The Semantic Elegance of Applicative Languages

D. A. Turner

University of Kent at Canterbury

In what does the alleged superiority of applicative languages consist? In the last analysis the answer must be in terms of the reduction in the time required to produce a correct program to solve a given problem. On reflection I decided that the best way to demonstrate this would be to take some reasonably non-trivial problem and show how, by proceeding within a certain kind of applicative language framework it was possible to develop a working solution with a fraction of the effort that would have been necessary in a conventional imperative language. The particular problem I have chosen also brings out a number of general points of interest which I shall discuss briefly afterwards.

Before proceeding it will be necessary for me to quickly outline the language framework within which we shall be working. Very briefly it can be summarised as (non-strict, higher order) recursion equations + set abstraction. Obviously what matters are the underlying semantic concepts, not the particular syntax that is used to express them, but for the sake of definiteness I shall use the notation of KRC (= "Kent Recursive Calculator"), an applicative programming system implemented at the University of Kent [Turner 81]. KRC is fairly closely based on the earlier language SASL, [Turner 76], but I have added a facility for set abstraction.

An overview of KRC

A KRC program is a collection of equations by means of which the user attaches names to various objects in KRC's universe of discourse. The universe of discourse contains four types of object. There are two basic data types, numbers and strings; and two types of structured object - lists and functions. Numbers and strings have the sort of properties one would expect. Lists are denoted by using square brackets and commas - thus `[1, 2, 3, 4]` and the empty list is written `[]`.

Lists may be accessed by indexing, so if L is the name of a list `L 3` denotes its third component. An important operator on lists is ":" which joins a new member at the front. So

```
0: [1, 2, 3]
```

has the value `[0, 1, 2, 3]`. The elements of a list may be of any type and need not all be of the same type. Lists may be finite or infinite, for example the equation

```
x = 1:x
```

defines x to be a name for the infinite list `[1, 1, 1 ...]`.

The universe of discourse contains all possible infinite lists (actually not all of them can be denoted by writing down equations, but for reasons of continuity in the semantics we have to say they are all there). A useful piece of shorthand is the ".." notation, thus

```
[a..b]
```

denotes the list of numbers (inclusive) from a to b. This also has an open-ended form, so for example

```
[1..]
```

denotes the list of all natural numbers.

Functions are denoted by writing down one or more equations, with the name of the function followed by some formal parameters on the left and an expression giving a value for the function on the right. So for example functions for squaring a number and for calculating factorials would be written respectively ("product" is a library function).

```
sq n = n * n
fac n = product [1..n]
```

A function can be defined by more than one equation, with the different cases being distinguished by the use of pattern matching in the formal parameters and/or the use of guards (boolean expressions, written on the right of an equation following a comma). Ackermann's function for example, could be defined by the following three equations

```
A 0 n = n + 1
A m 0 = A (m-1) 1, m>0
A m n = A (m-1)(A m (n-1)), m>0&n>0
```

Notice that the order in which the above equations are written has no logical significance (we insert guards where necessary to ensure this). As another example of the use of pattern matching, the library function "product" can be defined

```
product [] = 1
product (a:x) = a * product x
```

Note that the use of patterns involving ":" on the left of an equation avoids the need for explicit use of the selectors "hd" and "tl" on lists.

Functions can be non-strict if the equations imply that they should be. For example if we define f by

```
f x = 3
```

then of course f must always return the result 3, even if its argument is represented by a non-terminating computation. [We do not discuss implementations at all here, but as the reader will have already deduced from the presence of infinite lists, it must involve some form of lazy evaluation.]

Set Abstraction

We use a language construct closely based on Zermelo-Frankel set abstraction (except that the result is here a list rather than a set). An (informal) syntax for these expressions may be given as follows

```
zfexpression ::= {exp|qualifier; ...
                                ...; qualifier}
qualifier ::= generator|guard
generator ::= var ← list-expression
guard     ::= boolean-expression
```

The variable on the left of a generator is a local variable of the zf expression and ranges over the members of the list on the right of the "←" sign.

