HIGHER ORDER GENERALIZATION IN PROGRAM DERIVATION

Alberto Pettorossi
IASI - CNR
Viale Manzoni 30
00185 Roma (Italy)

Andrzej Skowron
Institute of Mathematics
Warsaw University
00-901 Warsaw (Poland)

ABSTRACT

We define and study a particular kind of generalization strategy for deriving efficient functional programs. It is called *higher order generalization* because it consists in generalizing variables or expressions into functions. That strategy allows us to derive efficient one-pass algorithms with low time$\times$space complexity.

Through some examples we show the power of our generalization strategy and its use together with the tupling strategy. Applying those strategies one may avoid the introduction of circular programs [Bir84].

1. INTRODUCTION

A major problem in the derivation of programs by transformation is the lack of a general theory which guarantees the improvements of program performances when applying the basic transformation rules.

In some cases, however, it is possible to realize those improvements by using powerful strategies. Some of them have been defined and studied in the past, as for instance, the composition strategy, the tupling strategy, and the generalization strategy. For a recent survey in this area the reader may refer to [Fea86].

We will define a new kind of generalization strategy and we will study its properties through a couple of examples. That strategy, together with the composition and the tupling strategy, avoids the multiple traversal of data structures and it saves time and space resources. Related work can be found in [Bir84].

We consider recursive equation programs, like the ones presented in the classical work by Burstall and Darlington [BuD77]. We will not give their formal definition here, but we hope that the reader will have no difficulties in understanding them. The actual language we use is a variant of HOPE [BMS80].

Obviously, the generalization strategy we propose is independent from the language chosen, and it can also be applied when one derives programs using different formalisms.

The basic transformation rules for recursive equation programs include:

- the **unfolding rule**. It is the replacement of a left hand side of a recursion equation by its corresponding rigth hand side.
  For instance, given the equations:
  ```
      f(x)=E1[g(a)],       g(x)=E2[x],
  ```
  where E[e] denotes the expression E with the occurrence of the subexpression e, the unfolding of g(a) in E1 produces the following new program version:
  ```
      f(x)=E1[E2[a/x]],    g(x)=E2[x].
  ```
- the **folding rule**. It is the inverse of the folding rule by interchanging the l.h.s. and the r.h.s.
- the **definition rule**. It is the introduction of a new recursive equation whose r.h.s. is not an instance of already existing equations.

Those basic rules have been often described in the literature (see, for instance, [Fea86]). We will not go into the details here. Let us only remark that we need to use some strategies, because a naive sequence of applications of the unfolding and folding rules may take us back to the initial program version.

In what follows we will apply the higher order generalization strategy for solving two problèms: the first one is a *compilation problem* due to Swierstra [Swi85] and the second one is a *tree transformation problem* due to Bird [Bir84]. We think that the programs we will derive have good merits with respect to efficiency and clarity of derivation. (Their correctness will be given us for free, as usual in the transformation approach).

2. HIGHER ORDER GENERALIZATION FOR A COMPILATION PROBLEM

We consider a compilation problem for transforming lists of letters denoting declarations and uses of identifiers into new lists, where for each use of an identifier we indicate the corresponding declaration.

In order to formally specify our problem, let us introduce the following data structures atom and prog (short for program):

data atom == decl(letter) ++ use(letter)

data prog == list atom

where letter is a given set, which we may consider to be {a,b,…,z}. decl, use, and list are type constructors. list atom denotes the type of *nested lists* of atoms.

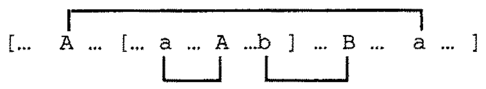
We also assume that together with a data definition we are given the corresponding *discriminators*. For instance, in our case we are given "isuse" and "isdecl". isuse satisfies the following axiom: isuse(y)=true iff y=use(x) for any $x \in$ letter, and analogously for isdecl.

For simplicity, we also adopt the convention of writing x instead of use(x) and X instead of decl(x). We hope that no confusion will arise between the letter x and the atom use(x), both written as x.

Thus, the set of atoms is {A,a,B,b,…,Z,z}.

