A Practical Approach to Structure Reuse of Arrays in Single Assignment Languages

Andreas Kågedal* Saumya Debray†
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721-0077, USA
Email: {andka,debray}@cs.arizona.edu

Technical Report 96-21
December 1996

Abstract

Array updates in single assignment languages generally require some copying of the array, and thus tend to be more expensive than in imperative languages. As a result, programs in single assignment languages sometimes suffer from a performance handicap compared to those in imperative languages. Traditional attempts to address this problem have typically involved either complex compile-time analyses, which tend to be slow and fragile; or new language constructs, which do not always interface with already existing code. In this paper, we propose a new approach to this problem, based on a simple and straightforward program transformation, that we believe addresses the shortcomings of both of these approaches: it is easy to understand, efficiently implemented, does not require new language constructs, and yet is applicable to most commonly encountered programs.

* Supported by the Swedish Research Council for Engineering Sciences (TFR) under grant number 282-95-974.
† Supported in part by the National Science Foundation under grant number CCR-9502826.

1 Introduction

Single assignment languages, such as pure functional and logic programming languages, do not have any notion of updatable variables: the value of a variable or structure, once defined, does not change during execution. Updates to the value of a variable or structure have to be effected, instead, by creating new copies. While this is semantically very clean, and simplifies many aspects of reasoning about and manipulating programs. it can lead to an undesirable degradation of performance. By analyzing the program at compile time it is possible to replace the inefficient copying with an efficient destructive update of the old data structure if it can be assured that the old structure will never be referenced again. We call this *structure reuse*. In a good optimizing compiler structure reuse would make it possible to improve the performance of important classes of programs to a level where it would be comparable with the performance of optimized C.

Because an update to an element of an aggregate structure, such as an array, requires copying (at least part of) the array in a single assignment language, direct implementations of such structures have been widely viewed as being too expensive to be practical in such languages. The usual approach proposed to address this problem is to use some sort of compile-time liveness analysis to discover opportunities where updates can be implemented destructively, and numerous analysis methods along these lines have been proposed in the literature. In practice, however, we are not aware of any system that employs structure reuse based on global analysis. There are two important reasons for this. The first is that the proposed analysis methods have been very general, and as a result have been rather complex and have suffered from performance problems. On the theoretical side, for example, liveness analysis using the Prop domain, or any domain of comparable precision, is EXPTIME-complete [8]; on the practical side, the system described by Mulkers *et al.* requires over 8 seconds to analyze the four-clause naive-reverse program [19]. The second reason is that in practice, global data flow analyses—especially the complex alias analyses required for structure reuse—often turn out to be very “fragile”: small and apparently minor changes to a program text can have profound effects on analysis results and the performance of the resulting code, often in a way that is difficult for the ordinary programmer to anticipate.

At this point, therefore, the choices available seem to be the following:

1. One can stay with structures like lists or trees. The behavior of such structures is simple and predictable, but they incur nontrivial overheads and can adversely affect the asymptotic complexity of a program.[1]

2. One can use arrays and rely on compiler analyses to eliminate the copying overheads. The disadvantage here is that the analyses are complex and potentially expensive. Moreover, the analysis results tend to be fragile, and the performance characteristics of the resulting code can be difficult for programmers to predict.

3. One can resort to special-purpose programming constructs, such as monads [3, 24, 25], or “uniqueness” declarations [22]. This can work well if one is writing fresh code in a new language, but it is not straightforward to integrate this with pre-existing code (the “dusty decks” problem). Unfortunately, it is not always straightforward or practical to rewrite a large volume of existing code.

What we need, in fact, is a mechanism that (*i*) is simple enough to give programmers a simple and robust performance model for their programs, so that they can determine, without undue effort, how expensive the use of arrays in a program is likely to be; (*ii*) is effective enough to eliminate (most of) the copying overhead for array updates for most commonly encountered programs; and (*iii*) does not rely on any special language features. This paper presents a simple approach to structure reuse that, we believe, meets these criteria. The approach is presented in a logic programming context, but works for any single assignment language, either logic or functional, as long as call

[1] Recent work by Winsborough on compile-time reuse of list cells [27] reduces some of the costs of using lists, but does not alter the asymptotic time costs they incur.

sequences can be precisely predicted (i.e., for functional programs with higher order constructs or `call/cc` constructs, additional work is necessary, just as it is for logic programs with `call/1` or `assert/retract` primitives: this should not come as a surprise, since this additional work would typically be necessary for most analyses and transformations of such programs in any case). The word "predicate" is thus used in the same sense as "function" in a functional language, or "procedure" in other languages.

2 Preliminaries

For our purposes, an array of size N is simply an abstract data type that can be viewed as an aggregate indexed by the set $\{1, \ldots, N\}$. In other words, we make no assumptions about the low-level realization of the array, e.g., as a vector or a tree. Arrays may be nested, and we will usually not distinguish between nested and multidimensional arrays. In the rest of the paper we will assume that the language is equipped with the following primitive for (nondestructively) updating a (possibly multi-dimensional) array:

`update(`*`Array`*`,`*`Cell`*`,`*`Val`*`,`*`NewArray`*`)` where *NewArray* is a new array with the same contents as the array *Array* except that the value in cell *Cell* is replaced with *Val*. *Cell* is a list `[`*`X`*`,`*`Y`*`,`*`Z`*`...]` of coordinates.

