MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN – POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

DERIVING PROGRAMS USING HIGHER ORDER GENERALIZATION

ALBERTO PETTOROSSI

IASI, Italian National Research Council, Roma, Italy

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. This strategy allows the derivation of efficient one-pass algorithms which save time and space resources.

Introduction

A major problem in the derivation of programs by transformation is the lack of a general theory which guarantees the improvement of program performances when applying the basic transformation rules. In some cases, however, it is possible to achieve that improvement by using particular strategies.

Some of them have been defined and studied in the past, as for instance the composition strategy, the tupling and the generalization strategy [9]. In this paper we will define a particular kind of generalization strategy and we will study its properties through an extended example.

That strategy, together with the composition and the tupling strategies, avoids the multiple traversals of data structures and it saves time and space resources.

We consider recursive equations programs like the ones used in the classical work by Burstall and Darlington [6]. We will not give their formal definition here, because it is not relevant to the higher order generalization strategy we will present. Indeed that generalization can be applied also when one derives programs using other programming languages and other formalisms. In the programs we will write we adopt a Hope-like syntax [4].

As basic transformation rules we will use the fold/unfold rules, i.e., the replacement of a right-hand side of a recursive equation by its corresponding left-hand side (for folding) or vice versa (for unfolding) [6].

Our generalization strategy will be presented in relation to a problem due to Swierstra [13]. It is a more complicated version of the tree transformation problem presented in [3].

The higher order generalization strategy: an extended example

The example we consider is related to a *compilation problem* for transforming a list of letters denoting declarations and uses of identifiers in a block structured language, into a new list, where for each use of an identifier we indicate the corresponding declaration.

Let us consider the following data structure [4]:

data atom == use(letter) ++ decl(letter),
data elem == list atom,

where letter $= \{a, b, c, \ldots\}$, and use, decl, and list are type constructors. Here we assume that list α is the type of the *nested* lists of elements of type α.

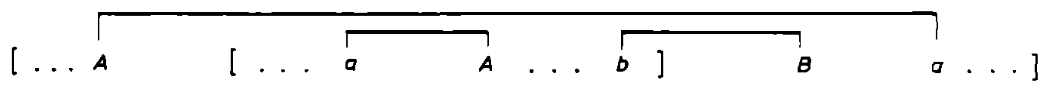
For simplicity we adopt the convention of writing x instead of use(x), and X instead of decl(x).

An instance of elem (short for program element) is:

$$\text{p1} = [A, a, b, [a, c, A, b, C], B, b].$$

In p1 the occurrence of the atom A denotes the declaration of the identifier a while the occurrence of the atom a denotes the use of the same identifier a. (The ambiguity of writing x both for the letter x and the atom use(x) is resolved by the context.)

Notice that for any letter x the declaration X may occur *after* its use. We assume that active declarations satisfy the familiar block discipline. For instance, if we have the following program element:



with no other occurrence of A's or B's, the correspondence use-declaration has been indicated by arcs.

We also assume that given a program element p, each use of a letter has at most one corresponding declaration in the same block, i.e., OKdecl(p, ϕ) = true, where

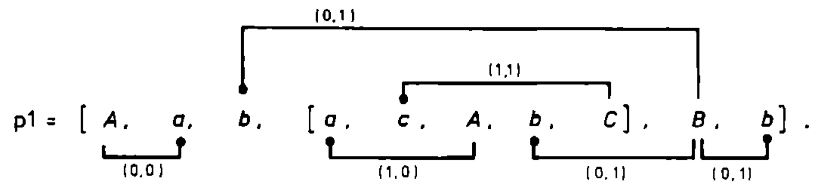
```
dec  OKdecl: elem × set letter → bool,
---  OKdecl (nil, v) = true,
---  OKdecl (e: : le, v) = if e = use(a) then OKdecl(le, v)
                  elseif e = decl(a) then    if a ∈ v then false
                                             else OKdecl(le, v ∪ {a})
                  else OKdecl (e, φ) and OKdecl(le, v).
```

For instance, OKdecl$([A, a, a, b, [a, A], B], \phi)$ = true and OKdecl$([A, A, a], \phi)$ = false, because there are two declarations for the letter a within the same block.

