Efficiency of Difference-List Programming

Ulrich Geske[1], Hans-Joachim Goltz[2]

[1]University of Potsdam
ugeske@uni-potsdam.de

[2]Fraunhofer FIRST, Berlin
goltz@first.fraunhofer.de

Abstract. The difference-list technique is described in literature as effective method for extending lists to the right without using calls of append/3. There exist some proposals for automatic transformation of list programs into difference-list programs. However, we are interested in construction of difference-list programs by the programmer, avoiding the need of a transformation step. In [GG09] it was demonstrated, how left-recursive procedures with a dangling call of append/3 can be transformed into right-recursion using the unfolding technique. For simplification of writing difference-list programs using a new cons/2 procedure was introduced. In the present paper, we investigate how efficieny is influenced using cons/2. We measure the efficiency of procedures using accumulator technique, cons/2, DCG's, and difference lists and compute the resulting speedup in respect to the simple procedure definition using append/3. Four Prolog systems were investigated and we found different behaviour concerning the speedup by difference lists. A result of our investigations is, that an often advice given in the literature for avoiding calls append/3 could not be confirmed in this strong formulation.

1 Introduction

Appending an additional element E as last element to an existing list could be performed by copying all existing list elements and the additional element E into a new list (using the Prolog procedure append/3).. Instead of, this operation could be performed by an efficient (physical) concatenation using the difference list notation. Every list may be presented as a difference list. For example, the list `[1,2,3]` could be represented as difference of the lists `[1,2,3|X]`and `X.` If list X contains E as first element (e.g. X=[E|Y]), E is the next element after 3 without a copying operation). A term [E|Y] is called an incomplete list. The Prolog standard does not provide any special notation for difference lists. A possible notation of a difference list from two lists `L` and `R` may be given by a notation `L⊕R`, e.g. `L-R` or `L\R` (the symbol used must be defined as oparator in the concrete Prolog system). If ⊕ denotes a comma (,), L and R are two arguments in an argument list. The earliest extended description of difference lists was given by Clark and Tärnlund in [CT77]. A concatenation of the difference lists `U⊕V=[1,2,3|X]⊕X` and `V⊕W>=X⊕nil` results in the difference list `U⊕W=[1,2,3,4]⊕nil≡[1,2,3,4]` as soon as `X` is computed to

4⊕nil. The difference list notation is a syntactic variant of the accumulator technique (a comma is used for ⊕). While in the accumulator technique accumulator and result parameter are separated into two terms, which needs two variables for accessing them, in the difference list notation both information are accessible by one variable with the advantage of an easy to survey structure of procedures.

Our analysis of presentations of the use of difference list in Prolog textbooks showed that this technique is often not adequately explained [see also GG09]. Especially, a clear and convincing rule, where in procedures to specify the incomplete list (e.g. [E|Y]) is not supplied. Dependent from a concrete problem, the incomplete list has to be specified in the head of a rule, in one of the calls of the body or in the last (recursive) call of the body of a rule. A solution of this problem could be the use of the paradigm of grammar rules or its extension, Definite Clause Grammar (DCG), which was originally developed for language processing but may be used for list processing, too. Natural language sentences, coded as list of words, must be processed phrase by phrase from left to right, consuming some words for a phrase and leaving the rest for the following phrases. In the DCG formalism it is sufficient to specify the sequence of phrases. The argument pattern for traversing the list will be generated automatically in accumulator technique.

```
%DCG-Specification for copying a list
dcg_copy([]) --> [].
dcg_copy([X|RR]) --> [X], dcg_copy(RR).

%Generated program by automatic program transformation
dcg_copy([],L,L).
dcg_copy([X|Xs],Acc,Res):-
  'C'(Res,X,RR),
  dcg_copy(Xs,Acc,RR).
```

Fig. 1 Copying a list as DCG specification

An specification example for DCG's is copying a list (Fig. 1). The specification [X] means, taking the first element X from a list [X|Xs]. This specification is transformed into a call of the built-in procedure 'C'/3, which could be defined as Prolog procedure by 'C'([X|Xs],X,Xs).

