Musical Equational Programs : A Functional Approach

P.M. van den Broek and K.G. van den Berg

Department of Computer Science
University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
email : {pimvdb,vdberg}@cs.utwente.nl

Abstract

In this paper we solve musical equational programs by means of higher order functions. The initial solution is written in a functional programming language (Miranda[1]). It is shown how a solution in an imperative language (Pascal) can be obtained by elimination of the higher order functions.

1. Introduction

Recently Desainte-Catherine and Barbar [1] have given a method to find solutions for musical equational programs using attribute grammars. Their method solves, in a rather ad-hoc way, the problem of the circular dependency of the attributes. Circular dependencies normally cause problems when using an imperative language, since the order of all steps of the computation has to be provided by the programmer. The situation is quite different when one uses a functional programming language with lazy evaluation. An elegant way to evaluate attributes of attribute grammars in these languages is described by Johnsson [2]. A general programming technique, suitable for writing programs with circular dependencies in lazy functional languages, has recently been given by Van Gilst and Van den Broek [3]. Musical equational programs can be solved in a number of straightforward ways: using the techniques of [2] or [3], or using the compiler generator which was developed as an example in [3]. All these solutions are easy to obtain; however, the fact that they can only be written in lazy functional languages may be seen as a drawback.
In this paper we take another approach. Instead of relying on lazy evaluation, we use another important feature of functional languages: higher order functions. We show that musical equational programs can be solved in a straightforward way using higher order functions, and that this method is directly translatable into imperative languages, although these languages do not support higher order functions. Due to the absence of circular dependencies and its single-pass property the resulting imperative program is easily shown to be correct and is easy to comprehend.

2. Musical equational programs

A musical equational program (see [1] for details) can be seen as a combination of a binary tree and an environment. In the syntax of Miranda, the type of the binary tree is given by:

[1] Miranda is a trademark of Research Software Ltd.

```
tree ::= Leaf char | Dot char tree tree | Bar char tree tree
```

Each node of the tree contains a character, which denotes a piece of music. We assume that all characters in a tree differ from each other. Trees are used to represent the structure of pieces of music with respect to concatenation and superimposition. A tree can be seen as a decomposition tree for a piece of music. The tree `Leaf 'A'` expresses that (the music piece which is denoted by) `'A'` is an 'atomic' piece. The tree `Dot 'A' t1 t2` expresses that `'A'` is the concatenation of the pieces corresponding to `t1` and `t2` respectively. The tree `Bar 'A' t1 t2` expresses that `'A'` is the superimposition of the pieces corresponding to `t1` and `t2` respectively. As an example, the tree depicted in figure 1 shows that `'A'` is the concatenation of `'C'`, `'D'` and the superimposition of `'E'` and `'F'`.

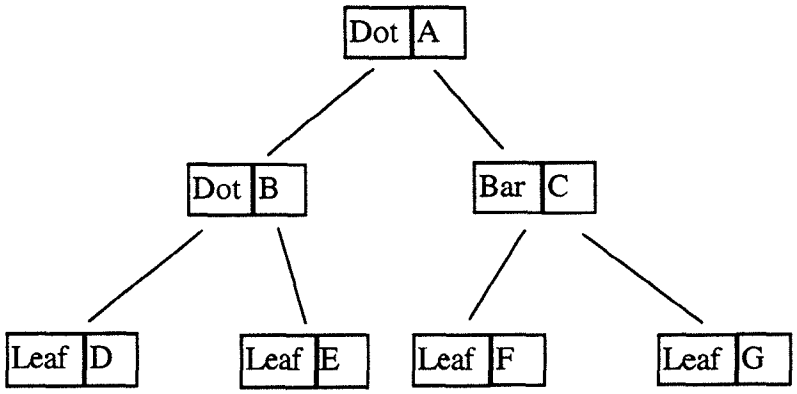


figure 1

This tree is defined (with name `t`) in Miranda by

```
t = Dot 'A' (Dot 'B' (Leaf 'D')(Leaf 'E')) (Bar 'C' (Leaf 'F')(Leaf 'G'))
```