We would like to write a program which given a nested list of atoms, produces a nested list of pairs of numbers, where each pair corresponds to a **use** occurrence. The first number of each pair gives us the *level of nesting* of the block where the corresponding declaration occurs, the second number gives us the *sequence order* of that declaration within the block where it occurs. For instance, given the above list pl we want to derive the list:

$$l1 = [(0, 0)\ (0, 1)\ [(1, 0)\ (1, 1)\ (0, 1)]\ (0, 1)],$$

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



The pair (0, 0) 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, 1) related to c tells us that the corresponding declaration C is at level of nesting 1 and it is the second declaration in it.

We can write the program for producing the list $l1$ by first obtaining an intermediate "decorated list"

$$\mathbf{pl} = [A00, a, b, [a, c, A10, b, C11], B01, b],$$

where we attached to each declaration the corresponding ⟨level of nesting, sequence order⟩ pair. Having the list **pl** it will be much easier to derive the required list $l1$, because we have already available the necessary information attached to each declaration. We pay that advantage by requiring multiple traversals of data structures. However, the application of the tupling strategy and the higher order generalization strategy will avoid that drawback, and it will allow us to derive an efficient one-pass algorithm, as we will see later.

For representing the list **pl** we need the following "decorated data structure":

```
data decoratom == use (letter) ++ decl(letter) × level × order,
data decorelem == list decoratom.
```

The program which produces **pl** from pl can be written as follows:

```
dec  decor: elem × level × order → decorelem
---  decor(nil, n, o) = nil
--- decor(e: : le, n, o) = if e = use(a) then use (a): : decor(le, n, o)
                elseif e = decl(a) then ⟨decl(a), n, o⟩: : decor (le, n, o+1)
                else decor (e, n+1, 0): : decor (le, n, o).
```

The initial call is decor(p1, 0, 0).

We can compute the new active declarations at each point in the given list of atoms by the function nad:

```
dec  nad: (decorelem ×(letter → level ×order)) →(letter → level ×order)
---  nad(nil, d) = d
---  nad(e: : le, d) = if e = use(a) then nad(le, d)
       elseif e = ⟨decl(a), n, o⟩ then update (⟨decl(a), n, o⟩, nad(le, d))
       else nad(le, d)
```

where as usual, update $(\langle x, n, o\rangle, f)$ defines the function g s.t.

$$g(x) = \langle n, o\rangle \quad \text{and} \quad g(y) = f(y) \quad \text{for} \quad y \neq x.$$

The inital call is nad(**pl**, emptyfunction).

The code for nad expresses the fact that when entering a list of atoms (denoting a new block), the new active declarations are computed by updating the old active declarations by the ones occurring within that list, but not within its nested lists. (see Fig. 1).

Entering a new block. The new active declarations for updating the old ones, occur here:

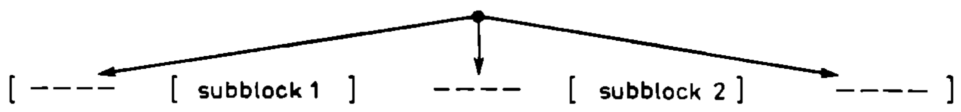


Fig. 1. Computing the new active declarations

The following function comp (short for compile) computes the desired list ll from **pl**.

```
dec  comp: (decorelem ×(letter → level ×order)) → list level ×order
---  comp (nil, d) = nil
---  comp (e: : le, d) = if e = use(a) then d(a): : comp(le, d)
           elseif e = ⟨decl(a), n, o⟩ then comp(le, d)
           else comp (e, nad(e, d)): : comp(le, d).
```

The initial call of comp is comp(**pl**, nad(**pl**, emptyfunction)), where **pl** = decor(p1, 0, 0).

Obviously the program we have constructed makes multiple traversals of the data structures involved. It seems very difficult to produce in our case a one-pass algorithm, because the declaration of an identifier may occur in a given list after its use. However, we will show that the higher order generalization strategy, together with the tupling strategy, is powerful enough to solve that problem. We do not present here a formal characterization of the power of that kind of generalization. Nevertheless we hope that the reader may convince himself that the proposed strategy does work in a large class of programs which occur in practice.

A first step towards the derivation of the one-pass algorithm is the application of the *composition strategy* for the expression comp(**pl**, nad (**pl**, emptyfunction)), because both comp and nad visit **pl**. The incorporation of the step for obtaining **pl** from pl into the one-pass algorithm will be done later on.

