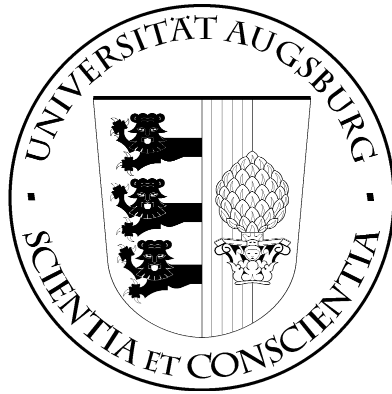
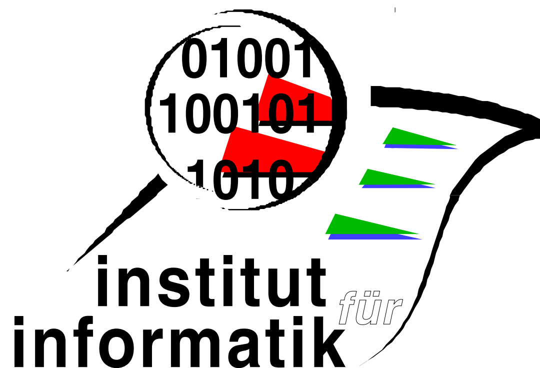
Universität Augsburg



Safer Ways to Pointer Manipulation

Bernhard Möller

Report 2000-4 Mai 2000



Institut für Informatik
D-86135 Augsburg

Copyright © Bernhard Möller
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

Safer Ways to Pointer Manipulation

Bernhard Möller

Institut für Informatik, Universität Augsburg
D-86154 Augsburg, Germany
e-mail: moeller@informatik.uni-augsburg.de

Abstract. We present a technique for passing in a safe way from high-level specifications of algorithms on pointer structures in a semi-functional style to efficient low-level implementations involving address calculations.

1 Introduction

This article was motivated by an experience I had while teaching a course on the programming language C [2]. I was trying to write down an algorithm for deleting an element from a singly linked list; this algorithm should not involve trailing auxiliary pointers and the like, but rather work on a single address variable. I failed several times in formulating this algorithm straightforwardly, always somehow mixing up the levels of dereferencing and the like. So I thought of more systematic ways of constructing such an algorithm and remembered my transformational background [1]. Now I first wrote down a simple recursive specification in a semi-functional style, passed to a corresponding procedure and applied the standard technique for recursion removal. The whole thing took less than ten minutes, and I was left with a very nice and correct algorithm of the desired characteristics. I think that this way of proceeding applies to a large class of pointer algorithms and therefore I want to report on it.

2 A General Transformation Scheme

We start with the well-known transition from a tail-recursive procedure to one with a loop in its body. Suppose procedure `p` is defined as

```
void p (type x)
     { if (cond)
            p(exp) ;
       else stat ;
     }
```

where `type` is some type that does not start with `*`, `cond` is an expression of type `int`, `exp` is an expression of type `type` and `stat` is a statement. Then the definition of `p` can equivalently be replaced by

```
void p (type x)
     { while (cond)
          x = exp ;
       stat ;
     }
```

For the case of a star type somewhat more care has to be taken to capture side effects on the variables whose addresses are passed as parameters. Here the recursive version

```
void p (type *x)
     { if (cond)
          p(exp) ;
       else stat ;
     }
```

is equivalent to

```
void p (type *y)
     { type *x = y ;
       while (cond)
          x = exp ;
       stat ;
     }
```

These rules generalize in a straightforward way to the case of several parameters if we use a feature which is not part of official C, viz. a collective assignment of the form

$$(\mathtt{x}_1, \ldots, \mathtt{x}_n) = (\mathtt{exp}_1, \ldots, \mathtt{exp}_n) \ .$$

Its semantics is that the tuple of expressions on the right hand side is evaluated in arbitrary order and the resulting values are assigned *simultaneously* to the variables on the left hand side. No variable may occur twice in the left hand side tuple. To remove this construct, one has to sequentialize the assignments involved which usually requires auxiliary variables. This is a spot of frequent error in attempts to write down algorithms straightforwardly. Therefore we prefer this intermediate step which allows us to avoid such errors by using the standard knowledge about the treatment of collective assignments.

3 Singly Linked Lists

We now want to exemplify the use of the above rule in the development of a function for deleting an element from a singly linked list. The data type for lists is defined by

```
typedef struct lis { int  item ;
                     lis *next ;
                   } listel, *list ;
```

Of course, `int` might be replaced by any other type.

Two basic operations on lists are the emptiness test

```
int isempty (list l)
    { return (l == NULL) ; }
```

and adding a new record to the beginning of a list

```
list prefix (int x, list l)
     { list new = (list) malloc(sizeof(listel));
       new->item = x;
       new->next = l;
       return new;
  }
```

Here, we have omitted the test for success of the call to `malloc`.