Each piece of music has a length. So a length may be associated with each node of a tree. Due to the interpretation of a tree as decomposition tree of a piece of music, the lengths associated to the nodes of a tree are not independent. The following relations should hold:

length (`Dot c t1 t2`) = length (`t1`) + length (`t2`)
length (`Bar c t1 t2`) = length (`t1`) = length (`t2`)

An environment is an association list, where lengths are associated to music pieces. In Miranda:

```
environment == [(char,num)]
```

An example of an environment is

```
env = [('A',20),('D',4),('E',7)]
```

The problem of solving a musical equational program now amounts to the following: given a tree and an environment, where the environment contains the lengths of a subset of the nodes of the tree, determine whether the lengths of all nodes of the tree are determined uniquely by the environment and by the relations between the lengths of the nodes. If so, give the environment containing the lengths of all nodes of the tree, and if not, report "Error".
As an example, the result of solving the musical equational program given by `t` and `env` is the environment:

```
[('A',20),('B',11),('C',9)('D',4),('E',7),('F',9),('G',9)]
```

3. Solution with higher order functions

We will write a function `eval`, which has an environment and a tree as arguments, and returns either an environment (which contains all lengths of the music pieces of the tree), or an indication of an error (when not all the lengths are uniquely defined). So

```
eval :: environment -> tree -> result
```

where we define the type `result` by

```
result ::= A environment | Error
```

This function `eval`, however, is not suited to be applied recursively to the subtrees of a tree. Consider for instance the subtree for piece `'C'` in figure 1. Applying `eval` to it would deliver `Error`. We have to take into account the fact that the length of the top node of a tree may be determined by a condition outside the tree. This value may be passed to the tree (as an "inherited attribute"), which would be the origin of cyclic dependencies. The solution we propose here does not pass the value to the tree, instead a function is returned which, when applied to the appropriate value, returns the appropriate environment. For instance, for the subtree for piece `'C'` the function `f` is returned for which `f n = [('C',n),('F',n),('G',n)]` holds.
So we will write a function `eval2`, which is a generalisation of `eval`, which also has an environment and a tree as arguments, and returns either an environment (and, for convenience, also the length of the top node), or a function (which has a length as argument and returns an environment), or an indication of an error:

```
eval2 :: environment -> tree -> result2
```

where we define the type `result2` by

```
result2 ::= B (environment,num) | C (num -> environment) | Error2
```

It is easy to express `eval` in terms of `eval2`:

```
eval env tree = f (eval2 env tree)
                where f (B (e,n)) = A e
                      f (C g) = Error
                      f Error2 = Error
```

It is now straightforward to write the function `eval2`. The code may seem rather lengthy, but this is because there are a lot of different cases to be considered: there are three kinds of trees, there are three different kinds of result for the recursive calls of `eval2`, and there is or there is not a binding for the music piece in the environment.

```
eval2 env (Leaf c) = B ([(c,n)],n), present
                   = C f, otherwise
                     where (present,n) = find env c
                           f n = [(c,n)]

eval2 env (Dot c t1 t2)
  = f (eval2 env t1) (eval2 env t2)
    where
    (present,n) = find env c
    f Error2 x                 = Error2
    f (B (e1,n1)) Error2       = Error2
    f (B (e1,n1)) (B (e2,n2)) = Error2, present & n ~= n1+n2
                               = B ((c,n1+n2):e1++e2,n1+n2), otherwise
    f (B (e1,n1)) (C g2)       = B ((c,n):e1++g2(n-n1),n),present
                               = C h, otherwise
                                 where h n = (c,n):e1++g2(n-n1)
    f (C g1) Error2            = Error2
    f (C g1) (B (e2,n2))       = B ((c,n):e2++g1(n-n2),n), present
                               = C h, otherwise
                                 where h n = (c,n):e2++g1(n-n2)
    f (C g1) (C g2)            = Error2

eval2 env (Bar c t1 t2)
  = f (eval2 env t1) (eval2 env t2)
    where
    (present,n) = find env c
    f Error2 x                 = Error2
    f (B (e1,n1)) Error2       = Error2
    f (B (e1,n1)) (B (e2,n2)) = B ((c,n1):e1++e2,n1),
                                  n1=n2 & (~present \/ n=n1)
                               = Error2, otherwise
    f (B (e1,n1)) (C g2)       = B ((c,n1):e1 ++ g2 n1,n1),
                                    ~present \/ n=n1
                               = Error2, otherwise
    f (C g1) Error2            = Error2

    f (C g1) (B (e2,n2))       = B ((c,n2):g1 n2 ++ e2,n2),
                                    ~present \/ n=n2
                               = Error2, otherwise
    f (C g1) (C g2)            = B ((c,n):g1 n++g2 n,n),present
                               = C h, otherwise
                                 where h n = (c,n):g1 n++g2 n
```

