PROGRAM DEVELOPMENT USING LAMBDA ABSTRACTION

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

ABSTRACT

We study the problem of avoiding multiple traversals of data structures in functional programming. A solution proposed in [Bir84] makes use of lazy evaluation and local recursion. We show that analogous perfomances can be achieved using higher order functions and lambda abstractions. The solution we propose works for the call-by-value evaluation rule and it does not require local recursion. We also show through some examples that higher order functions allow us to simulate the use of pointers of a Pascal-like language. We finally spend a few words on a theory of strategies for program development.

1. INTRODUCTION

We address the problem of program derivation from specifications, or more precisely, the problem of deriving efficient programs by transformation and symbolic evaluation. In particular, we focus our attention on the strategies one may adopt for the development of functional programs. We analyze the problems involved in the construction of programs which avoid multiple traversals of data structures, and we propose a novel approach as an alternative to the one suggested in [Bir84], which need local recursion and lazy evaluation. Moreover, our approach suggests a specific methodology, and we think that, precisely for that respect, it is an improvement on the approaches of [Bir84, Tak87]. In those papers, in fact, no general methods are introduced for the definition of the new functions needed during the development of programs.

We show that using higher order functions and the lambda abstraction strategy (defined in the paper), one can derive through program transformation, one-pass algorithms from multi-pass ones. We also give suggestions to the programmer on how to invent the auxiliary functions needed for the derivations and how to apply the proposed strategy (together with the tupling strategy [Pet84]). Those suggestions are given through the study of some challenging examples.

In functional programs the efficient manipulation of data structures is considered to be a difficult problem to tackle, because pointers are not avalable. We will study that problem and we will present a clean solution which makes use of lambda expressions.

We will also analyze the relationship between the "shifting of data boundaries" in program development [JøS86] on one hand, and the composition and lambda abstraction strategies on the other hand. We will also compare the generation of new bound variables for the lambda expressions

introduced using abstraction, with the use of extra arguments in Prolog-like programs.

Finally we will present some preliminary ideas for the study and the classification of various program derivation strategies.

2. THE PALINDROME EXAMPLE

As a first example of use of our proposed *lambda abstraction strategy* (also called *higher order generalization* strategy [PeS87], or simply, *abstraction* strategy) we will derive a program for testing whether or not a list of integers is palindromic. In this example we will also use the tupling strategy, which is already well known and often considered in the transformation methodology (for a brief account see [Fea86]). The *lambda abstraction strategy* consists in replacing the expression:

"e where x = e1" by the semantically equivalent expression: "(λx.e) e1".

Since the subexpression e1 occurring within e is replaced by the variable x, the application of the abstraction strategy may allow us to perform "folding steps" which are otherwise impossible.
It may also allow us to suspend computations because the evaluation of e is performed only when the lambda expression λx.e is applied to e1. This fact is very important, because often we may also want to simulate lazy evaluation using call-by-value.

For our palindrome problem we will obtain a one-pass algorithm whose execution is essentially the same as the one evoked by the program given in [Bir84], but it does not need either *local recursion* or *lazy evaluation*. We will achieve the same time × space performances via the use of the abstraction strategy. The bound variables will play the role of pointers and we will be able to manipulate them within the framework of functional languages in a referentially transparent way.

As in [Bir84] we start off from the following initial program version:

1. palindrome(l) = eqlist(l,rev(l))
2. eqlist([],[]) = true
3. eqlist(a:l1,b:l2) = (a=b) and eqlist(l1,l2)
4. rev([]) = []
5. rev(a:l1) = rev(l1) @ [a].

As usual ':' denotes the 'cons' operation on lists, @ denotes the concatenation of lists, and [] denotes the empty list. Now we may apply the composition strategy for the given expression of palindrome(l), that is, eqlist(l,rev(l)).
After a few unfolding steps we get:

palindrome([]) = true
palindrome(a:l) = eqlist(a:l, rev(a:l))
= a=hd(rev(l) @ [a]) and eqlist(l, tl(rev(l) @ [a]))
= a=hd(x @ [a]) and eqlist(l, tl(x @ [a])) where x=rev(l)

and by abstraction strategy we have:

palindrome(a:l) = λx.(a=hd(x @ [a]) and eqlist(l, tl(x @ [a]))) rev(l). (#)

Now, looking for a recursive definition of palindrome(l), it seems that our derivation process cannot proceed because we cannot fold eqlist(l,tl(x@[a])) as an instance of eqlist(l,rev(l)). At this

point we use the "mismatch information" [Dar81] for choosing the strategy to apply.

In order to allow a folding step between eqlist(l, tl(x @ [a])) and eqlist(l,rev(l)) we need to *generalize* rev(l) to a variable, say z. (That generalization technique has been often used in the literature [Aub76, BoM75]). Then, by abstraction from eqlist(l,z) we get: λz.eqlist(l,z). Moreover, since rev(l) and λz.eqlist(l,z) both visit the same data structure l we can apply the *tupling strategy* [Pet84]. Thus we are led to the following definition of the auxiliary function:

$Q(l) \equiv <$ λz.eqlist(l,z), rev(l) $>$. Its explicit definition is:

6. Q([]) = < λz.null(z), [] > (by unfolding);

7. Q(a:l)= < λz.eqlist(a:l,z), rev(a:l) >

= < λz. a=hd(z) and eqlist(l, tl(z)), rev(l) @ [a] >

= < λz. a=hd(z) and u(tl(z)), v@[a] > where <u,v>=Q(l) (by folding).