We define the function $f(l, d) = \text{comp}(l, \text{nad}(l, d))$ whose explicit definition can be obtained, after some simple folding/unfolding steps:

```
dec  f: (decorelem × (letter → level × order)) → list level × order
---  f(nil, d) = nil
---  f(e :: le, d) = if e = use(a) then nad (le, d)(a) :: f(le, d)
         elseif e = ⟨decl(a), n, o⟩
                     then comp (le, update(⟨decl(a), n, o⟩, nad(le, d)))
         else f(e, nad(le, d)) :: f(le, d).
```

From the above definition of f we notice that:

(i) the function nad(le, d) and $f(le, d)$ both visit the structure le;

(ii) it is impossible to fold into a recursive call of f both the expressions comp$(le$, nad$(le, d)) = f(le, d)$ and comp$(le$, update$(\langle \text{decl}(a), n, o\rangle$, nad$(le, d)))$.

As suggested in [12] we need to apply the tupling strategy (because of (i)). We also need to apply the higher order generalization strategy (because of (ii)) to make the folding possible. Therefore we define the function

$$H(b, l, d, g) = \langle \text{nad}(l, d), \text{compile}(l, g(b, l, d))\rangle$$

where

$$\begin{aligned} \text{compile}(l, g(b, l, d)) &= \text{comp}(l, \text{update}(b, \text{nad}(l, d))) \qquad &\textbf{if } g = \text{g1} \\ &= \text{comp}(l, \text{nad}(l, d)) &\textbf{if } g = \text{g2} \end{aligned}$$

where $\text{g1} = \lambda xyz.\text{update}(x, \text{nad}(y, z))$ and $\text{g2} = \lambda xyz.\text{nad}(y, z)$.

The functionality of H can be derived from the one of the function compile:

$$\begin{aligned} &(\text{decorelem} \times ((\text{atom} \times \text{level} \times \text{order}) \times \text{decorelem} \times (\text{letter} \to (\text{level} \times \text{order})) \\ &\qquad\qquad \to (\text{letter} \to \text{level} \times \text{order}))) \to \text{list level} \times \text{order}. \end{aligned}$$

Using the higher order function compile, we are able to make the necessary folding steps to allow the recursion to work. The suggestion for the suitable generalization comes from the need of folding the two expressions of point (ii) above.

This idea is related to the one in [8] where the author uses for a generalization step the mismatch information coming from a forced folding.

It is not difficult to derive the explicit expression of the function H and we get:

```
$H(b, \text{nil}, d, g) = \langle d, \text{nil}\rangle$
$H(b, e::le, d, g) =$ **if** $e = \text{use}(a)$ **then**
{**if** $g = \text{g1}$ **then** $\langle u, \text{update}(b, u)(a)::v\rangle$ **where** $\langle u, v\rangle = H(b, le, d, \text{g1})$
 **else** $\langle u, u(a)::v\rangle$                        **where** $\langle u, v\rangle = H(b, le, d, \text{g2})$}
            **elseif** $e = \langle \text{decl}(a), n, o\rangle$ **then**
{**if** $g = \text{g1}$ **then** $\langle \text{update}(\langle \text{decl}(a), n, o\rangle, u), v\rangle$
                               **where** $\langle u, v\rangle = H(b{:}\,\langle \text{decl}(a), n, o\rangle, le, d, \text{g1})$
 **else** $\langle \text{update}(\langle \text{decl}(a), n, o\rangle, u), v\rangle$
                               **where** $\langle u, v\rangle = H(\langle \text{decl}(a), n, o\rangle, le, d, \text{g1})$}
            **else**
{**if** $g = \text{g1}$ **then** $\langle u, a::v\rangle$ **where** $a = \pi 2H(b, e, \text{update}(b, u), \text{g2})$
                               **where** $\langle u, v\rangle = H(b, le, d, \text{g1})$
 **else** $\langle u, a::v\rangle$              **where** $a = \pi 2H(b, e, u, \text{g2})$
                               **where** $\langle u, v\rangle = H(b, le, d, \text{g2})$}.
```

We used the following notations:

$$H(b{:}c, l, d, \text{g1}) = \langle \text{nad}(l, d), \text{comp}(l, \text{update}(b, \text{update}(c, \text{nad}(l, d))))\rangle.$$

and $\pi i\,\langle a1, \ldots, an\rangle = ai$.

The initial call for producing $l1$ from **p1** is $\pi 2H(\square, \mathbf{p1}, d, \text{g2}) = \text{comp}(\mathbf{p1}, \text{nad}(\mathbf{p1}, \text{emptyfunction}))$ where update $(\square, y) = y$.