Here the function `find` searches for a length of a given piece of music in an environment:

```
find :: environment -> char -> (bool,num)
find [] x = (False,undef)
find ((c,n):env) c' = (True,n), c=c'
                    = find env c', otherwise
```

4. Solution in an imperative language

The solution of the previous section will work in any language which supports higher order functions. Since in imperative languages functions cannot return functions as result, we have to eliminate the higher order functions from the functional solution in order to obtain an imperative solution. We will show how this may be done in Pascal.

From the functional solution of the previous section we see that the functions (of type `num -> environment`) which are returned by the function `eval2` all have the property that they return an environment which depends in a simple way on the argument of the function : the result of applying the function to an argument can be obtained by applying the function to 0, followed by adding the argument to some of the lengths. This means that we may define the type `environment` in such a way that its values can both denote an environment and a function by representing a function by the result of its application to 0. Each binding in an environment then needs a new boolean field, indicating whether or not the corresponding `length` field is increased when the environment is applied to a number. So the type `environment` is defined by:

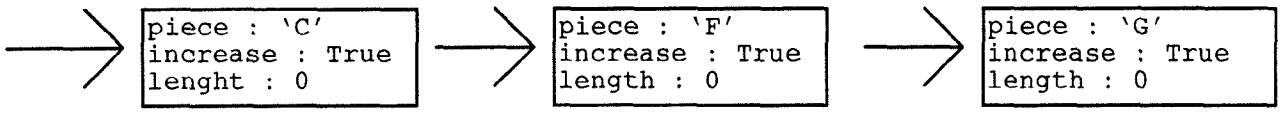
```
environment = ^element;
     element = RECORD
                  piece : CHAR;
                  increase : BOOLEAN;
                  length : INTEGER;
                  next : environment
               END;
```

If a value of type `environment` denotes an environment then its `increase` fields all contain the value False; if it denotes a function then its `increase` fields may contain the value True. The procedure which applies such a function to a number is given by:

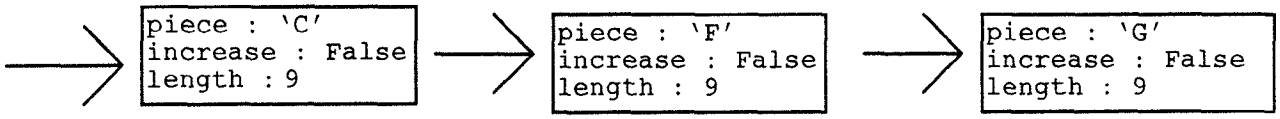
```
PROCEDURE apply (VAR f : environment ; n : INTEGER);

  BEGIN
    IF f <> NIL
    THEN BEGIN
           IF f^.increase
           THEN BEGIN
                  f^.length := n + f^.length;
                  f^.increase := False
                END;
           apply (f^.next,n);
         END;
  END;
```

This procedure transforms the function into the result; this is correct since each function is applied only once. As an example, the result of applying `eval2` to the right subtree of `t` and `env` (from section 2) will be:



Applying this to 9 will change it into:



It is now straightforward to translate the functional solution into an imperative solution.

5. Conclusion

We have given a functional solution for musical equational programs using higher order functions in a lazy functional language. We have subsequently shown how these higher order functions can be eliminated to obtain a solution in an imperative language.