The final program consists of the above equations 6, and 7, together with:

1'. palindrome(l) = u v where <u,v> = Q(l).

Notice that the abstraction strategy together with the tupling strategy allows us to express Q(a:l) in terms of Q(l) only. Therefore a linear recursive definition of Q(l) is derived and the list l is visited only once. During that visit we test the equality of lists and at the same time, we compute the reverse of the given list.

The above algorithm requires essentially the same time and space resources needed in the algorithm presented in [Bir84], which we rewrite here for the reader's convenience:

B1. Bird-palindrome(l) = π1(p) whererec p = eqrev(x, π2(p), [])

B2. eqrev([], y, z) = < true, [] >

B3. eqrev(a:l, y, z) = < (a=hd(y)) and u, v > where <u,v>=eqrev(x, tl(y), a:z).

As usual, πi(p) denotes the i-th projection of p.

Two questions naturally arise at this point: i) in what sense the given list l is visited only once, and ii) what is the relation between our program and Bird's program.

Here is the sequence of Q calls for computing palindrome([1,2]) using our algorithm:

palindrome([1,2]) = u v where <u,v>=Q([1,2]):

l=[1,2] Q(l) = < λz.1 =hd(z) and u([2]), v@[1] > where <u,v> = Q(tl(l))

tl(l)=l'=[2] Q(l') = < λz.2=hd(z) and u([]), v@[2] > where <u,v> = Q(tl(l'))

tl(l')=l''=[] Q(l'') = < λz.null(z), [] >.

During the visit of a given list l, a functional data structure u and a list data structure v are constructed as the first and second component of Q(l)=<u,v>. The final application (u v) could be considered as a second visit of the list rev(l) recorded in v, but it formally occurs without reference to hd or tl operations.

In any case the described behaviour is the same which is obtained using Bird-palindrome, but in Bird's approach it is necessary to compute using lazy evaluation. We are able to do without it, because the relevant computations are suspended via lambda abstractions.

Notice that the construction of one-pass algorithm depends on the access functions' which are available for the data structures under consideration. Indeed the one-pass palindrome algorithm would have been obvious if for any given list *l* we had the access functions 'last(*l*)' computing the last element of *l* and the function 'begin(*l*)' computing the list *l* without the last element.

The use of a one-pass algorithm for palindrome(*l*) allows 'on-line computations' in the sense that while the items of a given list are input in a left to right order, some computations can be performed, and at the end of the input, what it remains to be done is only a final function application. (That operation can be considered as an unexpensive task within an applicative language.)

Computer experiments in our ML implementation on VAX-VMS show that the derived program (1",6',7') runs faster than the original program (1,2,3,4,5) for lists of about 100 items or more.

The advantages we get with our higher order approach are the following ones:

i) we can explicitly manipulate bounded variables, which can be used like pointers. In Bird's approach pointers are implicitly used, and it is difficult to see that the derived programs preserve termination. In our approach termination is guaranteed by structural induction of the function definitions.

ii) We do not need the lazy evaluation mode, but call-by-value together with lambda abstraction is sufficient.

iii) The programmer is not required to learn any unfamiliar method. Bird himself says that his local recursive programs "require a little practice". For instance, it is not immediate to see that in Bird-palindrome the replacement of B3 by the following:

B3'. eqrev(a:*l*, b:y, z) = < (a=b) and u, v > where <u,v>=eqrev(x, y, a:z),

leads to a an infinite loop [Bir84].

On the contrary, we think that lambda expressions are familiar objects, and any functional programmer knows how to use functions as "first class citizens" (see [Bac78, Bir86]).

iv) In the derivation of the program B1-B2-B3, Bird does not give any fundamental reason for the introduction of new auxiliary function 'eqrev'. Sometimes in his paper analogous definitions are suggested by "little foresight" [Bir84, p. 245] or given without justification. Our aim is to provide methods for suggesting suitable function definitions: this is the essential role of the strategies we present and we will show through some examples that they work in a large variety of cases.

Let us finally remark that as in Bird's program we can perform the derivation using an "accumulator version" of the reverse function, that is, using rev(*l*) defined as rev1(*l*,[]) where:

rev1([],y) = y

rev1(a:*l*,y) = rev1(*l*,a:y), where the second argument behaves indeed as an accumulator.

In that case we use the function R(*l*,y) ≡ < λz.eqlist(*l*,z), rev1(*l*,y) > instead of the function Q(*l*) ≡ <λz.eqlist(*l*,z), rev(*l*)>.

The final program one can get is the following one:

1". palindrome(*l*) = u v where <u,v>=R(*l*,[]).