Here are two examples of lists of type prog:
[A a c [B b a] C a] and [[A b] a B]. Another one is the list:
p1=[A a b [a c A D b C] B b], which we will use as a running example.

We assume that the declarations in the lists of type prog obey the familiar block discipline, where blocks are identified by square brackets. For instance, if we have the following prog:



with no other occurrences of A's and B's, the declaration-use correspondences are denoted by the arcs we have drawn.

Notice that for any letter x its declaration X may occur *after* (that is, to the rigth of) its first use x. Indeed in p1 the declaration C occurs after c.

We also assume that in any given prog, each use of a letter has a unique corresponding declaration, which occurs in its block or in one of its enclosing blocks.

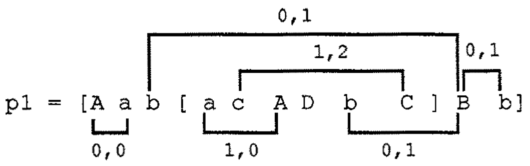
In the Appendix we provide the function OK:prog → bool, which checks that condition.

For instance, OK([A a a [B b]])=true, OK([A B [b a]])=true, and OK([A a a b [a A] B])=true, while OK([A A a])=false (because there are two A declarations within the same block) and OK([a B [b A]]) = false (because there is no an "active" declaration A for a).

We would like to compile a given nested list of atoms into a nested list of pairs of numbers, where each pair corresponds to a use-occurrence of an atom in the given list. The first number of each pair gives us the *level of nesting* of the block where the corresponding declaration occurs, while the second number gives us the *sequence order* of that declaration within the block where it occurs. For instance, for p1 we want to obtain the list:

l1 = [(0,0) (0,1) [(1,0) (1,2) (0,1)] (0,1)],

which encodes the use-declaration correspondence shown by the following arcs:



The pair (0,0) which is for the first a from the left, tells us that the corresponding declaration A is at level of nesting 0 and it is the first declaration (from the left) in that level. Analogously, the pair (1,2) for c tells us that the corresponding declaration C is at level of nesting 1 and it is the third declaration in it (after A and D).

For obtaining the list l1 from p1 a possible first step is to derive the "decorated list" dp1=[A00 a b [a c A10 D11 b C12] B01 b], where we have attached to each declaration the corresponding <level-of-nesting, sequence-order> pair. (For simplicity, we wrote Xnm instead of X<n,m>).

Having dp1, it will be easier to compute the list l1, because we have available for each declaration the relevant pair of numbers. Unfortunately, we have to pay that advantage, because we are forced to traverse the list dp1, after the first traversal of the given list p1 (which was necessary to derive dp1).

However, the application of the tupling strategy and the higher order generalization strategy will avoid that drawback, and it will allow us to obtain an efficient one-pass algorithm. The main contribution of this paper consists exactly in this point.

We also show the power of those strategies when we use them together, because we obtain the same efficiency results which are possible at the expenses of extending our language by allowing circular programs [Bir84].

Therefore, for representing the list dp1 we need the following data structures, where level and order are natural numbers:

```
    data decoratom == decl(letter) ×level ×order  ++  use(letter)
    data decorprog == list decoratom
```

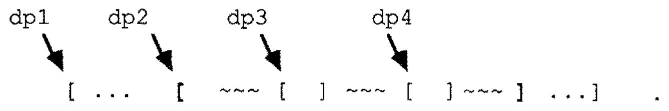
with the discriminators: isddecl and isuse.

The following function **decor** produces the decorprog dp1 from p1:

```
dec decor: prog ×level ×order → decorprog
--- decor(nil,n,r)=nil
--- decor(e::l,n,r)=if isuse(e) then e::decor(l,n,r)
                    elseif isdecl(e) then <e,n,r>::decor(l,n,r+1)
                    else decor(e,n+1,0)::decor(l,n,r)                  •
```

We have: dp1=decor(p1,0,0).