We will also assume that the following two operations exist, although they might preferably be hidden from the programmer:

`copy(`*`Array`*`,`*`ArrayCopy`*`)` Allocates a new array *ArrayCopy* with the same size as *Array*, and copies the contents of the cells of *Array* to the cells of *ArrayCopy*.[2] From a declarative point of view, *Array* and *ArrayCopy* will be the same array.

`destr_update(`*`Array`*`,`*`Cell`*`,`*`Val`*`)` The value in cell *Cell* of *Array* is destructively replaced with *Val*. *Cell* is a list `[`*`X`*`,`*`Y`*`,`*`Z`*`...]` of coordinates.

We will use the syntax `A[`*`X`*`,`*`Y`*`,`*`Z`*`...]` for the value of the cell specified by coordinates *X*, *Y*, *Z*... in the array `A`.

We assume that programs are *moded*, i.e., it is known which arguments of a procedure are input arguments and which are outputs. The input arguments in a call are assumed to be ground at the time the call is made. There is an extensive body of literature on compile-time inference of modes, and we do not pursue this further here except to note that mode (and, in general, groundness) information is of independent interest for a variety of other optimizations. To simplify the presentation, in the discussion that follows, the first and last argument positions of each predicate are assumed to have mode input and output respectively. Obviously, in real programs the array can be passed down and up from the iteration predicate in any argument positions of the iterative predicate as long as it is done consistently.

3 The Basic Idea

The basic intuition behind our approach is very simple. The only reason an arbitrary array update operation cannot be implemented destructively is that in general, the "old" version may be referenced subsequently, i.e., may have pointers to it. Now in most programs that use arrays, updates occur in loops, and typically, more than one element is updated: for example, as part of a Gaussian elimination routine, we may multiply each element of a row of a matrix by some number. So, if we first create a copy of the array and pass this copy into the loop that effects the updates, the copy will not have any pointers to it (the pointers to the "old" array will continue to point to that version,

[2] Observe that unlike the `copy_term/2` predicate of Prolog, the structures making up the contents of the cells are not copied and no new free variables are created.

not to the copy that has been made) and will therefore be amenable to destructive updates. The resulting program will be no worse in performance than the original program without destructive updates if at least one update takes place, and will be definitely better than the original program if more than one update takes place (which is what we expect).

There are, of course, various subtleties that have to be addressed in order to realize this idea in practice, and these are discussed in the remainder of this paper. First, however, we illustrate the approach with a simple example. The program in Fig. 1(a) iterates over an array of integers incrementing each cell by one.

```
q(A, NewA) :-
  inc_elems(A, 1, 20, NewA).


inc_elems(A , I, U, A) :-
  I > U.
inc_elems(A , I, U, NewA) :-
  I =< U,
  update(A, [I], A[I]+1, TempA),
  inc_elems(TempA, I+1, U, NewA).
```

(a) A simple iteration over an array.

```
q(A, NewA) :-
    copy(A, ACopy),
    inc_elems(ACopy, 1, 20, NewA).

inc_elems(A , I, U, A) :-
    I > U.
inc_elems(A , I, U, NewA) :-
    I =< U,
    destr_update(A, [I], A[I]+1),
    inc_elems(A, I+1, U, NewA).
```

(b) The program of Fig. 1(a) with array copying moved out of the loop.

Figure 1: A Simple example.

Instead of making a potentially complex check that the original array is indeed not going to be used after the call we *enforce* this criterion. Updating a cell of the array can conceptually be divided in two operations: first copying the array, and then making a destructive change in the cell of the copy. The copy operation can then be moved, or "pushed", out of the loop to the point just *before* the predicate implementing the iteration is called. Copying the array before the iteration begins is, in effect, a way to enforce the possibility of a destructive update inside the loop.

The resulting program can be found in Fig. 1(b) in effect, copying of the array in each iteration has been replaced by a single copy operation before the iteration starts, followed by a series of destructive updates, and resulting in a program with linear time complexity.

Before we go on to present this optimization in more detail it might be in order to make some remarks on what it does and does not try to accomplish. It is a *local* and simple optimization that does not require any global analysis. It should therefore be cheap to include in a compiler. The ambition of the proposed optimization is not to remove all array copying from the program. Instead it is targeted at iteration like structures where array copying during updating is likely to be both costly and unnecessary. Also, it should be worth noting that the optimization does not solve the general problem of structure reuse for recursive data structures such as lists and trees [27].

3.1 The Transformation and its Correctness

In this section, we describe the conceptual steps involved in the transformation that forms the heart of our approach. This serves to illustrate the thinking behind the transformation, and makes it easier to adapt the transformation to situations not explicitly discussed here. It also makes clear the preconditions necessary for each step of the transformation to be carried out, thereby developing the conditions under which the transformation is correct.