We now give a simple recursive formulation of a function that deletes the first occurrence, if any, of an element `x` from a list `l`:

```
list delete (int x, list l)
     { if (isempty(l))
          return l ;
       else if (l->item == x)
               return l->next ;
       else { l->next = delete(x,l->next) ;
              return l ;
            }
     }
```

We now head for an iterative version. The basic idea is to pass to a tail-recursive pure procedure and subsequently to apply our scheme for transition to loop form. We thus introduce the following procedure which overwrites a list variable with the result of the delete operation:

```
void pdelete(int x, list *l)
     { *l = delete(x, *l) ; }
```

This procedure is indirectly recursive through the call to `delete`. To obtain a directly recursive version we use the unfold/simplify/fold transformation strategy (see e.g. [5]).

First we *unfold*, i.e., substitute the body of `delete` for its call, replacing the formal parameters `x` and `l` by the actual parameters `x` and `*l` and, at the same time, distribute the assignment to `*l` into the branches of the conditional statements:

```
void pdelete (int x, list *l)
     { if (isempty(*l))
          *l = *l ;
       else if ((*l)->item == x)
          *l = (*l)->next ;
       else { (*l)->next = delete(x,(*l)->next) ;
              *l = *l ;
             }
     }
```

Next, we remove the trivial assignments `*l = *l`. To prepare the use of our transformation scheme we moreover reorganize the conditional, exploiting the properties of the logical connectives `&&` and `!`. This yields

```
void pdelete(int x, list *l)
     { if (!isempty(*l) && ((*l)->item != x))
          (*l)->next = delete(x,(*l)->next) ;
       else if (!isempty(*l))
               *l = (*l)->next ;
      }
```

The assignment with the call to `delete` has almost the shape of the body of `pdelete`; only the level of dereferencing is not right. To prepare *folding*, i.e., replacement of a statement by a suitable call to a procedure, we therefore have to adjust this so that we obtain a statement of the form

```
*m = delete(y,*m)
```

for suitable `m,y`. For `y` we can use `x`. To find `m` we use the fact that in C for arbitrary address-valued expression `exp` the expressions `exp` and `*&exp` are equivalent and obtain

```
void pdelete(int x, list *l)
     { if (!isempty(*l) && ((*l)->item != x))
          *&((*l)->next) = delete(x,*&((*l)->next)) ;
       else if (!isempty(*l))
               *l = (*l)->next ;
      }
```

so that we can take `&((*l)->next)` for `m`. We fold the then-branch into a (recursive) call of `pdelete` with the actual parameter `&((*l)->next))` instead of `l`:

```
void pdelete(int x, list *l)
     { if (!isempty(*l) && ((*l)->item != x))
          pdelete(x, &((*l)->next)) ;
       else if (!isempty(*l))
               *l = (*l)->next ;
      }
```

This recursion even is a tail recursion. Hence we may now apply our rule for passing to a loop and obtain

```
void pdelete(int x, list *l)
     { while (!isempty(*l) && ((*l)->item != x))
             (x, l) = (x, &((*l)->next)) ;
       if (!isempty(*l))
          *l = (*l)->next ;
      }
```

Here the collective assignment can be eliminated without introduction of an auxiliary variable, since the assignment to `x` is trivial anyway and can simply be omitted. This leads to our final version

```
 void pdelete(int x, list *l)
     { while (!isempty(*l) && ((*l)->item != x))
             l = &((*l)->next) ;
       if (!isempty(*l))
          *l = (*l)->next ;
      }
```

It should be noted here that a corresponding version in standard Pascal or Modula does not exist, since these languages do not allow variables of type `var var type` for addresses of variables. There one has to keep the recursive version which simulates such variables by passing parameters of type `var type` (see e.g. [6] for the case of a deletion algorithm on binary search trees).

4 Conclusion

We hope to have demonstrated that there is a systematic way of passing from simple and clear high-level specifications of algorithms on pointer structures to efficient versions which are much harder to understand and, more importantly, to get right. It may be argued that the proceeding here has not been completely formal so that there still is no complete correctness proof of the final algorithm.

However, the approach could fully be formalized using the pointer algebra introduced in [4, 3].

Acknowledgements. I am grateful to T. Ehm, M. Russling, S. Holland and, in particular, G. Wilhelms for valuable comments on draft versions of this paper.

References

[1] F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal program construction by transformations — Computer-aided, Intuition-guided Programming. IEEE Transactions on Software Engineering **15**, 165–180 (1989)

[2] B.W. Kernighan, D.M. Ritchie: Programming in C. Second Edition, ANSI C. Englewood Cliffs, N.J.: Prentice-Hall 1988

[3] B. Möller: Development of graph and pointer algorithms. In B. Möller, H.A. Partsch, S.A. Schuman (eds.): Formal program development. Lecture Notes in Computer Science **755**. Berlin: Springer 1993, 123–160

[4] B. Möller: Towards pointer algebra. Science of Computer Programming **21**, 57–90 (1993)

[5] H.A. Partsch: Specification and transformation of programs — A formal approach to software development. Berlin: Springer 1990

[6] N. Wirth: Algorithms and data structures. Englewood Cliffs, N.J.: Prentice-Hall 1986