6'. R([],y) = < λz.null(z), y >

7'. R(a:*l*,y) = < λz.a=hd(z) and u(tl(z)), v > where <u,v>=R(*l*,a:y).

That program can be gotten by replaying the same derivation steps presented above. This fact also shows the power of the abstraction and tupling strategies. Notice that the given example is an evidence for the need of replaying program derivations. Indeed most existing program-derivation systems have that "replay" feature.

The reader should also notice that the generalization strategy *without* the lambda abstraction, is *not* powerful enough to solve our problem of obtaining a one-pass algorithm. In fact, if we do not use abstraction we have, instead of the function R, the function:

$R^*(l,z,y) \equiv < eqlist(l,z), rev1(l,y) >$, whose explicit definition is:

8. $R^*([],z,y) = < null(z),\ y >$
9. $R^*(a{:}l,z,y) = < a{=}hd(z)$ and $u,\ v >$ where $<u,v>=R^*(l, tl(z), a{:}y)$. We also have:
10. $palindrome^*(l) = \pi 1\ R^*(l, rev1(l,[]), [])$.

But, unfortunately, palindrome*(*l*) is not good enough for our purposes: it visits twice the list *l* in a call-by-value mode of evaluation. In fact, we need to compute first the value of the argument rev1(*l*,[]) and then we have to compute R*(*l*, rev1(*l*,[]), []), visiting again the list *l*. It was exactly this point which stimulated Bird's analysis in his paper.

3. MODIFYING AND SORTING LEAVES OF TREES

Let us consider some more examples of program derivation using the lambda abstraction strategy. Through those examples we hope that the reader may convince himself that the abstraction strategy is not an ad-hoc mechanism. Those examples were used also by Bird to show that circular programs and call-by-need are nice features to include in applicative languages for deriving very efficient programs. We do not contrast that view here, we only want to show that the same efficiency can be achieved if we allow ourselves functions which return functions as values.

We will also see that the lambda abstraction strategy allows the clean handling of pointers within the framework of applicative languages (via the use of the bound variables) and it makes it possible to suspend computations in a call-by-value semantics.

The first example is about the construction of the binary tree Min-tree(t) which is like a given tree t, except that all its leaves are equal to the minimum leaf of t.

Let tip: num→tree and ^: tree×tree→tree be the obvious constructors for binary trees. A two-pass algorithm is as follows:

Min-tree(t) = repl(t,minl(t)), where minl computes the minimum leaf value:

minl(tip(n)) = n

minl(t1^t2) = min(minl(t1),minl(t2)), and repl(t,m) replaces the leaf values of a given tree t by its minimum leaf value m:

repl(tip(n),m) = tip(m)

repl(t1^t2,m) = repl(t1,m) ^ repl(t2,m).

Looking for a one-pass algorithm we may try to apply the composition strategy by defining: V(t) ≡ repl(t,minl(t)). However, having obtained:

V(tip(n)) = tip(n)

V(t1^t2) = repl(t1, minl(t1^t2)) ^ repl(t2,minl(t1^t2)),

we cannot fold the two above recursive calls of repl. We may try to apply the generalization strategy [AbV84, Aub76] by defining: V*(t,t') ≡ repl(t,minl(t')), but the reader may check that it does not work. A possible way to proceed is to apply the lambda abstraction strategy, so that:

V(t1^t2) = repl(t1,x) ^ repl(t2,x) where x = min(minl(t1),minl(t2))

= (λx.(repl(t1,x) ^ repl(t2,x))) min(minl(t1),minl(t2)).

Now we can apply the tupling strategy because repl and minl both visit the same data structure t1. We define: W(t) ≡ < λz.repl(t,z), minl(t) >. We get:

M1. W(tip(n)) = < λz.tip(z), n >

M2. W(t1^t2) = < λz.(repl(t1,z) ^ repl(t2,z)), min(minl(t1),minl(t2)) >

= < (λz. T1^T2), min(m1,m2) > where <λz.T1, m1> = W(t1)

and <λz.T2, m2> = W(t2)

M3. Min-tree(t) = f m where <f,m> = W(t).

The folding step for equation M2 is possible and the one-pass algorithm is obtained. In fact, W(t1^t2) is defined in terms of W(t1) and W(t2) only. During the visit of a given tree t, we construct in the first component of W(t) a functional data structure which basically encodes a copy of t and therefore, we wast space. One may say that in the above algorithm time efficiency has been achieved at the expense of memory efficiency. However, at this point the *destructiveness analysis* [Myc81, Pet78] can be applied, so that the cells necessary for the first component of W(t) are obtained from the ones of the given tree t, which can be discarded 'on the fly', while the visit progresses.

We think that our derivation of the algorithm M1-M2-M3, which is essentially the same given by Bird, has its importance, because it clarifies the folding requirement in the invention of the relevant auxiliary functions. In Bird's paper those functions are introduced without much explanation, so that the reader may find difficult to use his proposed methodology in other programs.

The example we have considered above shows also a correspondence between *logical variables* in Prolog-like languages and *bound variables* of lambda abstractions, which we will now explain.