The basic pattern of programs similar to the one in Fig. 1(a) is given in Fig. 2(a), where everything

```
q(A, ..., NewA) :-
   p(A, ..., NewA).

p(A, $\overline{X}_{bc}$, A) :- $B_{bc}$.
p(A$_0$, $\overline{X}_h$, A$_2$) :- $B_0$,
   update(A$_0$, $\overline{X}_u$, A$_1$), $B_1$,
   p(A$_1$, $\overline{X}_r$, A$_2$), $B_2$.
```

(a) Original program.

```
q(A, ..., NewA) :-
   p(A, ..., NewA).

p(A, $\overline{X}_{bc}$, A) :- $B_{bc}$.
p(A$_0$, $\overline{X}_h$, A$_2$) :- $B_0$,
   copy(A$_0$, A$_1$),
   destr_update(A$_1$, $\overline{X}_u$), $B_1$,
   p(A$_1$, $\overline{X}_r$, A$_2$), $B_2$.
```

(b) First the update operation is split in two operations,....

```
q(A, ..., NewA) :-
   p(A, ..., NewA).

p(A, $\overline{X}_{bc}$, A) :- $B_{bc}$.
p(A$_0$, $\overline{X}_h$, A$_2$) :-
   copy(A$_0$, A$_1$), $B_0$,
   destr_update(A$_1$, $\overline{X}_u$), $B_1$,
   p(A$_1$, $\overline{X}_r$, A$_2$), $B_2$.
```

(c) then the copy is moved first in the clause,....

```
q(A, ..., NewA) :-
     copy(A, ACopy),
     p(ACopy, ..., NewA).

p(A, $\overline{X}_{bc}$, A) :- $B_{bc}$.
p(A$_1$, $\overline{X}_h$, A$_2$) :- $B'_0$,
   destr_update(A$_1$, $\overline{X}_u$), $B_1$,
   copy(A$_1$, A$_1^c$),
   p(A$_1^c$, $\overline{X}_r$, A$_2$), $B_2$.
```

(d) then the copy operation is moved to before calls to p/2,...

```
q(A, ..., NewA) :-
   copy(A, ACopy),
   p(ACopy, ..., NewA).

p(A, $\overline{X}_{bc}$, A) :- $B_{bc}$.
p(A$_1$, $\overline{X}_h$, A$_2$) :- $B'_0$,
   destr_update(A$_1$, $\overline{X}_u$), $B_1$,
   p(A$_1$, $\overline{X}_r$, A$_2$), $B_2$.
```

(e) then the copy operation in the recursive clause is removed.

```
q(A, ..., NewA) :-
   copy(A, ACopy),
   p(ACopy, ...),
   NewA = ACopy.

p(A, $\overline{X}_{bc}$) :- $B_{bc}$.
p(A$_1$, $\overline{X}_h$) :- $B'_0$,
   destr_update(A$_1$, $\overline{X}_u$), $B_1$,
   p(A$_1$, $\overline{X}_r$), $B'_2$.
```

(f) Finally, the returning of the array in a separate argument is omitted.

Figure 2: The transformation illustrated on a simplified program.

in the bodies of the clauses not involving the array have been replaced with meta variables B_{bc}, B_0, B_1 and B_2, representing any goals, and the heads, the update operations and the recursive calls have been abbreviated such that all arguments except the array arguments have been replaced with meta variables $\overline{X_h}$, $\overline{X_{bc}}$, $\overline{X_u}$, $\overline{X_r}$. We assume that $\mathtt{A}_0$, $\mathtt{A}_1$, $\mathtt{A}_1$, $\mathtt{A}$ and $\mathtt{NewA}$ are all distinct variables. Also, to make the argument simpler, will assume that any occurrence in $\overline{X_u}$ of a reference to a cell in $\mathtt{A}_0$ has been replaced with a new variable $\mathtt{N}$ and a unification statement $\mathtt{N = A_0[...]}$ in B_0 before the transformation begins. This simplified form of programs will be used in the rest of the paper.

Fig. 2 will be used in an argument to determine under what criteria the transformation is correct. As mentioned earlier, the update operation can be divided into two operations: a copy of the array, and a destructive update of the copy. However, this is permissible only if we can guarantee that the resulting array, $\mathtt{A}_1$, is the one that is created by the copy operation, i.e., that $\mathtt{A}_1$ is not an already-defined array that is unified with the result of the update. The resulting program is shown in Fig. 2(b).

The following criterion is sufficient to ensure correctness:

Criterion 1 $\mathtt{A}_1$ does not occur in B_0 or $\overline{X_h}$.

We next move the copy operation first in the clause which yields the program in Fig. 2(c). This is OK due to Crit. 1.

Next, we observe that $\mathtt{A}_0$ is an input parameter of $\mathtt{p}$, which implies that $\mathtt{A}_0$ is ground when $\mathtt{p}$ is called. Since $\mathtt{A}_1$ is a copy of $\mathtt{A}_0$, the *value* of $\mathtt{A}_0$ is identical to that of $\mathtt{A}_1$: in other words, the result of the computation will not be affected if we move the copy operation outside $\mathtt{p}$ and pass $\mathtt{A}_1$ into $\mathtt{p}$ instead of $\mathtt{A}_0$. Once this has been done, since the input argument $\mathtt{A}_1$ is a copy of $\mathtt{A}_0$, we know there are no aliases for it: any existing pointers to $\mathtt{A}_0$ must continue to point to $\mathtt{A}_0$, not to $\mathtt{A}_1$. At this point, therefore, if $\mathtt{A}_0$ is not referenced in the body of $\mathtt{p}$ after the update, i.e. does not occur in B_1, B_2, $\overline{X_r}$ or $\overline{X_u}$, then $\mathtt{A}_1$ is dead after the update and can therefore be updated destructively (the array $\mathtt{A}_0$ may be accessed in B_0, but since $\mathtt{A}_1$ is a copy of $\mathtt{A}_0$ and, from Criterion 1 that $\mathtt{A}_1$ does not already occur in B_0, these accesses to $\mathtt{A}_0$ can be replaced by accesses to $\mathtt{A}_1$; in theory we can also replace occurrences of $\mathtt{A}_0$ in $\overline{X_u}$, but we will instead assume that it does not occur there. This results in the program of Fig. 2(d), where B_0' denotes the result of replacing occurrences of $\mathtt{A}_0$ by $\mathtt{A}_1$ in B_0. Due to the destructive update, $\mathtt{A}_1$ holds the value of the original array in B_0 and the value of the updated array in the rest of the clause. The reason there is a copy operation in the body, just before the recursive call to $\mathtt{p}$, is that a copy of the argument is being passed into the call, just as in the case for the call to $\mathtt{p}$ from $\mathtt{q}$. The correctness criterion can now be refined to:

Criterion 2 $\mathtt{A}_0$ does not occur in B_1, B_2, $\overline{X_u}$, $\overline{X_r}$ or $\overline{X_h}$

We have now assured that the array $\mathtt{A}_1$ passed in the first argument position to $\mathtt{p}$ has no other references when it enters the clause. Given that we know that no new aliases for $\mathtt{A}_1$ are produced in B_0' (i.e., no new aliases for $\mathtt{A}_0$ are produced in B_0) or B_1 and $\mathtt{A}_1$ does not occur in $\overline{X_u}$, [3] $\mathtt{A}_1$ will hold the only live reference to the array in the clause when $\mathtt{A}_1$ is subsequently copied into $\mathtt{A}_1^c$. Since the sole reason for copying is to ensure that we have a unique reference to the array, it is obvious that this copy operation is unnecessary provided that the value of $\mathtt{A}_1$ is not needed $\overline{X_r}$ or in B_2 i.e., $\mathtt{A}_1$ does not occur in $\overline{X_r}$ or in B_2. Removing this copy gives us the program in Fig. 2(e) and the new correctness criterion:

Criterion 3 No new aliases for $\mathtt{A}_0$ are produced in B_0, and
No new aliases for $\mathtt{A}_1$ are produced in B_1 or $\overline{X_u}$, and
$\mathtt{A}_1$ does not occur in B_2 or $\overline{X_r}$.

[3] Actually, at this point it is only necessary that no new aliases $\mathtt{A}_0$ for are produced in $\overline{X_u}$. But, since this can only happen when the update is a predicate call, which can be the case in Sec. 4.2, occurrence in $\overline{X_u}$ is in fact the same thing.

Finally, we see that as long as the array is being updated it is passed "downwards" in the recursion of `p` and it is not until the iteration stops in the first clause of `p` that the array is unified with the last argument of `p` which passes the final version back up again. Since we now have removed all copying operations from the looping predicate we know that this final array will in fact be present as the value of the variable `ACopy` in the clause of `q` containing the initial call to the iteration predicate `p`. It is therefore no longer necessary to have an extra argument of the looping predicate passing the array back up. Instead, the final value of the array can be obtained in the clause of the initial call to the iteration by unifying with `ACopy` *after* the iteration. We know that $\mathtt{A}_1$ will hold the same value as $\mathtt{A}_2$ in B_2 so if we remove $\mathtt{A}_2$ as an argument to `p` we can just replace any occurrence of $\mathtt{A}_2$ in B_2 with $\mathtt{A}_1$. In B_0', B_1, $\overline{X_u}$ and $\overline{X_r}$ we don't have another variable that contains the value of $\mathtt{A}_2$ so this only works if $\mathtt{A}_2$ does not occur there. As $\mathtt{A}_2$ occurs in B_0' iff it occurs in B_0 we get the criterion

Criterion 4 $\mathtt{A}_2$ does not occur in B_0, B_1, $\overline{X_u}$ or $\overline{X_r}$.

This leaves us with the program in Fig. 2(f), where $B_2' = B_2(\mathtt{A}_2/\mathtt{A}_1)$.

The final step of removing the returning of the resulting array in a separate argument has the obvious gain of avoiding the extra time and space it would require to handle. There is also another important advantage which we will exploit in Sec. 4.2: It turns the iterative predicate `p` into a predicate that in effect makes a destructive update of the array and can as such replace `destr_update` in the transformation scheme. For this to be OK we need to know that no new aliases for the array has been produced during the update. We thus need to add the following criteria:

Criterion 5 No new aliases for $\mathtt{A}_2$ are produced in B_2, and
$\mathtt{A}_2$ does not occur in $\overline{X_h}$, and
No new aliases for `A` are produced in B_{bc}, and
`A` does not occur in $\overline{X_{bc}}$.

To sum up, the criteria 1–5 for preserving correctness when transforming the program in Fig. 2(a) to the program in Fig. 2(f) can be stated as:

Criterion 6 No new aliases for $\mathtt{A}_i$ are produced in B_i, and
$\mathtt{A}_i$ does not occur in B_j, when $i \neq j$, and
$\mathtt{A}_i$ does not occur in $\overline{X_h}$, $\overline{X_u}$ or $\overline{X_r}$, and
No new aliases for `A` are produced in B_{bc}, and
`A` does not occur in $\overline{X_{bc}}$.