References

[1] M. Desainte-Catherine and K. Barbar, Using attribute grammars to find solutions for musical equational programs, *ACM SIGPLAN Notices* **29** (1994) 56-63

[2] T. Johnsson, Attribute grammars as a functional programming paradigm. In: Functional Programming Languages and Computer Architecture (ed. G. Kahn), Lecture Notes in Computer Science (Springer-Verlag) **274** (1987) 154-173

[3] F.A. van Gilst and P.M. van den Broek, A new programming technique for lazy functional languages. *Science of Computer Programming* **24** (1995) 63-81

Appendix

This appendix contains the result of the translation of the functional solution into an imperative solution.

```
TYPE tree_tag = (Leaf,Dot,Bar);
     tree = ^node;
     node = RECORD
               kind : tree_tag;
               piece : CHAR;
               left,right : tree
            END;

     environment = ^element;
     element = RECORD
                  piece : CHAR;
                  increase : BOOLEAN;
                  length : INTEGER;
                  next : environment
               END;

     result_tag = (B,C,Error);
     result = ^res_record;
     res_record = RECORD
                     kind : result_tag;
                     toplength : INTEGER;
                     env : environment
                  END;

FUNCTION eval2 (env : environment; tree : tree) : result;
VAR present : BOOLEAN;
    n : INTEGER;
    r,r1,r2 : result;
```

```
  PROCEDURE find (e : environment; piece : char; VAR b : BOOLEAN;
                                                 VAR n : INTEGER);
  BEGIN
    IF e = NIL
    THEN b := FALSE
    ELSE IF piece = e^.piece
         THEN BEGIN
                b := TRUE;
                n := e^.length
              END
         ELSE find (e^.next,piece,b,n)
  END;

  PROCEDURE apply (VAR f : environment ; n : INTEGER);

  BEGIN
    IF f <> NIL
    THEN BEGIN
           IF f^.increase
           THEN BEGIN
                  f^.length := n + f^.length;
                  f^.increase := False
                END;
           apply (f^.next,n);
         END;
  END;

  PROCEDURE shift (VAR f : environment ; n : INTEGER);
  BEGIN
    IF f <> NIL
    THEN BEGIN
           IF f^.increase
           THEN f^.length := f^.length + n;
           shift (f^.next,n)
         END
  END;

  PROCEDURE conc (VAR e1,e2 : environment );
  BEGIN
    IF e1 = NIL
    THEN e1 := e2
    ELSE conc (e1^.next,e2)
  END;

BEGIN
  find (env, tree^.piece, present, n);
  CASE tree^.kind OF
  Leaf : BEGIN
           new (r);
           new (r^.env);
           r^.env^.piece := tree^.piece;
           r^.env^.next := NIL;
           IF present
           THEN BEGIN
                  r^.kind := B;
                  r^.toplength := n;
                  r^.env^.increase := False;
                  r^.env^.length := n;
                END
           ELSE BEGIN
                  r^.kind := C;
                  r^.env^.increase := True;
                  r^.env^.length := 0;
                END
         END;
```

```
Dot : BEGIN
        r1 := eval2 (env, tree^.left);
        r2 := eval2 (env, tree^.right);
        CASE r1^.kind OF
        Error : r := r1;
        B : CASE r2^.kind OF
            Error : r := r2;
            B : BEGIN
                  new(r);
                  IF present AND (n <> r1^.toplength + r2^. toplength)
                  THEN r^.kind := Error
                  ELSE BEGIN
                         r^.kind := B;
                         r^.toplength := r1^.toplength +
                                         r2^.toplength;
                         new(r^.env);
                         r^.env^.piece := tree^.piece;
                         r^.env^.increase := False;
                         r^.env^.length := r1^.env^.length +
                                           r2^.env^.length;
                         r^.env^.next := r1^.env;
                         conc (r1^.env,r2^.env)
                       END
                END;
            C : BEGIN
                  new (r);
                  new (r^.env);
                  r^.env^.piece := tree^.piece;
                  IF present
                  THEN BEGIN
                         r^.kind := B;
                         r^.toplength := n;
                         r^.env^.increase := False;
                         r^.env^.length := n;
                         apply (r2^.env, n-r1^.toplength);
                       END
                  ELSE BEGIN
                         r^.kind := C;
                         r^.env^.increase := True;
                         r^.env^.length := 0;
                         shift (r2^.env, -r1^.toplength)
                       END;
                  conc (r1^.env, r2^.env);