For supporting teaching and application of difference list programming, we have proposed a procedure cons(X, [X|Xs]-Xs) in [GG09] which takes pattern from the 'C'/3 procedure. The advantages of cons/2 are both, its difference-list format, and that the definition of cons/2 could be added to each Prolog system while 'C'/3 and the DCG formalism are not part of the ISO-Prolog until now and are not available in each system. The use of cons/2 allows the application of a simple rule for writing explicit difference-programs and to find out simply the right place for the mentioned incomplete list [E|Y]. In this paper we investigate the efficiency applying cons/2 compared with other kinds of specification.

2 Top-down and Bottom-up Construction of Lists

The notions of top-down and bottom-up procedures for traversing structures like trees are well established. We will use the notions top-down construction of lists and bottom-up construction of lists in this paper to describe the result of the process building lists with a certain order of elements in relation to the order in which the elements are taken from the corresponding input list.

Top-down construction of lists

The order el_1' – el_2' of two arbitrary elements el_1', el_2' in the constructed list corresponds to the order in which the two elements el_1, el_2 are taken from the input term (perhaps a list or a tree structure).

Bottom-up construction of lists

The order el_2' – el_1' of two arbitrary elements el_1', el_2' in the constructed list corresponds to the reverse order in which the two elements el_1, el_2 are taken from the input term (perhaps a list or a tree structure).

An input list may be, e.g., [2, 4, 3, 1]. A top-down construction of the result list [2, 4, 3, 1] is given if the elements are taken from left to right from the input list and put into the constructed list in a left to right manner. If the elements are taken from the input list by their magnitude and put into the result list from left to right, the list [1 2 3 4] will be (top-down) constructed. A bottom-up construction of the result list [1, 3, 4, 2] is given if the elements of the list [2, 4,3,1] are taken from left to right from the input list and put into the constructed list in a right-to-left manner. If the elements are taken from the input list by their magnitude and put into the result list from right to left, the list [4 3 2 1] will be (bottom-up) constructed. Which programming techniques could be used for a top-down- respectively a bottom-up construction of lists? Accumulator technique is an often used technique, which allows both, top-down- and bottom-up construction of lists. Examples are the procedures for traversing in pre-order manner accapp_pre_td/3 and accapp_pre_bu/3 (Fig. 5).. These procedures use besides accumulators calls of append/3. But, also without use of accumulators, top-down and bottom-up list-constructions are possible. Examples are pre_order/2 and pre_order_bu/2 (see also Fig. 5). Again, calls of append/3 are needed in these definitions.

3 Construction Rules for difference list procedures

There are different possibilities to avoid the use of a call of append/3. A rather trivial improvement is given by unfolding an append/3 call which puts a single element X in front of a list A to give the result list RR, i.e RR=[X|A]. E.g., the append-free procedures acc_pre_td/3 result (Fig. 2), if in accapp_pre_td/3 the equivalent [X|L1] for Xs is inserted and the corresponding call of `append([X],L1,Xs)` is crossed.
The difference-list procedure dl_pre_td/2 (see Fig. 5) is a syntactic variant of the accumulator version of the corresponding procedure acc_pre_td/3, which results by substitution of the second and third argument, say ARG2 and ARG3 (ARG2 should be

the accumulator parameter, ARG3 the result parameter) with the difference list ARG3-ARG2.

```
acc_pre_td(tree(X,L,R),Rs, [X|L1]):-
       acc_pre_td(L, L2, L1),
       acc_pre_td(R, Rs, L2).
acc_pre_td([],L,L).
```

Fig. 2 Append-free pre-order tree-traversal by accumulator technique

The presented transformation steps for a procedure into difference-list notation has a serious disadvantage: the starting point is a procedure definition which uses an append/3-call (which should be avoided). Moreover, the precondition for the transformation, an admissible call of append/3, is not always given as examples pre_order_bu/3 (Fig. 5) shows. An admissible structure is given if a single element should be put in front of a list, i.e. this element is part of the first argument of the call of append/3. The fulfilment of this condition can not be always ensured. Therefore append/3 calls must be avoided at all. An alternative, we propose, is the use of the cons/2 procedure. The definition of cons/2 is choosen to suit the syntactic format of the difference list notation.

```
cons(Element,[Element|Rest]-Rest).
```

Fig. 3 Definition of the cons operation

The advantage of this format is its correspondence to the format which is needed for composing a resulting difference list from its parts. The position of the call of cons/2 in the body of a procedure is irrelevant, in general, but if this call occurs at its "natural" position a formal guideline for constructing difference list is possible.