This criterion expresses an intuition that is very similar to "single threadedness" (see, for example, [23]).

3.2 Applicability

To find opportunities for applying the transformation the clauses in of recursive predicates—i.e., in the strong components of the call graph—are searched for array updates. For each such predicate the analysis required to figure out whether it adheres to the criteria is then essentially local to its clauses.

Part of Crit. 6 (Sec. 3.1) requires that $\mathtt{A}_i$ is not aliased in B_i, i.e. $\mathtt{A}_i$ does not in B_i pass its value on to some other variable or into a data structure. Making a general check for this is difficult and must be approximated. Checking that $\mathtt{A}_i$ does not occur at all in B_i is much simpler and obviously sufficient. This criterion is unnecessarily restrictive and is easily extended by allowing $\mathtt{A}_i$ to occur in primitive operations such as extraction of cell contents etc., which does not involve any risk for aliasing.

There are, however, some programs that would be amenable to the transformation only if the array of $\mathtt{A}_i$ would be allowed to be passed down as an argument in a predicate call in B_i. For example, a straightforward implementation of Dijkstra's shortest paths algorithm (the `dijkstra` program in

Sec. 5) contains a nested loop in which one array is updated in the inner loop and another in the outer loop. The array updated in the outer loop is used in the inner loop and therefore must be passed as an argument in the call to the inner loop. To allow this requires a non-local analysis which can ensure that the array is not aliased in the clauses of the predicate where it is passed. This is, in general, expensive to check, but we believe the following scheme covers most cases occurring in practice and is also easy to understand and check by a programmer:

Trace the array "downwards" and make sure it is never put into another data structure, and never passed back up from any predicate it enters. This can be checked in a single pass over the relevant portion of the program and does not need to involve other program variables than those holding the array, so it should still be cheap to check in most cases, even though it is not, strictly speaking, a local analysis. To keep the check cheap in *all* cases, one can always give up, and assume that it might be aliased, in case the check takes too much time.

It should be noted that the check described above, as well as the criteria discussed in the previous section, are all straightforward to check: most involve only simple intra-clause syntactic checks, with the most complicated requirement—the only non-local one—requiring a single pass over part of the program. Because of this, we expect that it should not be difficult for a programmer to form a reasonably robust mental performance model indicating the expected performance of programs using arrays.

4 Extending the Scheme

For this transformation to be useful, it needs to be extended in several ways. Here we will describe some of the possible extensions. Some will just be mentioned and some will be described in more detail.

4.1 More Clauses

The extensions needed to cater for more clauses in the iteration predicate, including base case clauses containing updates and recursive clauses not containing updates, are trivial and will not be discussed in any length here. It suffices to note that in a logic programming setting backtracking might cause a problem since destructive updates might have to be trailed. Even if value trailing is necessary, however, it is likely to be considerably less expensive than array copying. It should also be noted that if some of the clauses do not contain update operations we get a "speculative" copying behavior. Solutions to this are discussed in Sec. 4.4.

It should also be evident from the correctness discussion in Sec. 3.1 that it is trivial to extend the scheme to allow several updates and recursive calls in the same clause, as long as the array is "threaded" between them.

4.2 Nested Iterations

Many common algorithms use nested iterations i.e., a loop within another loop. In a declarative ? language this is achieved using two recursive predicates as illustrated in Fig. 3(a). The inner loop of this program can trivially be transformed using the basic scheme of Fig. 2 yielding the program in Fig. 3(b). It is now not difficult to see that the inner loop predicate `p_inner/1` of Fig. 3(b) essentially implements a destructive update of the array, and thus the two lines marked `(*)` in the program in Fig. 3(b) implements a non-destructive update of the array $\mathtt{A}_0$ producing the new array `ACopy`.

Assuming that the conditions for transformation are fulfilled for `p_outer` after removing the explicit unification $\mathtt{A}_1$`=ACopy` (by replacing all occurrences of `ACopy` with $\mathtt{A}_1$ which is OK if $\mathtt{A}_1$ does not occur in G_0, which it won't if criterion 1 is fulfilled) in Fig. 3(c) the transformation can now be applied pushing the copy operation out another level yielding the final copy free iteration in Fig. 3(d).

```
q(A, NewA) :-
   p_outer(A, NewA).

p_outer(A, A).
p_outer(A0, A2) :- G0,
   p_inner(A0, A1), G1,
   p_outer(A1, A2), G2.

p_inner(A, A).
p_inner(A0, A2) :- B0,
   update(A0, A1), B1,
   p_inner(A1, A2), B2.
```

(a) A two level iteration.

```
q(A, NewA) :-
   p_outer(A, NewA).

p_outer(A, A).
p_outer(A0, A2) :- G0,
   copy(A0, ACopy), % (*)
   p_inner(ACopy), % (*)
   A1 = ACopy, G1,
   p_outer(A1, A2), G2.

p_inner(A).
p_inner(A1) :- B0',
   destr_update(A1), B1,
   p_inner(A1), B2.
```

(b) Inner loop transformed...

```
q(A, NewA) :-
    p_outer(A, NewA).

p_outer(A, A).
p_outer(A0, A2) :- G0,
   copy(A0, A1),
   p_inner(A1), G1,
   p_outer(A1, A2), G2.

p_inner(A).
p_inner(A1) :- B0',
   destr_update(A1), B1,
   p_inner(A1), B2.
```

(c) ...and, after removing the explicit unification ...

```
q(A, NewA) :-
   copy(A, ACopy),
   p_outer(ACopy),
   NewA = ACopy.

p_outer(A).
p_outer(A1) :- G0',
   p_inner(A1), G1,
   p_outer(A1), G2'.

p_inner(A).
p_inner(A1) :- B0',
   destr_update(A1), B1,
   p_inner(A1), B2'.
```

(d) ...the outer loop can be transformed.