Now we present the function **nad** which computes the **n**ew **a**ctive **d**eclarations (represented as functions from letters to <level,order> pairs) *at the top level* of a block in a given decorprog.
nad works by taking as a second argument the active declarations in the enclosing blocks. As an example, consider the following decorprog dp1:



nad(dp2,d) computes the declarations valid in the sections with tildes, for a given d representing the declarations valid in the sections with dots. The declarations valid in dp3 and dp4 can be computed by a recursive call of nad.

```
dec nad: decorprog × (letter → level×order)
        → (letter → level × order)
--- nad(nil,d)=d
--- nad(e::l,d)=if isuse(e) then nad(l,d)
                elseif isddecl(e) then update(e,nad(l,d))
                else nad(l,d)                                          •
```

Given a function f, update(<x,n,r>,f) produces the new function g s.t. g(x)=<n,r> and g(y)=f(y) for y≠x.

The active declarations at the top level of a given dp1 are computed by nad(dp1,emptyfunction), because there are no enclosing blocks.

Given dp1 and the active declarations at the top level of dp1, the following function **comp** (short for compile) computes the desired list l1.

```
dec comp: decorprog × (letter → level × order) → list level × order
--- comp(nil,d)=nil
--- comp(e::l,d)=if e=use(x) then d(x)::comp(l,d)
                 elseif isddecl(e) then comp(l,d)
                 else comp(e,nad(e,d))::comp(l,d)                    •
```

Therefore:
l1=comp(dp1,nad(dp1,emptyfunction)) where dp1=decor(p1,0,0), because we have first to decorate the list p1, and then we have to compute the new active declarations in dp1 for an empty enclosing block. Finally we have to compile dp1.

The compiling program we have constructed makes multiple traversals of the data structures involved. It seems very difficult to produce in this case a one-pass algorithm, because the declaration of the identifiers may occur after their use. However, we will show that the higher order generalization strategy, together with the tupling strategy, allows us to solve that problem.

We do not present here a formal characterization of the power of those strategies and their synergism, but we hope that the reader may convince himself that the proposed strategies do work in a large number of cases.

3. THE TRANSFORMATION PROCESS TOWARDS THE ONE-PASS COMPILATION

A first step towards the derivation of the one pass algorithm we have specified in the previous Section is the application of the *composition strategy* [Pet84a] for the initial expression comp(dp1,nad(dp1,emptyfunction)), because both comp and nad visit the same data structure, and the latter is an argument of the former.

That is a standard case for applying that strategy, which usually avoids the generation of intermediate data (see also [Wad85]).
The incorporation into the one-pass algorithm of the function decor(p1,0,0) which constructs dp1, will be done later.

By composition we define the function:
f(l,d)=comp(l,nad(l,d)). After a few folding/unfolding steps we get the following explicit definition:

```
dec f: decorprog × (letter → level × order) → list level × order
```

```
--- f(nil,d)=nil
--- f(e::l,d)=if e=use(x) then nad(l,d)(x)::f(l,d)
               elseif isddecl(e) then comp(l,update(e,nad(l,d)))
               else f(e,nad(l,d))::f(l,d)                              •
```

From the above definition of f we notice that:
i) the functions nad(l,d) and f(l,d) both visit the same data structure l, and
ii) it is impossible to fold into a recursive call of f the expression comp(l,update(e,nad(l,d))), because it does not match the expression comp(l,nad(l,d)).

As indicated in [Pet84b] the fact i) suggests us to apply the *tupling strategy*, while for point ii) we need to use the *higher order generalization strategy*, which consists in generalizing an expression into a function. In our case it works as follows.

We define the function compile(l,g(e1,l,d)) defined as:
comp(l,update(e1,nad(l,d))) if g=λxyz.update(x,nad(y,z)), and
comp(l,nad(l,d)) if g=λxyz.nad(y,z).

Now the folding step required in point ii) is possible, and we can use a recursive call of compile with the suitable higher order argument g. The idea of the higher order generalization is related to the one in [Dar81], where the author uses the *mismatch* information deriving from a forced folding, to find a suitable generalization step.

We define the function:
H(e1,l,d,g)=<nad(l,d), compile(l,g(e1,l,d))>.
The functionality of H can be obtained from those of nad and compile. The latter one is:
(decorprog × (atom × level × order) × decorprog × (letter→(level × order))
→ (letter→(level×order))) → list level × order.