Let us look first at a solution to the above tree transformation problem written in Prolog. (For this version we thank A. Falkner and T. Frühwirth of the Technical University of Vienna).

P1. mintree(Intree, Outtree) ← visit(Intree, Outtree, Min, Min).

P2. visit(treeof(InL,InR), treeof(OutL,OutR), Inmin, Outmin)

← visit(InL, OutL, Inmin,MinL), visit(InR, OutR, Inmin, MinR),
min(MinL, MinR, Outmin).

P3. visit(tip(N), tip(X), X, N).

min(M,N,X) binds X to the minimum value between M and N.

The above Prolog program is deterministic and it makes one visit only of the given input tree Intree when producing the output tree Outtree. It is the equality of the last two arguments of the initial call of the predicate 'visit' which realizes the desired tree modification.

The correctness of the program mintree is not too difficult to prove, but it is *not* given us for free,

while the correctness of our program Min-tree is straightforward, because it was derived by transformation. It would be interesting to develop a program transformation method for Prolog programs in order to obtain, for instance, the above program P1-P2-P3 from a two-pass algorithm of the form:

P1'. mintree1(Intree, Outtree) ← minleaf(Intree, Min), build(Intree,Outtree,Min).

P2'. minleaf(treeof(L,R), Min) ← minleaf(L,MinL), minleaf(R,MinR), min(MinL,MinR,Min).

P3'. minleaf(tip(N), N).

P4'. build(treeof(InL,InR), treeof(OutL,OutR), Min)

← build(InL,OutL,Min), build(InR,OutR,Min).

P5'. build(tip(N), tip(Min), Min).

The Prolog program P1-P2-P3 shows the correspondence between the role of the extra logical variable which is used by the predicate 'visit' (with 4 arguments) w.r.t. the predicate 'build' (with 3 arguments) and the bound variable z in the first component of W(t), that is, λz.repl(t,z).

The flow of information during the execution of the program P1-P2-P3 shows also an interesting relationship between the Prolog programs we have presented and the evaluation of attributes in *Attribute Grammars*. Indeed the two arguments Inmin and Outmin of the predicate 'visit' in the equation P2 above, are respectively inherited and synthesized attributes.

Prof. Swierstra recently showed us an applicative program for computing Min-tree(t) which was derived (without using transformation techniques) by analysing the given problem from the attribute grammars point of view [Swi85]. The ability of writing a one-pass algorithm is related to the fact that for Knuthian semantic systems [ChM79] with synthesized and inherited attributes one can obtain equivalent systems in a purely synthesized form. ■

Let us consider now a second example. In this case we are asked to construct a tree which is like a given one, but its leaves should be ordered 'from left to right'. That construction should take place in one-pass only. A two-pass version of the program is as follows:

S1. Sort-tree(t) = replace(t,sort(leaves(t))),

where: i) leaves(t) computes the list of the leaves of the tree t,

ii) sort(*l*) rearranges the list *l* of leaf values in an ascending order from left to right, and

iii) replace(t,*l*) uses the leftmost k values of the list *l* to replace the k leaf values of the tree t, in the left to right order.

We assume, as Bird does, that sort is an 'a priori' given routine, whose performances have been already optimized. The definition of leaves and replace are as follows:

S2. leaves(tip(n)) = [n]

S3. leaves(t1^t2) = leveas(t1) @ leaves(t2)

S4. replace(tip(n),*l*) = tip(hd(*l*))

S5. replace(t1^t2,*l*) = replace(t1,*l*1) ^ replace(t2,*l*2) where <*l*1,*l*2>=cut(size(t1),*l*),

where: size(t) computes the number of leaves in a given tree t,

cut(k,*l*) produces from a given list *l* a pair of lists <*l*1,*l*2> such that *l*1@*l*2=*l* and length(*l*1)=k.

We assume that: $0 \le k \le \text{length}(l)$ holds for any call of cut(k,*l*), and
$\text{size}(t) \le \text{length}(l)$ holds for any call of replace(t,*l*).

For instance, size(tip(4)^tip(5))=2 and cut(4,[5,3,7,8,6])=<[5,3,7,8],[6]>. We have:

S6. size(tip(n)) = 1

S7. size(t1^t2) = size(t1) + size(t2)

S8. cut(0,*l*) = < [], *l* >

S9. cut(n+1,*l*) = < *l*1 @ [hd(*l*2)], tl(*l*2) > where <*l*1,*l*2>=cut(n,*l*).

For instance, if t ≡ ((6^3)(4^3))(1^(4^5)) (for simplicity we write n instead of tip(n)) then Sort-tree(t) ≡ ((1^3)(3^4))(4^(5^6)).

As usual, for deriving a one-pass algorithm from the equations S1-...-S9 we proceed by applying the composition strategy to the equation S1. Unfortunately it is impossible to make a folding step. We get, in fact:

replace(tip(n),sort(leaves(tip(n)))) = replace(tip(n),sort([n])) = tip(n)

replace(t1^t2,sort(leaves(t1^t2))) = replace(t1,*l*1) ^ replace(t2,*l*2) (♦)
where <*l*1,*l*2>=cut(size(t1), sort(leaves(t1^t2))),

and it is not possible to derive the recursive calls to Sort-tree(t1) and Sort-tree(t2). That fact is due to the existence of *relevant information* shared between the two recursive calls of replace. Therefore we may try to apply the abstraction strategy by defining: λx.replace(t,x).