                  r^.env^.next := r1^.env;
                END;
            END;
        C : CASE r2^.kind OF
            Error : r := r2;
            B : BEGIN
                  new (r);
                  new (r^.env);
                  r^.env^.piece := tree^.piece;
                  IF present
                  THEN BEGIN
                         r^.kind := B;
                         r^.toplength := n;
                         r^.env^.increase := False;
                         r^.env^.length := n;
                         apply (r1^.env, n-r2^.toplength);
                       END
                  ELSE BEGIN
                         r^.kind := C;
                         r^.env^.increase := True;
                         r^.env^.length := 0;
                         shift (r1^.env, -r2^.toplength)
                       END;
```

```
                    conc (r1^.env, r2^.env);
                    r^.env^.next := r1^.env
                  END;
              C : BEGIN
                    new (r);
                    r^.kind := Error
                  END
            END
        END;
Bar : BEGIN
        r1 := eval2 (env, tree^.left);
        r2 := eval2 (env, tree^.right);
        CASE r1^.kind OF
        Error : r := r1;
        B : CASE r2^.kind OF
            Error : r := r2;
            B : BEGIN
                  new(r);
                  IF (r1^.toplength = r2^.toplength) AND
                              (NOT present OR (n = r1^.toplength))
                  THEN BEGIN
                         r^.kind := B;
                         r^.toplength := r1^.toplength;
                         new (r^.env);
                         r^.env^.piece := tree^.piece;
                         r^.env^.increase := False;
                         r^.env^.length := r1^.toplength;
                         conc (r1^.env,r2^.env);
                         r^.env^.next := r1^.env
                       END
                  ELSE r^.kind := Error
                END;
            C : BEGIN
                  new(r);
                  IF (NOT present OR (n = r1^.toplength))
                  THEN BEGIN
                         r^.kind := B;
                         r^.toplength := r1^.toplength;
                         new (r^.env);
                         r^.env^.piece := tree^.piece;
                         r^.env^.increase := False;
                         r^.env^.length := r1^.toplength;
                         apply (r2^.env, r1^.toplength);
                         conc (r1^.env,r2^.env);
                         r^.env^.next := r1^.env
                       END
                ELSE r^.kind := Error
                END
            END;
        C : CASE r2^.kind OF
            Error : r^.kind := Error;
            B : BEGIN
                  new(r);
                  IF (NOT present OR (n = r1^.toplength))
                  THEN BEGIN
                         r^.kind := B;
                         r^.toplength := r2^.toplength;
                         new (r^.env);
                         r^.env^.piece := tree^.piece;
                         r^.env^.increase := False;
                         r^.env^.length := r2^.toplength;
                         apply (r1^.env, r2^.toplength);
                         conc (r1^.env,r2^.env);
                         r^.env^.next := r1^.env
                       END
```

```
                    ELSE r^.kind := Error;
                  END;
              C : BEGIN
                  new(r);
                    IF present
                    THEN BEGIN
                           r^.kind := B;
                           r^.toplength := n;
                           new (r^.env);
                           r^.env^.piece := tree^.piece;
                           r^.env^.increase := False;
                           r^.env^.length := n;
                           apply (r1^.env, n);
                           apply (r2^.env, n);
                           conc (r1^.env,r2^.env);
                           r^.env^.next := r1^.env
                         END
                    ELSE BEGIN
                           r^.kind := C;
                           new (r^.env);
                           r^.env^.piece := tree^.piece;
                           r^.env^.increase := True;
                           r^.env^.length := 0;
                           conc (r1^.env,r2^.env);
                           r^.env^.next := r1^.env
                         END
                  END
              END
          END
        END
  END;
  eval2 := r
END;

FUNCTION eval (env : environment; tree : tree) : result;
VAR e,r : result;

BEGIN
  e := eval2 (env,tree);
  IF e^.kind = C
  THEN  r^.kind := Error;
  eval := e
END;
```