After a few folding/unfolding steps we get the recursive equations for the function H, where we used the following notations:
H(e1:e2,l,d,g1) = <nad(l,d), comp(l,update(e1,update(e2,nad(l,d))))>,
g1=λxyz.update(x,nad(y,z)), g2=λxyz.nad(y,z), and
πi <a1,…,an> = ai for i=1,…,n.

```
H(e1,nil,d,g)=<d,nil>
H(e1,e::l,d,g)=if e=use(x) then
                 <u, (if g=g1 then update(e1,u)(x) else u(x)) :: v>
                         where <u,v> = H(e1,l,d,g)
                elseif isddecl(e) then <update(e,u),v>
```

```
                              where <u,v> = if g=g1 then H(e1:e,l,d,g1)
                                                         else H(e,l,d,g1)
                     else <u,  (if g=g1 then π2 H(e1,e,update(e1,u),g2)
                                       else π2 H(e1,e,u,g2)) :: v>
                              where <u,v> = H(e1,l,d,g)                        •
```

Therefore, for producing the list l1 from dp1 we compute:

π2 H(◊,dp1,emptyfunction,g2) = comp(dp1,nad(dp1,emptyfunction)) (by definition), where ◊ satisfies this equality: update(◊,d)=d.

Notice that during the computation, the function H visits its second argument e::l only once. Indeed, H(…,e::l,…) is computed in terms of H(…,e,…) and H(…,l,…).

Therefore, by using the tupling strategy and the higher order generalization strategy we avoided the multiple traversals of e::l. On the contrary, they would have been necessary if we used the functions nad and comp. We will come back to this point later.

Testing the equality of functions when computing H is easy, because it amounts to check syntactical identities only. (Indeed one could simply code g1 and g2 using the numbers 1 and 2.)

A final step remains to be done. We need to avoid the visit of the given prog p1 for producing the corresponding decorated prog dp1.

In order to do so, we have to redo the steps we have presented above for the derivation of H from comp(l,nad(l,d)).

We will replay that derivation using as a starting point suitable variants of the functions nad and comp. We call those variants Nad and Comp. They are produced by applying again the composition strategy. Their inputs are prog's, not decorprog's. By that process we realize the promised incorporation of the function decor into nad and comp.

We have: Nad(p,d,n,r)=nad(decor(p,n,r),d). Its definition is:

```
dec Nad: prog × (letter → level × order) × level × order
                         → (letter → level × order)
--- Nad(nil,d,n,r)=d
--- Nad(e::l,d,n,r)=if isuse(e) then Nad(l,d,n,r)
                    elseif isddecl(e) then update(<e,n,r>,Nad(l,d,n,r+1))
                    else Nad(l,d,n,r)                                        •
```

The call nad(decor(p1,0,0),emptyfunction) is replaced by:
Nad(p1,emptyfunction,0,0).

We also have: Comp(p,d,n)=comp(decor(p,n,0),d). Its definition is:

```
dec Comp: prog × (letter → level × order) × level
                         → list level × order
--- Comp(nil,d,n)=nil
--- Comp(e::l,d,n)=if e=use(x) then d(x)::Comp(l,d,n)
                   elseif isdecl(e) then Comp(l,d,n)
                   else Comp(e,Nad(e,d,n+1,0),n+1)::Comp(l,d,n)        •
```

Notice that, in analogy to Nad, we could have defined the function Comp(p,d,n,r)=comp(decor(p,n,r),d), but a simple analysis of the resulting equations would have shown that the argument r is not necessary.
The call comp(decor(p1,0,0),emptyfunction) is replaced by:
Comp(p1,emptyfunction,0).

Now, as for nad and comp, the tupling and generalization strategies suggest us the definition of the following function L (analogous to H):
L(e1,l,d,g,n,r)=<Nad(l,d,n,r), Compile(l,g(e1,l,d,n,r),n)>, where:
Compile(l,g(e1,l,d,n,r),n)=Comp(l,update(e1,Nad(l,d,n,r)),n) if g=g1,
=Comp(l,Nad(l,d,n,r),n) if g=g2.

The types of the arguments of L are:
e1:atom×level×order, l:prog, d:letter→level×order,
g:(atom×level×order)×prog×(letter→ level×order)×level×order
→(letter→level×order), n:level, r:order, and the type of the output of L is: (letter→ level×order)×(list level×order).