Notice that the function H visits the list $e::le$ (which is its second argument) only once. By using the tupling strategy and the higher order generalization strategy we avoided the multiple traversals of $e::le$, which are necessary if we use the functions nad and comp. Indeed $H(\ldots, e::le, \ldots)$ is computed only in terms of the calls $H(\ldots, e, \ldots)$ and $H(\ldots, le, \ldots)$. Testing the equality of functions when computing the function H is not difficult, because it amounts to check a syntactic identity.

A final transformation step remains to be done because we need to avoid the visit of the given list of atoms for producing the corresponding decorated list.

In order to do so, we basically have to redo the steps we have presented above when deriving the function H. This time, however, we need to consider as a starting point suitable variants of the functions nad and comp, which we call Nad and Comp, whose input is a list of atoms (not decorated atoms).

We leave to the reader the task of completing the necessary transformation steps in detail.

We have:

```
dec Nad: elem × (letter → level × order) × level × order
→ (letter → level × order)
   --- Nad(nil, d, n, o) = d
   --- Nad(e::le, d, n, o) = if e = use(a) then Nad(le, d, n, o)
      elseif e = decl(a) then update (⟨decl(a), n, o⟩, Nad(le, d, n, o+1))
      else Nad(le, d, n, o).
```

The initial call is Nad(p1, emptyfunction, 0, 0).

```
dec Comp: elem × (letter → level × order) × level → (list level × order)
--- Comp(nil, d, n) = nil
--- Comp(e::le, d, n) = if e = use(a) then d(a)::Comp(le, d, n)
       elseif e = decl(a) then Comp(le, d, n)
       else Comp(e, Nad(e, d, n+1, 0), n+1)::Comp(le, d, n).
```

The initial call is Comp(p1, emptyfunction, 0).

The tupling and the generalization strategies suggest us the definition of the following function L (analogous to H):

$$L(b, l, d, g, n, o) = \langle \mathrm{Nad}(l, d, n, o), \mathrm{Compile}(l, g(b, l, d, n, o), n)\rangle$$

where b: atom × level × order, l: elem, d: letter → level × order,

g: ((atom × level × order) × elem × (letter → level × order) × level × order)

→ (letter → level × order)), n: level,

and the functionality of the output of L is: (letter → level × order) × (list level × order).

Compile $(l, g(b, l, d, n, o), n) = \mathrm{Comp}(l, \mathrm{update}(b, \mathrm{Nad}(l, d, n, o)), n)$ if $g = \mathrm{g1}$. It is equal to $\mathrm{Comp}(l, \mathrm{Nad}(l, d, n, o), n)$ if $g = \mathrm{g2}$.

After some folding and unfolding steps, we can derive the following explicit definition of L:

```
L(b, nil, d, g, n, o) = ⟨d, nil⟩
L(b, e: : le, d, g, n, o) = if e = use(a) then
|if g = g1 then ⟨u, update(b, u)(a): : v⟩ where ⟨u, v⟩ = L(b, le, d, g1, n, o)
 else ⟨u, u(a): : v⟩                        where ⟨u, v⟩ = L(b, le, d, g2, n, o)|
                         elseif e = decl(a) then
|if g = g1 then ⟨update(⟨decl(a), n, o⟩, u), v⟩
                     where ⟨u, v⟩ = L(b:⟨decl(a), n, o⟩, le, d, g1, n, o+1)
 else ⟨update(⟨decl(a), n, o⟩, u), v⟩
```

```
                    where ⟨u, v⟩ = L(⟨decl(a), n, o⟩, le, d, g1, n, o+1)}
                               else
{if g = g1 then ⟨u, a: : v⟩ where a = π2L(b, e, update(b, u), g2, n+1, 0)
                            where ⟨u, v⟩ = L(b, le, d, g1, n, o)
 else ⟨u, a: : v⟩           where a = π2L(b, e, u, g2, n+1, 0)
                            where ⟨u, v⟩ = L(b, le, d, g2, n, o)}.
```

The initial call is $\pi 2L(\square$, p1, emptyfunction, g2, 0, 0) = Comp(p1, emptyfunction, 0) where update ($\square$, a) = a.

By $L(b{:}c, \ldots,$ g1, ...) we mean

$$\langle \ldots, \mathrm{Comp}(\ldots, \mathrm{update}(b, \mathrm{update}(c, \ldots)), \ldots)\rangle.$$

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

Notice the power of the synergism between the tupling strategy and the generalization strategy. By tupling we collect the necessary information in the individual components of the tuple, so that multiple traversals are avoided, and by generalizing we make the folding step possible, so that recursive programs can be derived.