Since λx.replace(t,x), sort(leaves(t)), and size(t) visit the same data structure t, the Tupling Strategy leads us to the definition of: Z(t) ≡ < λx.replace(t,x), sort(leaves(t)), size(t) >. We then have:

Z(tip(n)) = < λx.replace(tip(n),x), sort(leaves(tip(n))), size(tip(n)) >
= < λx.tip(hd(x)), [n], 1 >.

Z(t1^t2) = < λx.replace(t1^t2,x), sort(leaves(t1^t2)), size(t1^t2) >
= < λx.replace(t1,*l*1) ^ replace(t2,*l*2) where <*l*1,*l*2>=cut(size(t1), x),
merge(sort(leaves(t1)),sort(leaves(t2)), size(t1) + size(t2) >
= < λx.((A1 *l*1) ^ (B1 *l*2) where <*l*1,*l*2>=cut(A3, x)), merge(A2,B2), A3+B3 >
where <A1,A2,A3>=Z(t1) and <B1,B2,B3>=Z(t2)

HOSort-tree(t) = (A1 A2) where <A1,A2,A3>=Z(t).

In the definition of Z(t) merge(*l*1,*l*2) is regarded as a primitive function, and it gives us the ordered list which is the result of merging the two ordered list *l*1 and *l*2.

The above program HOSort-tree(t) is a one-pass program in the sense that Z(t1^t2) is defined in terms of Z(t1) and Z(t2) only. As in the Palindrome Example, during the visit of the tree t, information is collected in the components of Z(t), so that one visit only need to be made.
Computer experiments made in our VAX-VMS system using ML show that HOSort-tree(t) is faster than Sort-tree(t) for trees t of about size 30 or larger.

Let us now make a few comments on the comparison of our algorithm with the one in [Bir84]:
i) we have shown, as in the Palindrome Example, that lambda abstraction and tupling strategies are powerful enough to derive a one-pass algorithm, so that circular programs and local recursion are not necessary;
ii) Bird uses two functions, take(k,*l*) and drop(k,*l*), while we use the function cut(k,*l*) (see the equation S5). Their defining relationship is as follows: cut(k,*l*) = <take(k,*l*),drop(k,*l*)>.
A question may arise here: whether or not it would be possible to perform a derivation analogous to the one we have presented above, if we had the following equation S5' [Bir84], instead of S5:
S5'. replace(t1^t2,*l*) = replace(t1,take(k,*l*)) ^ replace(t2,drop(k,*l*)) where k=size(t1).

The answer is positive, and this fact shows again the power of our strategies. The derivation proceeds as follows. By composition strategy the above equation (♦) becomes:

replace(t1^t2,sort(leaves(t1^t2))) = replace(t1,take(size(t1),x)) ^ replace(t2,drop(size(t1),x))
where x=sort(leaves(t1^t2)). (♦♦)

Now we have to look for a folding of the recursive calls of replace. In order to do so, we may use a simple fact (similar to those used in [Bir84, p. 248]): for any list x, take(length(x),x) = x. We then have:
- take(size(t),sort(leaves(t))) = sort(leaves(t)), and therefore:
- if size(t1)+size(t2) = length(x) then drop(size(t1),x) = take(size(t2),drop(size(t1),x)).

Then we can rewrite the equation (♦♦) as follows:
replace(t1^t2,take(size(t1^t2),sort(leaves(t1^t2))))
= replace(t1,take(size(t1),x)) ^ replace(t2,take(size(t2),drop(size(t1),x)))
where x=sort(leaves(t1^t2)).
We are "almost" on the position of folding the recursive calls of replace, but we first need to apply the abstraction strategy.
i) replace(t1,take(size(t1),x)) where x=sort(leaves(t1^t2)) is lambda-abstracted to:

((λx.replace(t1,take(size(t1),x))) sort(leaves(t1^t2))), and

ii) replace(t2,take(size(t2),drop(size(t1),x))) where x=sort(leaves(t1^t2))
is lambda-abstracted twice, and we get:

(λx.((λy.replace(t2,take(size(t2),y))) drop(size(t1),x)) sort(leaves(t1^t2))).

Therefore, since we want to visit the tree t1 only once, by tupling strategy we are prompted to tuple the following three functions together because they all visit the same tree t1:

Z*(t) ≡ < λx.replace(t,take(size(t),x)), λx.drop(size(t),x), sort(leaves(t)) >.

After a few unfolding steps we have:

Z*(tip(n)) = < λx.tip(hd(x)), λx.tl(x), [n] >

Z*(t1^t2) = < λx.(replace(t1,take(size(t1),x)) ^ replace(t2,take(size(t2),drop(size(t1),x)))),
λx.drop(size(t1)+size(t2),x), merge(sort(leaves(t1)), sort(leaves(t2))) >
= < λx.((A1 x) ^ (B1 (A2 x))), λx.(A2 (B2 x)), merge(A3,B3) >
where <A1,A2,A3>=Z*(t1) and <B1,B2,B3>=Z*(t2).