Top-down construction of a difference list Res-Acc results from the "natural" order of sub-difference-lists **Res**-Temp1, ..., TempI-TempI+1, ..., TempN-**Acc**.

Bottom-up construction of a difference list Res-Acc results from the "natural" order of sub-difference-lists Temp1-**Acc**, ...TempI-1 - TempI, ..., **Res**-TempN

Rule 1 Informal rules for top-down- and bottom-up composition of difference lists

4 Benchmarks

4.1 Benchmark tests - processing trees

There exist, corresponding to [Sterling-Shapiro86], three different possibilities for the linear traversal of trees. Any node X in a binary tree, besides the leave nodes, has a *Left* and a *Right* successor tree. A pre-order traversal visits the tree in the following order: X, Left Right, which may be programmed using a call append([X|Left], Right, Tree) (pre_order/2 in Fig. 5). Correspondingly, an in-order traversal is given by the

call append(Left, [X|Right],Tree) and a post-order traversal by the sequence of calls append(Right,[X],T), append(Left,T,Tree). A concrete example is shown in Fig. 13.

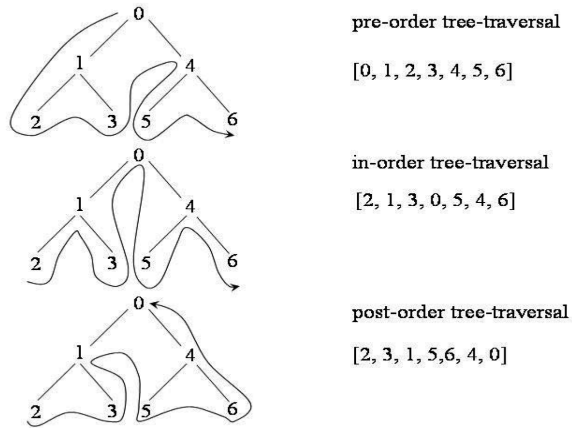

Fig. 4 Examples for linear traversals of binary trees

Fig. 4 presents four different algorithms for the pre-order-functions. A naive algorithm uses the append procedure to compose the result from the different parts. An extension of this algorithm is the additional use of an accumulator with the advantage that the first argument of the append/3-call is always a list of one element – there is no danger for looping forever. A further improvement is the avoidance of append/3-calls at all by substituting it by the new introduced cons/2 procedure. Finally, unfolding of the call of cons/2 leads to the known difference list format. A procedure in a difference-list format could be derived step-by-step, as explained or it may be specified in one step as described in the following for pre-order tree-traversal. *Specification of a pre-order traversal*: The result of a pre-order traversal is the difference-list L-LN. In a top-down construction of the result, the node X of the structure tree(X,Left,Right) is visited first and supplies the difference-list L-L1, the traversal of the left subtree supplies the difference-list L1-L2, and the traversal of the right subtree supplies L2-LN. In a bottom-down construction of the result, the node X of the structure tree(X,Left,Right) is visited first and supplies the difference-list L2-LN, the traversal of the left subtree supplies the difference-list L1-L2, and the traversal of the right subtree supplies L-L1. The in-order and post-order traversals are specified analogous (see also Fig. 6, Fig. 7).

For processing the procedures a tree of a certain depth is automatically generated by a call of the procedure

```
binary_tree([],_,0).
binary_tree(tree(LR-T,Left,Right),LR,T) :-
   T1 is T-1,
   binary_tree(Left,LR-l,T1),
   binary_tree(Right,LR-r,T1).
```