One example is the following definition of the (2nd order) library function "map" which applies a function to every element of a list

```
map f x = {f a | a ← x}
```

The library "filter", which filters a list through a given predicate, could be defined

```
filter f x = {a | a ← x; f a}
```

The use of the construct is particularly convenient when there is more than one generator involved, for example a function for generating the cartesian product of two lists (i.e. a list of all pairs formed by drawing one from each) can be written

```
cp x y = {[a,b]|a←x;b←y}
```

The addition of set abstraction represents a very considerable increase in the expressive power of an applicative language. Consider for example the problem of defining a function which will return a list of all possible partitions of a number into positive integers. For the sake of definiteness, let us say with permutations, so [2,1] and [1,2] are two different partitions of 3. Such a function can be expressed rather succinctly as follows

```
partitions 0 = [[]]
partitions n = {i:p | i← [1..n];
                      p← partitions(n-i)}
```

Notice by the way that the variables introduced by generators come into scope from left to right, so later generators can involve earlier ones (but not vice versa).

Following this extremely brief overview of the notation of KRC, we now turn our attention to a somewhat larger problem, as promised.

The statement of the problem

The problem which I have chosen to use as an illustration here arises from the field of organic chemistry. I dare say that from the point of view of a chemist who has made a study of these matters it will seem a rather easy problem and perhaps somebody, somewhere has a FORTRAN program sitting on their desk which already solves it. Nevertheless from the point of view of a programmer who had not tried it before, the problem seemed difficult enough to be interesting. At least several competent programmers I know reported to me that they had found it so. It therefore seemed a good testing ground for the language framework outlined in the opening section of this paper. In what follows I have concentrated on reproducing fairly honestly the thought processes which first led me to a runnable solution, rather than on presenting the most elegant or efficient solution possible.

The problem is to enumerate, without repetitions and in order of increasing size, all possible paraffin molecules. For those who have forgotten their high school chemistry, paraffins are built up using only carbon, which has a valency of 4 and hydrogen, which has a valency of 1. The first few paraffin molecules, together with their names are shown in figure 1.

In a paraffin one is allowed neither double bonds nor cycles, so all paraffins with n carbon atoms share the empirical formula

$$C_n H_{2n+2}$$

but for all $n \geq 4$ there are several distinct molecules ("isomers") with the same formula. The number of isomers rises rather rapidly with n. In counting isomers it should be borne in mind that the four bond positions on a given carbon atom are in fact indistinguishable (i.e. the four bond positions can be freely interchanged). So what seem at first to be different molecules may in fact turn out to be different orientations of the same molecule. Notice in particular that the phenomenon of "stereo-isomerism" (a molecule being different from its mirror image) is not possible with paraffins.

The problem as posed is not merely to count the number of isomers for each n, but actually to produce a representation (just one) of each molecule.

Choosing a Representation for the Data Type "Paraffin Molecule"

Let us begin by choosing a representation for the molecules. Normally in top-down programming one is advised to delay decisions about representation as long as possible, but in this case it is clear that a central problem is going to be defining an equivalence relation on molecules, and I found it difficult to bring this into focus without fixing on an (at least provisional) representation for molecules.

Ideally we would like to define a canonical orientation for each molecule so that each distinct isomer is represented by one and only one data structure. There does not seem to be any straightforward way of defining such a canonical orientation, however, (at least I could not see one) so we shall go for the alternative plan of giving a non-unique representation together with a set of laws for determining which representations are equivalent (in the sense that they are different orientations of the same molecule).

One obvious way to represent a paraffin molecule in terms of list structures is as follows. We pick on one carbon atom arbitrarily and deem it to be the "leading" one. We then represent the molecule as a 4-list, the components of the list being the sub-molecules ("radicals" in chemical terminology) attached to the 4 bonds of the leading carbon atom. Each radical is represented either by the string "H", if it is just a hydrogen atom, or else by a 3-list corresponding to a carbon atom with 3 further radicals attached to it.

In terms of these conventions, then, methane would be represented thus -

```
["H", "H", "H", "H"]
```

and one of several possible representations for propane is

```
[["H","H","H"],["H","H","H"],"H","H"]
```

We could obviously save a great deal of space by not representing hydrogen atoms explicitly, but following a principle of separation of concerns let us leave that as an optimisation to be carried out later.