As we did for H, we then derive the equations for L, where L(b:c,…,g1,…) stands for <…, Comp(…,update(b,update(c,…)),…)>:

```
L(e1,nil,d,g,n,r)=<d,nil>
L(e1,e::l,d,g,n,r)=if e=use(x) then
                  <u, (if g=g1 then update(e1,u)(x) else u(x)) :: v>
                    where <u,v> = L(e1,l,d,g,n,r)
                  elseif isdecl(e) then <update(<e,n,r>,u),v>
                    where <u,v>= if g=g1 then L(e1:<e,n,r>,l,d,g1,n,r+1)
                                 else L(<e,n,r>,l,d,g1,n,r+1)
                  else <u, (if g=g1 then π2 L(e1,e,update(e1,u),g2,n+1,0)
                            else π2 L(e1,e,u,g2,n+1,0)) :: v>
                    where <u,v> = L(e1,l,d,g,n,r)                      •
```

Thus the required list l1 is equal to π2 L(◊,p1,emptyfunction,g2,0,0), which is equal to Comp(p1,emptyfunction,0).
As usual, for ◊ we have: update(◊,d)=d.

The derivation process is now completed and we derived a one-pass algorithm as required.

Let us clarify the notion of *one-pass* in our context. It is a notion relative to a particular argument of the function being defined. In our case it is the second argument of L, which is the given list of atoms to be "compiled". We say that L is one-pass because for each branch of its conditional definition, the recursive calls of L have as arguments disjoint substructures of the relevant argument. Indeed, L(…,e::l,…) is defined in terms of L(…,e,…) and L(…,l,…) only.

One could doubt whether it is actually convenient to use one-pass algorithms at the expenses of having functions with higher order parameters. For that respect let us remark that: i) the initial versions of our programs may already have higher order parameters (like ñad, in our case) and ii) the higher order generalization may result in the use of a *low order* parameter only (g = 1 or 2, in our case).

For our derivation computer experiments showed that for progs of length 40 or more, the one-pass algorithm is indeed faster than the initial multi-pass version. (Obviously, those performances depend also on how fast parameters are passed among recursive calls in the available machine.)

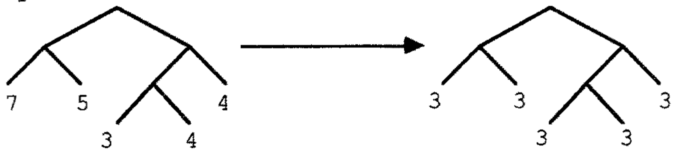
The transformation steps we have shown, are quite tedious to be made by hand. A transformation system like the one described in [BaP77, Fea79] can be of great help.

A final remark concerns the readability of the derived versions. Indeed, it is difficult to understand the definition of the function L. That fact should not be regarded as a drawback of the transformation method. One only need to understand the initial program versions. The application of the basic rules and strategies will guarantee that correctness is preserved and performances are improved.

An alternative solution to our compilation problem is presented in [Swi85]. Also in that case higher order functions are used, and the solution is found by applying a method based on attribute grammars evaluation.

4. A TREE TRANSFORMATION PROBLEM

Let us consider a second example of application of the higher order generalization strategy. It is taken from a problem described in [Bir84]. We are asked to replace the value of all leaves in a given tree by their minimal value. For instance:



The obvious solution of the problem requires two traversals of the tree: the first one for computing the minimal leaf value and the second one for performing the replacement. We get the program:

```
data tree(num) == niltree ++ tip(num) ++ tree(num)Δtree(num)
dec transform: tree(num) → tree(num)
--- transform(t) = replace(t,minv(t))
dec minv: tree(num) → num
--- minv(niltree) = +∞
--- minv(tip(n)) = n
--- minv(t1Δt2) = min(minv(t1),minv(t2))
dec replace: tree(num) × num → tree(num)
--- replace(niltree,m) = niltree
--- replace(tip(n),m) = tip(m)
--- replace(t1Δt2,m) = replace(t1,m) Δ replace(t2,m)                 •
```

A way of avoiding a second traversal of the given tree is to remember its structure when computing the minimum leaf value. If one does so, a second visit for replacing the leaf values is not necessary. Remembering the tree structure and finding the minimum leaf can be done at the same time by defining a higher order function and using the tupling strategy as follows.

```
dec struct_min: tree(num) → ((num → tree(num)) × num)
--- struct_min(niltree)=<λx.niltree,+∞>
--- struct_min(tip(n))=<λx.tip(x),n>
--- struct_min(t1Δt2)=<λx.str1Δstr2, min(m1,m2)>
      where <λx.str1,m1>=struct_min(t1), <λx.str2,m2>=struct_min(t2) •
```

Thus, transform(t) becomes: a1(a2) where <a1,a2>=struct_min(t).