<table>
<tr><th colspan="2">Definitions for pre-order tree-traversal</th></tr>
<tr><th>Top-down construction of result</th><th>Bottom-up construction of result</th></tr>
<tr><td>%use of append/3; accumulator-free
pre_order(tree(X,L,R), Xs) :-
pre_order(L,LN),
pre_order(R,L0),
append([X|LN],L0,Xs).
pre_order([],[]).</td><td>%use of append/3; accumulator-free
pre_order_bu(tree(X,L,R), Xs) :-
pre_order_bu(L,LN),
pre_order_bu(R,L0),
append(LN,[X],L1),
append(L0,L1,Xs).
pre_order_bu([],[]).</td></tr>
<tr><td>%use of append/3;use of accumulator
accapp_pre_td(tree(X,L,R),L0,LN):-
append([X],L1,LN),
accapp_pre_td(L,L2,L1),
accapp_pre_td(R,L0,L2).
accapp_pre_td([],L,L).</td><td>%use of append/3;use of accumulator
accapp_pre_bu(tree(X,L,R), L0,LN) :-
append([X],L0,L2).
accapp_pre_bu(L,L2, L1),
accapp_pre_bu(R,L1,LN).
accapp_pre_bu([],L,L).</td></tr>
<tr><td>%use of cons/2
%use of accumulator (part of DL)
d_pre_td(tree(X,L,R),LN-L0):-
/*LN=[X|L1]*/ cons(X,LN-L1),
d_pre_td(L, L1-L2),
d_pre_td(R, L2-L0).
d_pre_td([],L-L).</td><td>%use of cons/2
%use of accumulator (part of DL)
d_pre_bu(tree(X,L,R),LN-L0):-
/*L2=[X|L0]*/ cons(X, L2-L0),
d_pre_bu(L, L1-L2),
d_pre_bu(R, LN-L1).
d_pre_bu([],L-L).),</td></tr>
<tr><td>%call of cons/2 unfolded
%use of accumulator (part of DL)
dl_pre_td(tree(X,L,R),[X|L1]-L0):-
dl_pre_td(L,L1 -L2),
dl_pre_td(R,L2 -L0).
dl_pre_td([],L-L).</td><td>%call of cons/2 unfolded
%use of accumulator (part of DL)
dl_pre_bu(tree(X,L,R),LN-L0):-
dl_pre_bu(L, L1-[X|L0]),
dl_pre_bu(R, LN- L1).
dl_pre_bu([],L-L).</td></tr>
</table>

Fig. 5 Different procedure definitions for pre-order tree-traversal

<table>
<tr><th colspan="2">Definitions for in-order tree-traversal</th></tr>
<tr><th>Top-down construction of result</th><th>Bottom-up construction of result</th></tr>
<tr><td>%use of append/3; no accumulator
in_order(tree(X,L,R), Xs) :-
in_order(L,LN),
append(LN,[X|L0],Xs),
in_order(R,L0).
in_order([],[]).</td><td>%use of append/3; no accumulator
in_order_bu(tree(X,L,R), Xs) :-
in_order_bu(L,LN),
in_order_bu(R,L0),
append(L0,[X|LN],Xs).
in_order_bu([],[]).</td></tr>
<tr><td>%use of append/3;use of accumulator
accapp_in_td(tree(X,L,R), L0,LN) :-
accapp_in_td(L,L1,LN),
append([X],L2,L1),
accapp_in_td(R,L0,L2).
accapp_in_td([],L,L).</td><td>%use of append/3;use of accumulator
accapp_in_bu(tree(X,L,R), L0,LN) :-
accapp_in_bu(L,L0,L2),
append([X],L2,L1),
accapp_in_bu(R,L1,LN).
accapp_in_bu([],L,L).</td></tr>
<tr><td>%use of cons/2;accumulator (in DL)
d_in_td(tree(X,L,R),LN-L0):-
d_in_td(L, LN-L1),
/*L1=[X|L2]*/ cons(X,L1-L2),
d_in_td(R, L2-L0).
d_in_td([],L-L).</td><td>%use of cons/2
%use of accumulator (part of DL)
d_in_bu(tree(X,L,R),LN-L0) :-
d_in_bu(L, L2-L0),
/*L1=[X|l2]*/ cons(X,L1-L2),
d_in_bu(R, LN-L1).
d_in_bu([],L-L).),</td></tr>
<tr><td>%cons/2 unfolded; accumulator(in DL)
dl_in_td(tree(X,L,R),LN-L0):-
dl_in_td(L,LN-[X|L2]),
dl_in_td(R,L2-L0).
dl_in_td([],L-L).</td><td>% cons/2 unfolded;accumulator(in DL)
dl_in_bu(tree(X,L,R),LN-L0):-
dl_in_bu(L, L2-L0),
dl_in_bu(R, LN-[X|L2]).
dl_in_bu([],L-L).</td></tr>
</table>