Defining an Equivalence Relation on Molecules

The above representation for a molecule has two elements of arbitrariness - first that any carbon atom could have been picked on as the leading one - second that for each carbon atom the order in which its dependent radicals are listed is arbitrary and could be freely permuted.

We capture this below in terms of three transformations which we can carry out on the representation of a molecule - "invert", "rotate" and "swap". Each transformation is expressed here by a function which can be applied to a representation of a molecule to return a (perhaps

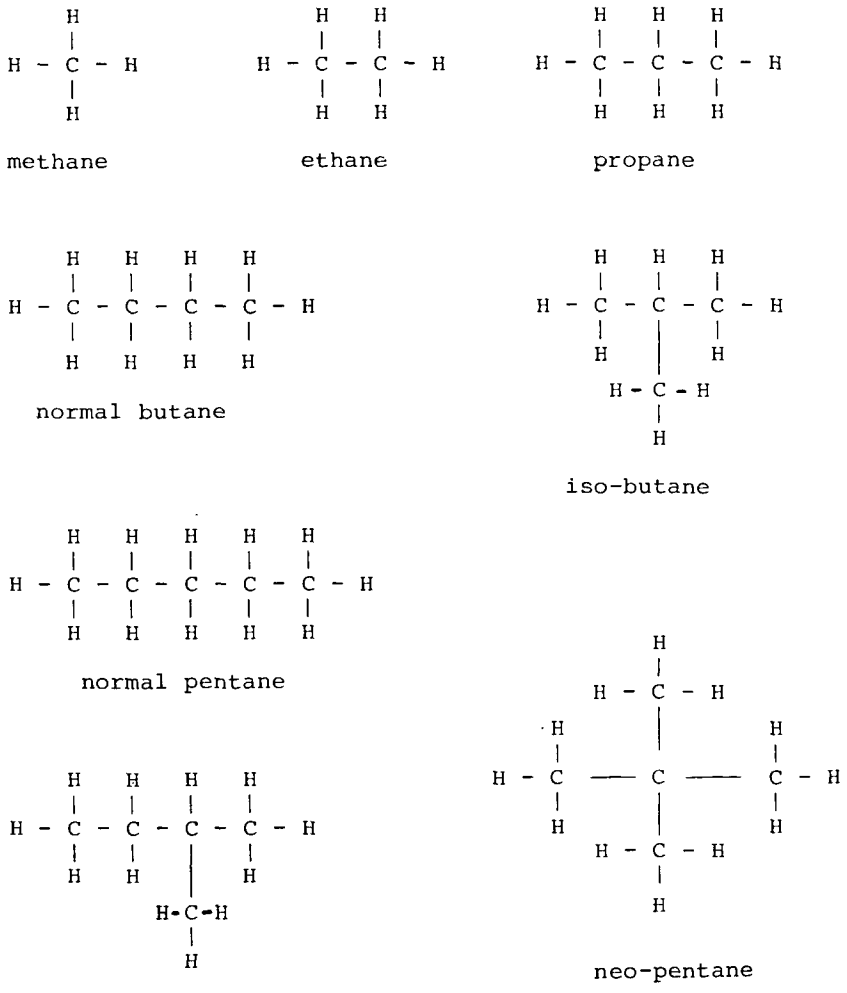

Figure 1 "Some Paraffin Molecules"

modified) representation of the same molecule.

```
invert [[a, b, c] d, e, f]
        = [a, b, c [d, e, f]]

invert x = x, x 1 = "H"

rotate [a, b, c, d] = [b, c, d, a]

swap [a, b, c, d] = [b, a, c, d]
```

It is clear, moreover, that all the representations of a given molecule can be obtained from any one representation by repeated applications of the above transformations. We can freely permute the bonds of the leading carbon atom by combined applications of "rotate" and "swap" and by combining these with applications of "invert" any carbon atom can eventually be brought into the leading position. We can now define a predicate "equiv" on representations of paraffin molecules such that "equiv a b" determines whether a and b represent the same molecule. Thus:

```
equiv a b = member (equivclass a)b

equivclass a = closure_under_laws
                  [rotate,invert,swap][a]
closure_under_laws f s =
                     s ++ closure' f s s

closure' f s t = closure" f s
   (mkset{a|f'←f;a←map f' t;
                        ~ member s a})

closure" f s t = [], t = []
   = t ++ closure' f (s++t) t, t ≠ []
```

In the above "member", "map" and "mkset" are library functions - "mkset" removes repetitions from a list, and "++" is the append operator on lists.