One may object that in the above program the given tree has been copied when constructing the first component of the output of struct_min, and therefore the program is not space efficient.

However, since struct_min visits the tree only once, one may discard the leaves of the tree after their visit. A *destructiveness analysis* can be helpful in this case [Pet84c]. Thus, given a tree t, for constructing the first component of struct_min(t) we can reuse the memory cells which were needed for storing t.

The computation evoked by struct_min(t) when producing the output <a1,a2> and the subsequent application of a1 to a2 can be seen as follows: first, the given tree is visited to find the minimum leaf and pointers to the leaf positions are recorded, and then, the pointed positions are filled with the value which has been found.

Thus, the generalization strategy can be applied also for avoiding the use of pointers. They will be represented by parameters of suitable functions, and then function applications, that is, passing actual parameters, realize the required manipulations.

The higher order generalization strategy is used in this example for generalizing a *data structure* into a *function which manipulates it* (not for allowing a folding step, as in the previous compilation problem). From a tree t we indeed obtained the tree transformer: $\pi 1$(struct_min(t)).

The use of a higher order object, like the first component of struct_min, allows us to achieve in our tree transformation problem the same performances obtained by using circular programs and lazy evaluation in [Bir84].

5. CONCLUSIONS

We defined a higher order generalization strategy, and we illustrated through examples its important role in the derivation of efficient programs by transformations. That role has been already recognized in the area of automated deduction and theorem proving for the invention of suitable lemmas [Aub76, BoM75, Cha76].

We want to stress that the *mismatch* information for a forced folding was crucial for suggesting our generalization steps. Related work has been done by [AbV84, HuH82, MaW79] for proving properties of recursively defined functions and various approaches to program synthesis.

A point for further investigation is the generalization technique for obtaining data structure transformers, which we presented in the previous Section. It can be viewed as realizing *communications among agents* [Pet84a].

A final point to be underlined is the synergism between the generalization strategy and the tupling strategy. Neither of them, if used separately, could have been powerful enough to solve with the required efficiency the transformation problems we considered. Their joint use was essential for our derivations.

6. ACKNOWLEDGEMENTS

Many thanks to the members of the IFIP WG.2.1 for their stimulating conversations. R.Bird, R.Paige, D.Swierstra, and the referees made valuable suggestions.

This work was supported by the IASI Institute of the National Research Council in Rome (Italy) and the Institute of Mathematics of Warsaw University (Poland).

7. REFERENCES

[AbV84] Abdali, K.S. and Vytopil, J.: "Generalization Heuristics for Theorems Related to Recursively Defined Functions" Report Buro Voor Systeemontwikkeling. Postbus 8348, Utrecht, Netherlands (1984).

[Aub76] Aubin, R.: "Mechanizing Structural Induction" Ph.D. Thesis, Dept. of Artificial Intelligence, University of Edinburgh (1976).

[BaP77] Bauer, F.L., Partsch, H., Pepper, P. and Wössner, H.: "Notes on the Project CIP: Outline of a Transformation System" TUM-INFO-7729 Tech. Report Institut für Informatik, der Technischen Universität München, Germany (1977).

[Bir84] Bird,R.S.: "Using Circular Programs to Eliminate Multiple Traversal of Data" Acta Informatica 21 (1984), 239-250.

[BMS80] Burstall, R.M., MacQueen, D.B., and Sannella, D.T.: "HOPE: An Experimental Applicative Language" Proc. LISP Conference 1980 Stanford USA (1980), 136-143.

[BoM75] Boyer, R.S. and Moore, J.S.: "Proving Theorems About LISP Functions" J.A.C.M. 22, 1 (1975), 129-144.