Fig. 6 Different procedure definitions for in-order tree-traversal

Definitions for post-order tree-traversal	
Top-down construction of result	Bottom-up construction of result

`%use of append/3; no accumulator`
`post_order(tree(X,L,R), Xs) :-`
`   post_order(L,LN),`
`   post_order(R,L0),`
`   append(L0,[X],L1),`
`   append(LN,L1,Xs).`
`post_order([],[]).`
	`%use of append/3; no accumulator`
	`post_order_bu(tree(X,L,R), Xs) :-`
	`   post_order_bu(L,LN),`
	`   post_order_bu(R,L0),`
	`   append([X],L0,L1),`
	`   append(L1,LN,Xs).`
	`post_order_bu([],[]).`

`%use of append/3 and accumulator`
`accapp_post_td(tree(X,L,R),L0,LN) :-`
`   accapp_post_td(L,L1,LN),`
`   accapp_post_td(R,L2,L1),`
`   append([X],L0,L2).`
`accapp_post_td([],L,L).`
	`%use of append/3 and accumulator`
	`accapp_post_bu(tree(X,L,R),L0,LN) :-`
	`   accapp_post_order_bu(L,L0,L2),`
	`   accapp_post_order_bu(R,L2,L1),`
	`   append([X],L1,LN).`
	`accapp_post_order_bu([],L,L).`

`%use of cons/2`
`%use of accumulator (part of DL)`
`d_post_td(tree(X,L,R),LN-L0):-`
`         d_post_td(L, LN-L1),`
`         d_post_td(R, L1-L2),`
`/*L2=[X|L0]*/  cons(X,L2-L0).`
`d_post_td([],L-L).`
	`%use of cons/2`
	`%use of accumulator (part of DL)`
	`d_post_bu(tree(X,L,R),  LN-L0):-`
	`         d_post_bu(L,   L2-L0),`
	`         d_post_bu(R,   L1-L2),`
	`/*LN=[X|L1]*/   cons(X, LN-L1).`
	`d_post_bu([],L-L). ),`

`%call of cons/2 unfolded`
`dl_post_td(tree(X,L,R),LN-L0):-`
`         dl_post_td(L, LN-L1),`
`         dl_post_td(R, L1-[X|L0]).`
`dl_post_td([],L-L).`
	`%call of cons/2 unfolded`
	`dl_post_bu(tree(X,L,R),[X|L1]-L0):-`
	`            dl_post_bu(L, L2-L0),`
	`            dl_post_bu(R, L1-L2).`
	`dl_post_bu([],L-L).`

Fig. 7 Different procedure definitions for post-order tree-traversal

and finally processed by an ordering processed by an ordering procedure, e.g. (measurement of cpu-time not included):

```
?- binary_tree(Tree,0,16),!,pre_order(Tree,L).
```

A depth of 16 for the tree was a good compromise for the used computer with 331 MHz processor takt rate and 192 MB memory concerning consumption of time and memory. Each benchmark test was repeated 10 times and the mean value was computed.
The investigated systems, in which the benchmark procedures were consulted (using `consult/1`), are CHIP version 5.8.0.0, ECLiPSe version 5.8 #95, SWI-Prolog version 5.6.64, and SICStus-Prolog version 4.07.

4.2 Speedup

The speedup is the relationship of the processing time of the (naive) accumulator-free version of a procedure by the processing time of the improved version for the same algorithm. In Table 1 the speedup is presented for each tree-order traversal (in-order, pre-order, post-order), each method (top-down (TD), bottom-up (BU)), and each algorithm (append/3+accu, cons/2, DCG's, difference lists) in each of the investigated Prolog system. For comparision purpose the naive programming method using calls

of the built-in append/3 without using an accumulator argument is used (procedures in_order/2, pre_order/2, post_order/2, in_order_bu/2, pre_order_bu/2, post_order_bu/2 – in Table 1 denoted by: append/3). The rows denoted by myappend/3 show the speedup with a user-defined append/3-procedure. The other classes of algorithms use an accumulator either as separate argument or as part of a difference list. The algorithm which complements the naive procedure by an accumulator is called append/3+accu in Table 1 (procedures accapp_in_td/2, accapp_pre_td/2, accapp_post_td/2, accapp_in_bu/2, accapp_pre_bu/2, accapp_post_bu/2). A substitution of append/3 by a call of the new procedure cons/2 leads to the class of algorithms which is called cons/2 in Table 1 (procedures d_in_td/2, d_pre_td/2, d_post_td/2, d_in_bu/2, d_pre_bu/2, d_post_bu/2).