The key idea is embodied in the function "closure_under_laws" which takes a set of functions and a set of objects and finds the closure of the latter under repeated applications of the members of the former. Clearly this is a function which could find applications in a wide range of problems beyond the present one. The above somewhat indirect definition, via the auxiliary function closure' and closure", was chosen for reasons of efficiency.

Generating All Molecules of a Given Size.

Because of the absence of cycles every paraffin molecule must contain at least one occurrence of the methane radical CH_3 and we can without loss of generality[3] choose this to the "leading" carbon atom. A function for generating a list containing (once each) all the paraffin molecules with n carbons can therefore be written

```
paraffin n = quotient equiv
  {[x, "H", "H", "H"] | x ← para (n-1)}

quotient f (a:x) = a:{b|b←quotient f x;
                                ~ f a b}

quotient f [] = []
```

Where "para", to be defined below is a function which returns a list (perhaps with repetitions) of all paraffin radicals of a given size. The function "quotient" defined above takes the quotient of a set with respect to a given equivalence relation (i.e. returns a set containing only one representative of each equivalence class present in the original set) and is used above to ensure that each molecule is represented only once in the final output. There follows a definition of "para"

```
para 0 = ["H"]

para n = {[a, b, c] | i,j,k←[0..n-1];
                  i≤j≤k; i+j+k=n-1;
    a ← para i; b ← para j; c ← para k}
```

At this point we have everything we need to produce a runnable solution to our problem. We have only to define an output structure in terms of "paraffin", thus

```
output = layn (append(map paraffin
                              [1..]))
```

and printing "output" will give us the required list of paraffin molecules in order of increasing size. The list is infinite and so the process of printing it will go on forever, or at least until the user interrupts it at the terminal. (Note - "append" is a library function which takes a list of lists and joins then all together with "++"; "layn" is a standard layout function which causes the elements of a list to be printed one per line with numbered lines.)

This solution, however, runs with appalling slowness (I tried it) mainly because of easily removable inefficiencies in our definition of "para". There is a minor problem and a major problem.

The minor problem is that the way we choose i, j and k in the definition of para n is needlessly wasteful. We could in fact first choose i in the set [0..(n-1)/3] then choose j in the set [i..(n-1-i)/2], whereupon k is fixed to be n-1-i-j. This leads us to rewrite the second line in the definition of "para" as

```
para n = {[a, b, c] | i ← [0..(n-1)/3];
  j ← [i..(n-1-i)/2];  a ← para i ;
  b ← para j ; c ← para (n-1-i-j)}
```

The major problem is that in evaluating "para n" we repeatedly re-evaluate para i for each i < n a large number of times. We need to make para into a "memo-function" [Michie 68] i.e. a function whose value is calculated only once for

each argument, namely on its first call and thereupon stored in a table, so that if it is required again it can be found by table lookup, rather than by recalculation. For a recursive function, like "para", memo-isation leads to an exponential improvement in performance. (or to put it another way, failure to memo-ise leads an exponential deterioration in performance!)

At first sight it seems that a memo-function involves in an essential way the use of side-effects for its expression. This is, however, not the case. There is in SASL a standard transformation for turning a function into a memo function in a purely applicative way - see [Turner 81a, Chapter 4]. Applying the idea to "para" leads us to rewrite its definition as follows:

```
para 0 = ["H"]

para n = paralist n

paralist = map genpara [1..]

genpara n = {[a, b, c] | i ← [0..(n-1)/3];
    j ← [i..(n-1-i)/2]; a ← para i ;
  b ← para j ; c ← para (n-1-i-j)}
```

In the above "genpara" performs the calculation, but the recursion is replaced by table lookup. The lookup table is represented by the (infinite)list "paralist" (in SASL and KRC lists can be indexed by applying them to an integer). The elements of "paralist" are initialised by calling "genpara" but thanks to lazy evaluation they only come into existence as they are accessed.

