom have contributed to the work reported above over the years and to its presentation in this form, notably Trevor Chorvat, Simeon Cran, Ming Gong, Ian Peake and Sean Seefried.

References

1. 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, Springer, Berlin (1985) 1-16
2. Hughes, J., "Why Functional Programming Matters", The Computer Journal, vol. 32, no. 2 (1989) 98-107
3. Bird, R., "Introduction to Functional Programming", Prentice-Hall (2000)
4. http://www.haskell.org
5. Hutton, G., "A Tutorial on the Universality and Expressiveness of Fold", Journal of Functional Programming", vol. 9, no. 4 (1999) 355-372
6. Barendregt, H.P., "The Lambda Calculus - Its Syntax and Semantics", North-Holland, Amsterdam (1984)
7. Aho, A.V. and Ullman, J.D., "The Theory of Parsing, Translation and Compiling", Prentice-Hall (1972)
8. Hutton, G., "Parsing Using Combinators", Proc. Glasgow Workshop on Functional Programming, Springer (1989)
9. 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) 43-64
10. Sheard, T. and Fougaras, L., "A fold for all seasons", Proc. ACM Conference on Functional Programming and Computer Architecture, Springer (1993)
11. Milner, R., "A Theory of Type Polymorphism in Programming", J. Comp. Syst. Scs., vol. 17 (1977) 348-375
12. Reynolds, J.C. "Three approaches to type structure", in Mathematical Foundations of Software Development, LNCS Vol 185, Springer-Verlag (1985)
13. Jones, Mark P., "First-class Polymorphism with Type Inference", Proc. 24th ACM Symposium on Principles of Programming Languages (1997)
14. Peyton Jones, S., "The Implementation of Functional Programming Languages", Prentice-

Hall International, Hemel Hempstead (1987)
15. Royer, J.S., & J.Case, J., “Subrecursive Programming Systems: Complexity & Succintness”, Birkhauser (1994)
16. Turner, D.A., “Elementary Strong Functional Programming”, Proceedings of the first international symposium on Functional Programming Languages in Education, Springer LNCS vol. 1022 (1995) 1-13
17. Backus, J., “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs”, Comm. ACM, vol. 21, no. 8 (1978) 613-641
18. 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) 14-18
19. Bailes, P.A., Chorvat, T. and Peake, I., “A Formal Basis for the Perception of Programming as a Language Design Activity”, Proc. 1994 International Conference on Computing and Information, Peterborough (1994)
20. Plotkin, G.D., “PCF Considered as a Programming Language”, Theoretical Computer Science, vol. 5 (1977) 223-255
21. Chikofsky, E. and Cross, J.H.II, “Reverse engineering and design recovery: a taxonomy”, IEEE Software, January (1990) 13-17