Table 1: Speedup for TD/BU-in-/pre-/post-order traversal of a tree

Prolog system	CHIP 5.8		ECLiPSe 5.8		SWI 5.6.64		SICStus 4.07	
Construction	TD	BU	TD	BU	TD	BU	TD	BU
pre-order								
myappend/3	1.27	1.25	0.96	1.06	0.99	1.00	0.32	0.24
append/3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
append/3+accu	2.90	4.84	4.21	6.33	3.72	8.48	1.09	1.44
cons/2	3.90	6.10	3.24	5.15	4.00	8.70	0.96	1.23
DCG(‘C’/3)	4.21	6.74	5.35	8.77	4.74	10.58	0.93	1.22
DCG(-->/2)			5.30	8.51	6.24	11.73	1.22	1.58
difference lists	5.00	7.80	4.50	7.06	5.19	12.84	1.35	1.91
in-order								
myappend/3	1.22	1.22	0.97	1.06	1.01	0.99	0.36	0.34
append/3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
append/3+accu	2.51	2.56	2.90	3.85	2.41	2.45	1.08	0.99
cons/2	3.72	3.32	2.47	3.18	2.71	2.61	0.86	0.93
DCG(‘C’/3)	3.60	3.69	4.07	5.54	3.15	3.11	0.92	0.85
DCG(-->/2)			4.00	5.64	3.63	3.50	1.18	1.17
difference lists	4.29	4.25	3.61	4.66	3.98	3.73	1.48	1.42
post-order								
myappend/3	1.25	1.14	0.97	0.96	0.99	1.00	0.25	0.34
append/3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
append/3+accu	4.52	3.32	6.41	4.44	7.96	5.04	1.42	1.34
cons/2	5.78	4.25	5.20	3.57	8.16	5.37	1.23	1.14
DCG(‘C’/3)	6.57	4.50	8.43	5.65	10.48	6.45	1.21	1.16
DCG(-->/2)			8.68	5.69	12.05	8.57	1.47	1.53
difference lists	7.59	5.55	7.87	4.66	12.34	7.24	1.91	1.83

Finally, unfolding the call of cons/2 results in the procedures which form the class called “difference lists” in Table 1 (procedures dl_in_td/2, dl_pre_td/2, dl_post_td/2, dl_in_bu/2, dl_pre_bu/2, dl_post_bu/2). A possible class of procedures with an additional argument for the accumulator instead of combing it with the result parameter in a difference list is not considered here. It is a syntactic variant of the difference-list notation and is denoted in the literature as “accumulator” version.
Table 1 shows that each of the investigated Prolog systems behave in a different manner. But, roughly we may classify the systems into two groups. One group consists of the CHIP system and SWI-Prolog.. Each improvement in the algorithm (in the assumed order of improvement: append/3+accu, cons/2, difference list) mirrors in a partly significant speedup of the execution. Depending of the problem, the speedup of execution times of the method append/3+accu compared with the append/3 algorithm is between about 2.5 and 5 for CHIP and about 2.5 and 8.5 for SWI-Prolog. Application of the cons/2-algorithm gives a speedup of about 25% less the speedup of the difference list method (cf. also Table 2). The maximum speedup by the cons/2-method is about 8-fold compared with the execution time by the append/3-method. Both systems process obviously a built-in procedure and a user-defined procedures comparable fast (see also rows myappend/3 and append/3 in Table 1).
The second group of Prolog systems contains the systems ECLiPSe and SICStus-Prolog. The characteristic of this class is a missing strong connection between the assumed improvement given by the algorithms and the speedup of execution. The accumulator-procedures which contain a call of append/3 may have a greater speedup than the corresponding procedures which use a call of cons/2 instead of append/3. An explanation for this effect may be that calls of the built-in (compiled) procedure append/3 will be processed faster than calls of the user-procedure cons/2. The cons/2-algorithm supplies a speedup of about 30% less the speedup of the difference list method (see also Table 2). The maximum speedup by the cons/2 procedure in this group is 5.2. For SICStus-Prolog the maximum speedup of about 2 occurs for the difference list procedure.