Figure 3: Two level iteration.

How the remaining copy operation, just before the iteration takes off, can be avoided is discussed in Sec. 4.4.

4.3 Other Extensions

There are several other ways in which the basic transformation scheme can be extended. For instance, if the array should be updated only in some iterations one can implement this in two ways. (1) By having several recursive clauses where some update the array, and some don't. (2) By having just one recursive clause that calls another non-recursive predicate that implements the "if-statement" that determines whether the array should be updated or not. In a bubble sort program this could be a `swap_maybe` predicate that determines if the contents of two cells should be swapped.

Only case (1) is covered by the basic transformation, but case (2) is easily included by regarding the updating predicate called from the looping predicate as an inner loop without any recursive clauses.

Also, the transformation as formulated does not handle mutual recursion. It seems, however, that this should not be difficult to include.

4.4 Avoiding Speculative Copying

If some of the clauses of the iteration predicate do not contain updates of the array the transformation is "speculative" in the sense that the number of array copy operations avoided in the iteration might be zero, but they are always replaced with one copy operation before the iteration starts. Thus, it is possible that the transformed program might actually end up doing more copying than the untransformed.

There are two approaches to this problem. One can try to refine the transformation in various ways to avoid this problem. This is done in the remainder of this section. Or, one can say that this does not really matter that much. The possibility of avoiding large amounts of copying is worth the price of making one copy, even if that sometimes means it is done unnecessarily.

If it could be determined that, at the point just before the call to the iteration predicate, there is only one reference to the array (note that references to individual elements of an array count as references to the array), then also this initial copy could be omitted. One possibility is to fall back on a global data flow analysis to infer this information. Alternatively, if we want to avoid a global data flow analysis we can still handle cases such as when the array was created just before the call, or was returned from another, just transformed, iteration. Nevertheless, even if we decide to forego a global analysis and as a result are unable to remove this one copy operation, we have succeeded in reducing the total number of copy operations considerably: thus, in a program where each element of an n-element vector is updated, our transformation results in a single copy operation on the vector, followed by n destructive updates, with an overall amortized time complexity of $O(1)$ per update.

However, if we cannot establish that the initial copying can be trivially omitted, we can delay the initial copying until it is known that at least one update of the array will take place. This can be achieved by dividing the loop in two parts, one that takes care of the iteration *before* the first update, and the other one *after*. This is illustrated in Fig. 4.

This technique does, however, only work for one level iterations. If the initial copy of a program such as the one in Fig. 3 is to be avoided we need some additional machinery. The inner loop has to "tell" the outer loop that a copy has indeed occurred. A conceptually simple way to accomplished this is to add an argument to the inner loop predicate that passes back a flag that says whether the array was copied or not. The outer loop predicate can then, based on this flag, decide if the iteration should continue checking whether a copy will occur, or if it is safe to just do destructive updates. This is illustrated in Fig. 5.

If several arrays are updated in different ways and by different clauses in the same loop, and we have not been able to remove the initial copying for any of them, then we need lots of versions of the looping predicates. If there are n arrays, the transformation will have to produce 2^n versions of the looping predicates: note, however, that determining which of these 2^n versions to use can be

```
p(A, A).
p(A, NewA) :-
    update(A, TempA),
    p(TempA, NewA)
p(A, NewA) :-
    p(A, NewA).
```

(a) A program...

```
p(A, A).
p(A, NewA) :-
    update(A, TempA),
    p'(TempA),
    NewA = TempA.
p(A, NewA) :-
    p(A, NewA).

p'(A).
p'(A) :-
    destr_update(A),
    p'(A).
p'(A) :-
    p'(A).
```

(b) ...transformed

Figure 4: A program with two recursive clauses where one updates the array and one does not.

effected with n tests using a decision tree. It may be possible to reduce the number of specialized predicates using techniques based on automata minimization techniques [20, 26].

However, on a lower level than can be expressed directly in a declarative language the same effect can be achieved more efficiently. This idea is illustrated in Fig. 6 adding some low level instructions into the program. We pass out from the inner loop not a flag saying whether a copy has occurred or not, but instead the address of where to continue after the initial call to `p_inner`. This requires that `p'_outer` behaves exactly like `p_outer` as far as stack frames, variables and backtracking goes. The only difference should be that it calls `p'_inner` instead of `p_inner`.

It is also easy to see that if `p_inner` is tail recursive, it is not necessary to pass the continuation out through an argument and do an explicit *goto* when the inner loop is done. Instead the continuation register can be updated immediately, and only at the point in `p_inner` where the update is made since it will initially hold the address of *lbl1*.

If we have several calls to `p_inner` from `p_outer` this does not work. It could be fixed by passing the alternative address as an argument to `p_inner`. However, if there are several arrays being updated in the inner loop this still does not solve the problem. In this case the previously described technique with returned "flags" must be used.