HOSort-tree*(t) = (A1 A3) where <A1,A2,A3>=Z*(t).

The last folding step is based on the fact that:

$\lambda x.drop(size(t1)+size(t2),x) = ((\lambda x.drop(size(t1),x))\ (\lambda y.drop(size(t2),y)))$.

As in the Palindrome Example the reader may check that using generalization only (and not lambda abstraction), the derivation process is not successful.

Notice that while Bird need intuition for the invention of the tuple function Z*(t), in our approach that function is derived as a straightforward application of the abstraction and tupling strategies by looking for a "forced folding" [Dar81].

To be more precise Bird uses a function which is not exactly Z*(t) because the third component is leaves(t) instead of sort(leaves(t)). (Indeed Bird uses the two-arguments function leaves(t,*l*) instead of leaves(t) for avoiding the expensive concatenation operation occurring in S3 in favour of a 'cons' operation). That decision derives from the fact that he chooses the function 'sort' as a primitive function, while we choose the function 'merge'. Our choice allows us to perform the sorting of the leaves while the visit of the input tree is in progress.

With reference to [Bir84] let us finally notice that:
i) as in Bird's solution our program in terms of Z(t) (and also the one in terms of Z*(t)) computes some components of the tupled functions which are never used, and
ii) the nested applications (B1 (A2 x)) and (A2 (B2 x)) in the first and second components of Z*(t1^t2) correspond to the double recursion of the function 'repnd'.

As usual, in our approach we do not need to perform any termination analysis, because we have it for free by structural induction of the function definitions.

4. RELATING LAMBDA ABSTRACTION TO OTHER DERIVATION STRATEGIES

1) *Lambda Abstraction and Composition Strategy.*

Often the Composition Strategy (also called Specialization Strategy [Sch81]) is used for eliminating the construction of intermediate data structures which are to be passed from one function to another. A typical example is when we have to compute $h(x)\equiv f(g(x))$ and the result of f(y) only depends on some 'limited' portion of the input y. In that case we do not need to construct the entire data structure g(x), and we are able to save memory [Wad85].

However, that improvement is often possible only if we can derive a recursive definition for h(x). For that purpose we need to make a folding step, and it may happen that the expression of h(x) is such that folding is impossible. In that case the strategies one can apply are essentially:

- strategies which modify the structure of the expression of h(x) by 'promoting' constants to variables (generalization strategies) so that the matching may be successful, and
- strategies which make h(x) to equal to the result of applying a lambda expression, that is:

  h(x) = (λz.e) e', where x occurs free in e or e'. The lambda abstraction λz.e allows the insertion of variables for subexpressions, namely z for e', and it also allows folding steps otherwise

impossible, because one can deal only with the subexpression λz.e instead of the whole expression h(x).

A problem with the Abstraction Strategy is that while it allows for folding, one cannot know in advance the structure of the bound arguments, because they occur as simple variables, and one may not take advantage of their properties for further efficiency improvements. Therefore, the Abstraction Strategy may be considered as opposite to the Composition Strategy where indeed one takes advantage of the properties of the arguments.

The Composition Strategy allows high memory and time perfomances when one can make folding steps and obtain recursive definitions. However, if folding steps are not possible then lambda abstractions steps are to be performed first.

By the Abstraction Strategy we tend to use more memory cells at run time, because we should reserve space for bound variables. But it is possible to save computation time, because data structures are visited less often. For that same reason, in fact, one can release the memory used by the data as soon as it has been visited for the last time.

2) *Lambda Abstraction and Partial Evaluation.*

The relationship between the two techniques can be viewed in the light of the discussion of the previous point. Indeed, the Partial Evaluation approach to program development and program construction includes also the Composition and Specialization Strategies [Sch81].

Partial Evaluation allows us to 'insert' extra information into the functions to be evaluated, so that at compile time we may derive more efficient code. The Lambda Abstraction approach is related to partial evaluation in the sense that it allows us to 'withdraw' information from function definitions, so that they may have enough parameters, and therefore allow (by folding) the derivation of improved program versions. Only at later stages we can 'reinsert' information into the function definitions and that reinsertion will take place when we apply those functions to their actual arguments.

The basic idea is that program manipulation and program derivation, in general, cannot proceed only by specialization of procedures, but it is often the case that in order to derive efficient programs, we need to withdraw the information concerning the structure of the arguments. This is basically what lambda abstraction does for us when it transforms the expression "e[e1]" (that is, "e" where the subexpression "e1" occurs one or more times), into the expression "(λx.e) e1" where "e" is obtained from "e[e1]" by replacing the occurrences of e1 by x.

An example of these ideas is given by the equation (#) of the Palindrome program. In that equation we withdraw from the expression "a=hd(rev(*l*) @ [a]) and eqlist(*l*, tl(rev(*l*) @ [a]))" the information that the initial part of the arguments of hd and tl is indeed the reverse of a list.