We now have a runnable program for our paraffin problem. The complete text of the program is shown in figure 2, accompanied by an initial segment of its output. The command "output!" is an instruction to the KRC system to print the list, output.

Lessons drawn

The above program was the fruit of about an hour's labour at a terminal and seems a reasonably convincing demonstration of the utility of recursion equations plus set abstraction as a language framework. The program is far from fully polished and has very much the status of a first cut. By the application of two obvious optimisations - (a) the removal of the redundant "H"s from the internal representation of molecules and (b) the use of an idea called "filter promotion" (see later), I was subsequently led to a program which ran perhaps ten times faster than the above. Rather than pursuing these further refinements in detail here, however, this seems an appropriate moment to break off from the consideration of this particular problem and draw some general lessons.

The first general lesson I would draw is that by the use of an appropriately designed applicative language the effort necessary to arrive at (and the space necessary to express) an executable solution to a problem can be reduced to a small fraction of that required in a traditional programming language. Even in the present situation, where we lack the hardware necessary for the direct support of applicative languages, an implementation of an applicative language can be an extremely valuable tool for the development and testing of algorithms. For example I had a number of misconceptions about the paraffins problem (which I elided from the above account) of which I was fairly quickly disabused by interacting with the KRC system. If I now had to solve the problem in, say, PASCAL, I would do so with much greater confidence.

The second general abservation is that the language framework we are using here supports very nicely the following separation of concerns (which has of course been advocated many times before). In a first step we concentrate on writing down a logically correct definition of the desired function, completely ignoring considerations of efficiency. Recursion plus set abstraction is a very powerful combination for this purpose, enabling us to think very "big thoughts" in one go. Typically, however, the definitions we arrive at in this way have an exponential or combinatorial run-time, whereas there may exist an algorithm which is linear (or at least polynomial). In a second step we repair the efficiency of the definition, by applying transformations know to preserve correctness. In a surprising large number of cases it turns out that a small number of standard optimisations are sufficient to bring about the necessary improvement in performance. Two in particular seem to be of such general applicability as to deserve special mention in a next (and final) section of this paper.

The third and final observation I wish to make relates more specifically to the paraffin problem. I believe that the reason why this seems on first inspection to be rather a hard problem is because it involves an unfree data type and I suspect this is characteristic of a lot of the more recalcitrant problems one meets. A general way of characterising an unfree data type, which we used in this example, is as the quotient of a free data type under an equivalence relation and a good way of defining an equivalence relation is to give a set of laws of which it is the closure. Our function "closure_under_laws" gives us a convenient handle onto this and I hope it will turn out to be useful for other applications in the future.

```
output = layn (append (map paraffin [1..]))
paraffin n = quotient equiv {[x,"H","H","H"]|x<-para (n - 1)}
para 0 = ["H"]
para n = paralist n
paralist = map genpara [1..]
genpara n = {[a,b,c]|i<-[0..(n-1)/3];j<-[i..(n-1-i)/2];
                    a<-para i;b<-para j;c<-para (n - 1 - i - j)}
equiv a b = member (equivclass a) b
equivclass x = closure_under_laws [invert,rotate,swap] [x]
invert [[a,b,c],d,e,f] = [a,b,c,[d,e,f]]
invert x = x, x 1 = "H"
rotate [a,b,c,d] = [b,c,d,a]
swap [a,b,c,d] = [b,a,c,d]
closure' f s t = closure'' f s (mkset {a|f'<-f;a<-map f' t;\member s a})
closure'' f s t = [], t = []
                = t ++ closure' f (s ++ t) t
closure_under_laws f s = s ++ closure' f s s
quotient f (a:x) = a:{b|b<-quotient f x;\f a b}
quotient f [] = []
output!
1)  ["H","H","H","H"]
2)  [["H","H","H"],"H","H","H"]
3)  [["H","H",["H","H","H"]],"H","H","H"]
4)  [["H","H",["H","H",["H","H","H"]]],"H","H","H"]
5)  [["H",["H","H","H"],["H","H","H"]],"H","H","H"]
6)  [["H","H",["H","H",["H","H",["H","H","H"]]]],"H","H","H"]
7)  [["H","H",["H",["H","H","H"],["H","H","H"]]],"H","H","H"]
8)  [["H","H","H"],["H","H","H"],["H","H","H"]],"H","H","H"]
9)  [["H","H",["H","H",["H","H",["H","H",["H","H","H"]]]],"H","H","H"]
10) [["H","H",["H","H",["H",["H","H","H"],["H","H","H"]]]],"H","H","H"]
11) [["H","H",["H",["H","H","H"],["H","H",["H","H","H"]]]],"H","H","H"]
12) [["H","H",["H","H","H"],["H","H","H"],["H","H","H"]]],"H","H","H"]
13) [["H",["H","H","H"],["H",["H","H","H"],["H","H","H"]]],"H","H","H"]
. . . .
```