5 Evaluation

Program	Size	Before	After	Factor
dijkstra	300	3587.6	398.3	9.0
shortestp	20×20	377.7	33.3	11.3
bubblesort	200	593.9	22.1	26.9
insertsort	200	593.2	20.6	28.8
quicksort	500	10470.0	180.0	58.2

Table 1: Speedups after transformation (times in ms).

Neither the analysis or the transformation has been implemented. To investigate the viability of the approach a few programs were implemented and compiled with the `jc` compiler [11] (with the optimize switch '`-O`') which supports arrays. The intermediate C code for these programs were then transformed by hand and recompiled. The resulting improvements in execution time is shown in Tab. 1. The system used for the timings was a Sun SPARCstation 20 612MP with two 60MHz SuperSPARC processors (only one was used) and 128MB memory, running Solaris 2.5.1. Timings include runtime garbage collection.

```
p_outer(A, A).
p_outer(A, NewA) :-
    p_inner(A, TempA),
    p_outer(TempA, NewA).

p_inner(A, A).
p_inner(A, NewA) :-
    update(A, TempA),
    p_inner(TempA, NewA).
p_inner(A, NewA) :-
    p_inner(A, NewA).
```

(a)

```
p_outer(A, A).
p_outer(A, NewA) :-
    p_inner(A, TempA, C),
    ( C = not_copied
           -> p_outer(TempA, NewA)
    ; C = copied
           -> p'_outer(TempA),
              NewA = TempA ).

p_inner(A, A, not_copied).
p_inner(A, NewA, copied) :-
    update(A, TempA),
    p'_inner(TempA),
    NewA = TempA.
p_inner(A, NewA, C) :-
    p_inner(A, NewA, C).
```

```
p'_outer(A).
p'_outer(A) :-
     p'_inner(A),
     p'_outer(A).

p'_inner(A).
p'_inner(A) :-
    destr_update(A),
    p'_inner(A).
p'_inner(A) :-
    p'_inner(A).
```

(b)

Figure 5: The inner loop "tells" the outer loop whether an update has taken place or not.

```
p_outer(A, A).
p_outer(A, NewA) :-
    p_inner(A, TempA, Cont),
    goto(Cont),
  lbl1:
    p_outer(TempA, NewA).

p_inner(A, A, Cont) :-
    Cont = lbl1.
p_inner(A, NewA, Cont) :-
    Cont = lbl2,
    update(A, TempA),
    p'_inner(TempA),
    NewA = TempA.
p_inner(A, NewA, Cont) :-
    p_inner(A, NewA, Cont).
```

```
p'_outer(A,_).
p'_outer(A,_) :-
    p'_inner(A),
  lbl2:
    p'_outer(A,_).

p'_inner(A).
p'_inner(A) :-
    destr_update(A),
    p'_inner(A).
p'_inner(A) :-
    p'_inner(A).
```

Figure 6: The inner loop returns a continuation address which depends on whether an update has taken place.

The speedup factor is, of course, not constant but a function of the problem size since the complexity is reduced. This is the reason for the big factor for the quicksort program compared to the other sorting algorithms (`quicksort` with an array of size 200 gets a speedup factor of only 14.4).

The program `dijkstra` implements Dijkstra's algorithm for finding the shortest path from one node to all other nodes in a directed graph [1]. It is a two level nested loop where one array is updated in the inner loop and one in the outer. The array updated in the outer loop is, however, used in the inner loop, which means that the transformation w.r.t. this array relies on the check described in Sec. 3.2.

The program `shortestp` finds the shortest path from every node to every node of a graph with n nodes using the algorithm as described by, for instance, Baase [2]. It is a three level nested loop iterating over an $n \times n$ array. The transformation is straight forward and only needs the basic scheme of Sec. 3.1. Since `jc` only supports one dimensional arrays, the $n \times n$ was implemented as a one dimensional array of size n^2.

`bubblesort`, `insertsort` and `quicksort` implement the standard sorting algorithms. `bubblesort` is a two level nested loop where the inner loop (sometimes) swaps the contents of two cells. `bubblesort` does not update an already sorted array and would therefore need the techniques of Sec. 4.4 to eliminate the initial copy.

The program `insertsort` is a three level nested loop where the middle loop searches for the location where an element should be inserted and the inner loop shifts all elements after this location one step.

The program `quicksort` is interesting in that it is an inherently recursive algorithm that does not really have a purely "iterative" form, since it contains two recursive calls in the same clause. As noted in Sec. 4.1, the transformation scheme is easily extended to handle this.

6 Previous Work

A number of authors have considered the optimization of programs in single-assignment languages to incorporate destructive updates, e.g., see [4, 14, 13, 15, 16, 17] in the context of functional programming languages, and [6, 5, 10, 18, 19, 23] in the context of logic programming languages. The work of Bruynooghe [6, 5], Foster and Winsborough [10], Hudak and Bloss [4, 15, 16], Mulkers *et al.* [19], and Sastry *et al.* [23] focus on compile-time reference counting schemes to determine when a data structure being updated has at most one reference to it, and can therefore be safely updated in place. The work of Draghicescu and Purushothaman [9], Gopalakrishnan and Srivas [12], and Sastry and Clinger [21], is aimed at determining an evaluation order for expressions in a functional program so that uses of a structure can be evaluated before updates to the structure, allowing updates to be carried out in place wherever possible. All of these involve compiler analyses of different degrees of complexity and precision, with the drawbacks discussed in Section 1. The related problem of *how* best to reuse structures, given that we know *which* structures to reuse, is considered by Debray [7] and Winsborough [27].