At later stages we reinsert the withdrawn information. We do so when that information is relevant and it may play a crucial role for efficiency improvements. This is done by the application of (λx.e) to e1 and by the subsequent substitutions evoked by β-reductions. In particular, in the Palindrome program for a list *l*, we reinserted the information concerning the initial part of the arguments of hd and tl only at the end of the construction of Q(*l*)=<u,v> when computing the application (u v).

If we view the Partial Evaluation as a function Part: Prog×Data → Prog such that: Sem(p,(d1,d2)) = Sem(Part(p,d1),d2) [Esh82], then the Abstraction Strategy may be viewed as a sort of *inverse*: indeed, for that strategy we may think of a function Lambda: Prog → Prog×Data such that: Sem(p)=Sem((λx.p1) d) for some data d occurring in p.
The examples we have given in the previous Sections indicate that there is scope in program development for such 'inverse-partial-evaluation'.

3) *Lambda Abstraction and Higher Order Functions.*

Lambda abstraction may generate functions which return functions as values. Sometimes however, the composition which could follows the abstraction step, eliminates higher order objects. An example of that possibility is given in [PeS87]. In that paper one can see that through function composition, higher order expressions involved in the definition of a one-pass toy compiler, vanish in favour of ground type values.

5. TOWARDS A THEORY OF STRATEGIES

As suggested by various people (see for instance [Ers82]), we may consider the problem of program development in the framework of the following equation:

i) Sem(Specs) = Sem(Prog),

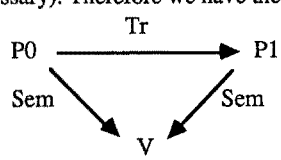
meaning that given a specification Specs we should produce a program Prog with the same semantics.

Actually, we may think that Specs itself is a preliminary version of the desired program which can be executed in some machine. In this case we may want to produce a program which is more efficient than the initial one. Therefore, if we are given a cost function C for executing programs (that is, applying Sem to its argument) we also want:

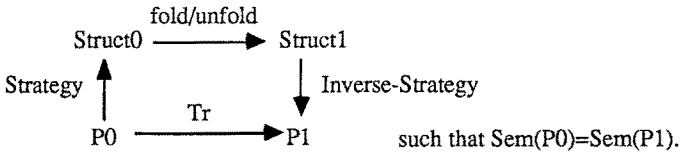
ii) C(Sem, Prog) ≤ C(Sem, Specs).

It would be very good to have a transformation function Tr such that: Prog=Tr(Specs) and the above conditions i) and ii) hold. We cannot hope to find for any given cost function C a universal function Tr, satisfying i) and ii), but we can provide programming tools so that the task of finding Prog from Specs can be simplified, at least in some cases.

Let us now assume that: i) Specs is a program P0, that is, a preliminary executable version of the desired program, and ii) Prog is a program P1, that is, an improved derived version. We also assume that the rules for transforming programs are essentially the fold/unfold rules [BuD77]. Those rules preserve partial correctness only with respect to the semantics function, but we will require that they preserve total correctness as well. (For that purpose it is enough to show the termination of the derived program P1, when necessary). Therefore we have the following commutative diagram:

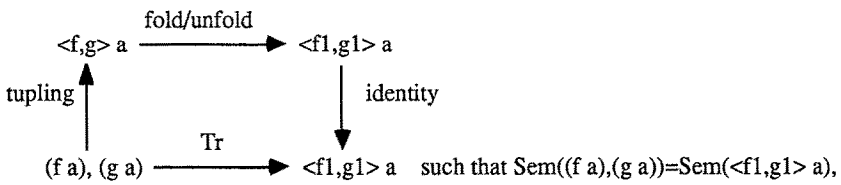


The problem of deriving P1 from P0 amounts to the construction of two program structures: Struct0 and Struct1, and three program transformations: Strategy, fold/unfold, Inverse-Strategy, as specified by the commutative diagram:



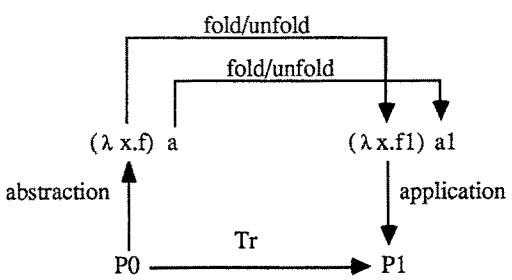
The construction of the above structures is often suggested by some *isomorphisms of data.* This is actually the case for the Tupling Strategy and the Abstraction Strategy.

The Tupling Strategy, in fact, is based on the isomorphism: $(A\rightarrow B)\times(A\rightarrow C) \cong A\rightarrow(B\times C)$. This means that the corresponding diagram is of the form:



and time×memory requirements for the evaluation of ((f a), (g a)) may be reduced because the given data 'a' is traversed only once while computing (<f1,g1> a).