FIGURE 2 The Paraffins program in K R C , with some initial output.

Two useful optimisations

The two optimisations which warrant special mention here are memo-isation (originally due to Donald Michie) and "filter promotion" (both the name and the idea of which are due to John Darlington) [Darlington 79].

(a) Memo-isation

This optimisation technique has already been demonstrated earlier, on the function "para". A similar example, however, may bring out the method more clearly. Consider the following"obvious" definition of the function "fib n" which returns the n'th fibonacci number.

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2), n>2
```

Although this has some claim to be considered the most natural definition it suffers from a run-time that increases exponentially with n. We could of course program the well-known linear algorithm explicitly (by tail recursion) but it is in fact possible to achieve a linear run-time without abandoning the structure of the above definition.

We do this by turning "fib" into a memo-function. We introduce a data structure "fiblist" in which we store the values of the function and replace all calls to the function in the rest of the program, including the recursive calls inside the definition of "fib" itself, by table-lookup. Thus

```
fiblist = map fib [1..]
fib 1 = 1
fib 2 = 1
fib n = fiblist (n-1)
                  + fiblist(n-2), n>2
```

The run-time of "fib" is now linear instead of exponential!

Obviously the technique can be applied to any function of integer arguments. Notice that this purely applicative approach to memo functions depends heavily on the fact that the language in which we are working has a non-strict semantics. (Incidentally, a more radical approach, which might be worth pursuing, would be to try and get the system to perform this class of optimisations automatically. One quite promising approach, with which I have been experimenting, is to modify the run-time system by keeping an associative cache of the results of all recent function applications.)

(b) Filter Promotion

Again we can best bring out the technique by means of a simple example. Suppose we are asked to modify our earlier definition of the function "partitions" so as to eliminate permutations, say by deciding to allow only increasing partitions, e.g. among the partitions of 3 we allow [1,1,1] and[1,2] but not [2,1]. Our first thought could be to apply a filter to our original function

```
partitions' n = filter increasing
                          (partitions n)
```

where we can easily give a recursive definition of "increasing" as a predicate on lists. We can get a considerable improvement in performance, however, by pushing the filter inside the generator "partitions" so that the unwanted lists are not created in the first place (the reader should compare this with the definitions of partitions earlier in the paper):

```
partitions' 0 = [[]]
partitions' n = {i : p | i ← [1..n];
                    p←partitions' (n-i);
                      p = [] ∨ i ≤ hd p}
```

Like memo-isation, filter promotion can lead to very dramatic improvements in performance.

Note

The use of Zermelo-Frankel set abstraction as an implementable language feature seems originally to have been proposed by John Darlington [Darlington 75].

REFERENCES

Darlington 75 "Applications of program transformation to program synthesis" Proceedings conference on proving and improving programs, Arc et Senans 1975

Darlington 79 "A synthesis of several sorting algorithms" Acta Informatica 1979

Michie 68 "Memo-functions - a language feature with role learning properties" in EXPERIMENTAL PROGRAMMING 1966-7, Edinburgh University, Dept.of Machine Intelligence and Perception, January 1968

Turner 76 "SASL Language Manual", St. Andrews University Technical Report, December 1976

Turner 81 "KRC Language Manual", University of Kent (in preparation)

Turner 81a D.Phil.Thesis, Oxford University 1981