A second example and some comments on the generalization strategy

Let us consider as a second example of application of the higher order generalization strategy, the following problem [3]. We are asked to change in a given tree the values of all the leaves by replacing them by their minimal value. The obvious solution to the problem requires two traversals of the tree: the first one for computing the minimum leaf value, and the second one for replacing the leaf values.

The corresponding program is as follows.

```
data tree(num) == niltree ++ tip(num) ++ tree(num) ∧ tree(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(t1 ∧ t2) = min(minv(t1), minv(t2))
dec replace: tree(num) × num → tree(num)
--- replace(niltree, m) = niltree
--- replace(tip(n), m) = tip(m)
--- replace(t1 ∧ t2, m) = replace(t1, m) ∧ replace(t2, m).
```

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

$$\textbf{dec}\ \text{gmin: tree(num)} \rightarrow ((\text{num} \rightarrow \text{tree(num)}) \times \text{num}).$$

Given a tree(num), gmin produces a pair whose first component is the tree structure, i.e., a "tree without leaf values", and whose second component is the minimum leaf value. The required tree is obtained by applying the first component of the value of gmin to the second one.

```
--- gmin(niltree) = ⟨λx.niltree, +∞⟩
--- gmin(tip(n)) = ⟨λx.tip(x), n⟩
--- gmin (t1 ∧ t2) = ⟨λx.c1 ∧ c2, min(m1, m2)⟩
                        where ⟨λx.c1, m1⟩ = gmin(t1)
                              ⟨λx.c2, m2⟩ = gmin(t2).
```

The initial call for a given tree t is:

$$\text{a1 (a2)}\ \textbf{where}\ \langle \text{a1, a2} \rangle = \text{gmin}(t).$$

Looking at the above program one may object that the given tree has been copied when constructing the first component of the result, and therefore the program is not space efficient. However, since the function gmin visits the tree only once, one may discard the leaves of the tree after their visit. Thus, given a tree t, for constructing the first component of gmin(t) we can reuse the memory cells which are needed for storing t.

Notice that the use of a higher order structure, like the first component of gmin, allows us to achieve in this case the same program performances which can be obtained by circular programs and lazy evaluation [3].

For lack of space we do not present here more examples of the higher order generalization strategy, but we hope that we succeeded in illustrating the important role that such generalization play when deriving programs. That role has already been recognized in the area of automated deduction and theorem proving for the generation of suitable lemmas [2], [5], [7].

Another point we want to stress is the role of the mismatch information for a forced folding in suggesting us the generalization steps. That idea goes back to [8], [2]. Related work has been done by [1], [10], [11] for program synthesis and for proving properties of recursively defined functions.

A final point to be underlined is the synergism among the generalization strategy we proposed, and the tupling strategy [12]: neither of them, if used separately, could have been powerful enough to solve with the required efficiency, the transformation problems we considered in this paper. Their joint use was essential for our derivations.

Acknowledgements. I would like to thank Prof. H. Rasiowa and Prof. G. Mirkowska for their kind invitation at the Banach Semester 1985. Without their generosity this paper could not have been written. This work was also partly supported by IASI of the Italian National Research Council.

References

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

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

[3] R. S. Bird, *Using Circular Programs to Eliminate Multiple Traversals of Data*, Acta Informatica 21 (1984), 239–250.

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

[5] R. S. Boyer and J. S. Moore, *Proving Theorems About LISP Functions*, J. ACM. 22, 1 (1975), 129–144.

[6] R. M. Burstall and J. Darlington, *A Transformation System for Developing Recursive Programs*, J. ACM. 24 (1977), 44–67.

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

[8] J. Darlington, *An Experimental Program Transformation and Synthesis System*, Artifical Inteligence 16 (1981), 1–46.

[8] M. S. Feather, *A System for Developing Programs by Transformation*, Ph. D. Thesis University of Edinburgh (1979).

[10] G. Huet and J. M. Hullot, *Proofs by Induction in Equational Theories with Constructors*, JCSS 25, 2 (1982), 239–266.

[11] Z. Manna and R. Waldinger, *Synthesis: Dreams → Programs*, IEEE Transactions Software Engineering SE-5, 4 (1979), 294–328.

[12] A. Pettorossi, *A Powerful Strategy for Deriving Efficient Programs by Transformation*, ACM Symp. on LISP and Functional Programming. Austin Texas (1984), 273–281.

[13] D. Swierstra, Communication 513 SAU-15, IFIP WG.2.1 Sausalito, California (1985).

Presented to the semester
Mathematical Problemes in Computation Theory
September 16–December 14, 1985