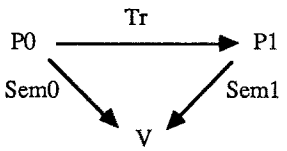
The Lambda Abstraction Strategy we have introduced in this paper, is based on the isomorphism: $((A\times B)\rightarrow C) \cong (A\rightarrow(B\rightarrow C))$ and it can be represented by the following diagram:



Notice that the strategies "come in pairs", that is, we need to find both an ascending arrow 'Strategy' and a descending arrow 'Inverse-Strategy' for our commutative diagrams. Isomorphisms of data domains give us those pairs in a simple and straightforward way.

Notice also that the two objects P0 and P1 in the lower sides of the diagrams we have presented above, need not be identical, but they must have the same semantics. Therefore, the class of morphisms to be considered for the invention of strategies includes, but it is not restricted to, isomorphisms of domains. Obviously, the application of strategies and rules should also allow better performances, as specified by the above condition ii).

It may also be the case that the transformation from program P0 to program P1 can be done by *language extensions* or *language refinements*. It means that we are given two different semantic functions, say Sem0 and Sem1, and the required commutative diagram is the following:



where, as usual, C(Sem1,P1) ≤ C(Sem0,P0). In that case we may get better performances by adopting a more efficient evaluator, namely the one for Sem1 (instead of the one for Sem0). We will not analyze in more detail this issue here.

5. CONCLUSIONS

We have studied the problem of deriving efficient programs which avoid multiple visits of data structures. We have devised a strategy, called Lambda Abstraction Strategy, which achieves the goal and it does not require lazy evaluation and circular programs, as suggested in [Bir84]. We have analyzed the relation of that strategy with respect to other known techniques for program derivation, like the Specialization technique.

We have also proposed a unifying way of considering various strategies for program development, and in particular, we have looked at the *isomorphism of data domains*. That *algebraic* view is a complementary approach to the *deductive* view which have been considered in [ScS85], for instance. In an intuitionistic framework that result does not come as a surprise, but from a methodological point of view it may offer to programmers deeper insights into the program development process.

6. AKNOWLEDGEMENTS

Many thanks to R. Bird who has been for me of great stimulation and encouragement. My gratitude also goes to N. Jones, R. Paige, A. Skowron, D. Swierstra, and the friends of the IFIP WG. 2.1. The referee's suggestions were very valuable.

This work was supported by the IASI Institute of the National Research Council of Rome (Italy).

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).

[Bac78] Backus, J.: "Can Programming be Liberated from the Von Neumann Style?" Comm. A.C.M. 21(8) (1978), pp. 613-641.

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

[Bir86] Bird, R.S.: "An Introduction to the Theory of Lists" Technical Monograph PRG-56, Oxford University Computing Laboratory, Oxford, England (1986).

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

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

[ChM79] Chirica, L.M. and Martin, D.F.: "An Order-Algebraic Definition of Knuthian Semantics" Mathematical System Theory 13, 1-27 (1979), pp. 1-27.

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

[Ers82] Ershov, A. P.: "Mixed Computation: Potential Applications and Problems for Study" Theoretical Computer Science Vol.18, (1982), pp. 41-67.

[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).

[JøS86] Jørring, U. and Scherlis, W. L.: "Compilers and Staging Transformations" Thirteenth Annual A.C.M. Symposium on Principles of Programming Languages, St. Petersburgh Beach, Florida, USA (1986), pp. 86-96.

[Myc81] Mycroft, A.: "Abstract Interpretations and Optimising Transformations for Applicative Programs" Ph.D. Thesis, Computer Science Department, University of Edinburgh, Scotland (1981).

[Pet78] Pettorossi, A.: "Improving Memory Utilization in Transforming Recursive Programs" Proc. MFCS 78, Zakopane (Poland), Lecture Notes in Computer Science n.64, Springer Verlag (1978), pp. 416-425.

[Pet84] Pettorossi, A.: "A Powerful Strategy for Deriving Efficient Programs by Transformation" ACM Symposium on Lisp and Functional Programming, Austin, Texas (1984), pp.273-281.

[PeS87] Pettorossi, A. and A. Skowron: "Higher Order Generalization in Program Derivation" Proc. Intern. Joint Conference on Theory and Practice of Software Development. Pisa, Italy. Lecture Notes in Computer Science n. 250, Springer-Verlag (1987), pp. 182-196.

[Sch81] Scherlis, W. L.: "Program Improvement by Internal Specialization" A.C.M. Eighth Symposium on Principles of Programming Languages, (1981), pp. 146-160.

[ScS85] Scherlis, W. L., and Scott, D.: "Semantical Based Programming Tools" Mathematical Foundations of Software Development, TAPSOFT '85, Lecture Notes in Computer Science n. 185, Springer-Verlag, Berlin (1985), pp. 52-59.

[Swi85] Swierstra, D.: "Communication 513 SAU-15". IFIP WG.2.1, Sausalito, California, USA (1985).

[Tak87] Takeichi, M.: "Partial Parametrization Eliminates Multiple Traversals of Data Structures" Acta Informatica 24, (1987), pp.57-77.

[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).