A very different approach to the aggregate update problem involves the development of language constructs aimed specifically at supporting a style of programming that allows the compiler to determine, without excessive effort, updates that can be implemented destructively. The work on monads [3, 24, 25] falls into this category, as does the "unique" declarations of Mercury [22]. As mentioned in Section 1, this can work well if one is writing fresh code in a new language, but it is not straightforward to integrate this with pre-existing code (the "dusty decks" problem).

7 Conclusions

Compile-time analyses aimed at implementing array updates in single-assignment languages via destructive assignment have been the subject of a great deal of research in the last decade. Most approaches that have been proposed either involve complex and potentially fragile compiler analyses, or require special language constructs that may not be available in pre-existing code. In this paper, we propose another approach that is able, we believe, to avoid the drawbacks of either of these approaches: it is conceptually very simple to understand and straightforward to implement, and does not require any special language support. Preliminary experimental results indicate that it leads to promising performance improvements.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1947.

[2] Sara Baase. *Computer Algorithms, Introduction to Design and Analysis.* Addison-Wesley, second edition, 1988.

[3] Y. Bekkers and P. Tarau, "Monadic Constructs for Logic Programming", *Proc. International Symposium on Logic Programming*, 1995. The MIT Press.

[4] A. Bloss. *Path Analysis and Optimization of Non-strict FunctionalLanguages.* PhD thesis, Dept. of Computer Science, Yale University, 1989.

[5] M. Bruynooghe, A. Mulkers, and K. Musumbu. Compile-time garbage collection for prolog. Technical report, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, 1988.

[6] Maurice Bruynooghe. Compile time garbage collection or how to transform programs in an assignment-free language into code with assignments. In *IFIP TC 2 Working Conference on Program Specification and Transformation*, Bad Tölz, F.R. Germany, 1986.

[7] S. K. Debray, "On Copy Avoidance in Single Assignment Languages", *Proc. Tenth International Conference on Logic Programming*, Budapest, Hungary, June 1993, pp. 393–407.

[8] S. K. Debray, "On the Complexity of Dataflow Analysis of Logic Programs", *ACM Transactions on Programming Languages and Systems* vol. 17 no. 2, March 1995, pp. 331–365.

[9] M. Draghicescu and S. Purushothaman. An uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science.* To appear.

[10] Ian Foster and Will Winsborough. Copy avoidance through compile-time analysis and local reuse. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, San Diego, USA, 1991. The MIT Press.

[11] D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint Int. Conf. and Symp. on Logic Programming*, Nov. 1992, pp. 399–413. The MIT Press.

[12] G. C. Gopalakrishnan and M. K. Srivas. Implementing functional programs using mutable abstract data types'. *Information Processing Letters*, 26(6), 1988.

[13] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Stanford University, 1989. (Also available as Technical Report CSL-TR-89-384.).

[14] K. Gopinath and John L. Hennessy. Copy elimination in functional languages. In *16th ACM POPL Symposium*, pages 303–314, Austin, Texas, 1989. ACM Press.

[15] P. Hudak. A semantic model for reference counting and its abstraction. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretations of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[16] P. Hudak and A. Bloss. The aggregate update problem in functional programming languges. In *Proc. 12th ACM POPL Symposium*, pages 300–314. ACM, 1985.

[17] S. B. Jones and D. Le Metayer. Compile-time garbage collection by sharing analysis. In *Proc. Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1989.

[18] F. Kluźniak. Compile-time garbage collection for ground Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1490–1505, Seattle, 1988. The MIT Press, Cambridge.

[19] Anne Mulkers, William Winsborough and Maurice Bruynooghe. Live-Structure Dataflow Analysis for Prolog. *ACM Transactions on Programming Languages and Systems* vol. 16 no. 2, March 1994, pp. 205–258.

[20] Germán Puebla and Manuel Hermenegildo. Implementation of multiple specialization in logic programs. In *ACM SIGPLAN PEPM'95*, pages 77–87, La Jolla, Ca, USA, 1995. ACM Press.

[21] A. V. S. Sastry and W. Clinger. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. Technical Report CIS-TR-92-14, Dept. of Computer and Information Science, University of Oregon, July 1992.

[22] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, October-December 1996.

[23] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable threadedness analysis for concurrent logic programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 493–508, Washington, USA, 1992. The MIT Press.

[24] P. Wadler, "The essence of functional programming", *Proc. 1992 ACM Symposium on Principles of Programming Languages*, Jan. 1992, pp. 1–14.

[25] P. Wadler, "How to Declare an Imperative", *Proc. International Logic Programming Symposium*, Dec. 1995. The MIT Press.

[26] Will Winsborough. Multiple specializaion using minimal-function graph semantics. *The Journal of Logic Programming*, 13(1&2):259–290, July 1992.

[27] W. Winsborough, "Update In Place: Overview of the Siva Project", *Proc. 1993 International Symposium on Logic Programming*, pp. 94–113. The MIT Press.