Table 2: Average relative speedup for TD/BU-in-/pre-/post-order traversal of a tree

Prolog system	CHIP 5.8		ECLiPSe 6.0.82		SWI 5.6.64		SICStus 4.0/7	
Construction	TD	BU	TD	BU	TD	BU	TD	BU
append/3	0.33		0.51		0.36		0.69	
append/3+accu	0.60		0.88		0.66		0.74	
cons/2	0.77		0.72		0.72		0.65	
DCG(‘C’/3)	0.85		1.18		0.85		0.64	
DCG(-->/2)			1.19		1.02		0.82	
difference list	1.00		1.00		1.00		1.00	

In these tests the minimum measured speedup of the algorithms append/3+accu and cons/2 reaches 58% of the speedup by difference lists (TD-pre-order with CHIP and TD-in-order in SICStus-Prolog). This result may be important for programming practice. DCG’s algorithms are able to perform procedure execution more efficient than difference list algorithms (Table 2).

5 Summary and Future Work

We have proposed simple, schematic rules for using difference lists. Our rule generalizes both bottom-up construction of lists using accumulators and top-down construction of lists using calls to append/3 to the notion of difference list. The introduction of the cons/2 operation serves as a didactic means to facilitate and simplify the use of difference lists. This operation could easily be removed from the procedures by an unfolding operation.

The benchmark tests demonstrate that the gain concerning the speedup depends from the used Prolog system. Speedup factors of 1.35 minimum to 12.84 maximum could be found for the same traversal order (pre-order) in different systems by using difference lists instead of the naive algorithm with calls of append/3. SWI-Prolog supplies for the procedures of the benchmark tests a maximum speedup of about 12, for CHIP-system this figure is about 8, for ECLiPSe a maximum speedup of about 2 results, and the for SICStus system the maximum speedup is about 2. The highest possible speedup occur when difference lists or DCG's are used. A reasonable speedup occurs when a call of cons/2 is used, with the advantage that such a procedure is easier to read and to maintain. Because of considerable high speedup values for the append+accu algorithm the often given advice "calls of append/3 should be avoided" should be substituted by "try using accumulators as often as possible".

A comparison of the efficienies of the difference-list algorithm and the DCG algorithm leads to the assumption which is to verify yet that a compiled version of the proposed cons/2 procedure will improve the efficieny significant.

References

[AFSV00] Albert, E.; Ferri, C.; Steiner, F.; Vidal, G.: Improving Functional Logic-Programs by difference-lists. In He, J.; Sato, M.: Advances in Computing Sciece – ASIAN 2000. LNCS 1961. pp 237-254. 2000.

[CT77] Clark, K.L.; Tärnlund, S,Å: A First Order Theory of Data and Programs. In: Inf. Proc. (B. Gilchrist, ed.), North Holland, pp. 939-944, 1977.

[GG09] Geske, U.; Goltz, H.-J.: A guide for manual construction of difference-list procedures. In: Seipel, D.; Hanus, M.; Wolf, A. (eds): Applications of Declarative Programming and Knowledge Management, Springer-Verlag, LNAI 5437, pp 1-20, 2009.

[MS88] Marriott, K.; Søndergaard, H.: Prolog Transformation by Introduction of Difference-Lists. TR 88/14. Dept. CS, The Univ. of Melbourne, 1988.

[MS93] Marriott, K.; Søndergaard, H.: Prolog Difference-list transformation for Prolog. New Generation Computing, 11 (1993), pp. 125-157, 1993.

[SS86] Sterling, L; Shapiro, E.: The Art of Prolog. The MIT Press, 1986. Seventh printing, 1991.

[ZG88] Zhang, J.; Grant, P.W.: An automatic difference-list transformation algorithm for Prolog. In: Kodratoff, Y. (ed.): Proc. 1988 European Conf. Artificial Intelligence. pp. 320-325. Pittman, 1988.