[BuD77] Burstall, R.M. and Darligton, J.: "A Transformation System for Developing Recursive Programs" J.A.C.M. Vol.24, 1 (1977) 44-67.

[Cha76] Chatelin, P.: "Program Manipulation: to Duplicate is not to Complicate" Report Université de Grenoble, CNRS Laboratoire d'Informatique (1976).

[Dar81] Darligton, J.: "An Experimental Program Transformation and Synthesis System" Artificial Intelligence 16, (1981), 1-46.

[Fea79] Feather, M.S.: "A System for Developing Programs by Transformations" Ph.D. Thesis, Dept. of Artificial Intelligence, University of Edinburgh (1979).

[Fea86] Feather, M.S.: "A Survey and Classification of Some Program Transformation Techniques" Proc. TC2 IFIP Working Conference on Program Specification and Transformation. Bad Tölz, Germany (ed.

L. Meertens) (1986).
[HuH82] Huet, G. and Hullot, J.M.: "Proofs by Induction in Equational Theories with Constructors" JCSS 25, 2 (1982), 239-266.
[MaW79] Manna, Z. and Waldinger, R.: "Synthesis: Dreams →Programs" IEEE Transactions of Software Engineering SE-5, 4 (1979), 294-328.
[Pet84a] Pettorossi, A.: "Methodologies for Transformation and Memoing in Applicative Languages" Ph. D. Thesis, Computer Science Department, Edinburgh University, Edinburgh (Scotland) (1984).
[Pet84b] Pettorossi, A.: "A Powerful Strategy for Deriving Efficient Programs by Transformation" ACM Symposium on Lisp and Functional Programming, Austin, Texas (1984), 273-281.
[Pet84c] Pettorossi, A.: "Constructing Recursive Programs which are Space Efficient" in: Computer Program Synthesis Methodologies (Biermann, Guiho, and Kodratoff, eds.) Macmillan Publ. Co., New York (1984), 289-303.
[Swi85] Swierstra, D.: "Communication 513 SAU-15". IFIP WG.2.1, Suasalito, California, USA (1985).
[Wad85] Wadler, P.L.: "Listlessness is Better than Laziness" Ph. D. Thesis, Computer Science Department, CMU-CS-85-171, Carnegie Mellon University, Pittsburgh, USA (1985)

8. APPENDIX

The following function OK tests whether or not a given prog has exactly one declaration occurrence for each letter.

```
dec OK: prog → bool
--- OK(p) = let <b,v> = activedecl(p,ϕ) in
            if b then OKdecl(p,v) else false                  •
```

OK calls the function OKdecl(p,v) tests whether or not in a given context there is *at least* one declaration for each use of a letter. OKdecl takes as a second argument a set v of letters, which includes: i) the definitions occurring in the blocks enclosing the prog p, and ii) the definitions which are active at the top level of p (not in subblocks, i.e. sublists of p). It is defined as follows:

```
dec OKdecl: prog × set letter → bool
--- OKdecl(nil,v) =true
--- OKdecl(e::l,v)=if e=use(x) then (decl(x)∈v and OKdecl(l,v))
                   elseif isdecl(e) then OKdecl(l,v)
                   else let <b,u> = activedecl(e,ϕ) in
                     if b then (OKdecl(e, u ∪v) and OKdecl(l,v))
                     else false                                •
```

OK and OKdecl call the following function activedecl which given a prog p, tests the existence of *at most* one declaration for each letter. In the case of a positive answer, that is the first component of the answer is true, the second component gives us the active declarations for the top level of p. (The second argument for activedecl is used only for collecting the declarations encountered so far while visiting p.)

```
    dec activedecl: prog × set letter → bool × set letter
    --- activedecl(nil,v)=<true,v>
    --- activedecl(e::l,v)=if isdecl(e) then
                 (if e∈v then <false,ϕ> else activedecl(l, v ∪{e}))
                 else activedecl(l,v)                                    •
```

The function OK requires multiple visits of the prog p. We leave to the reader the task of deriving a one-pass algorithm as we did in Sections 2. and 3. That derivation requires again the application of the strategies we have described in the paper.

It will not be difficult (although a bit cumbersome to do by hand) to incorporate also that program into the compilation algorithm of Section 3.