

Handling Pointers and Dynamic Memory

Laurent Hascoët and Jean Utke

Abstract Proper handling of pointers and the (de)allocation of dynamic memory in the context of an adjoint computation via source transformation has so far had no established solution that is both comprehensive and efficient. This paper gives a categorization of the memory references involving pointers to heap and stack memory along with principal options to recover addresses in the reverse sweep. The main contributions are a code analysis algorithm to determine which remedy applies, memory mapping algorithms for the general case where one cannot assume invariant absolute addresses and an algorithm for the handling of pointers upon restoring checkpoints that reuses the memory mapping approach for the reverse sweep.

Key words: reverse mode, pointers, dynamic memory, checkpointing

1 Introduction

Computing derivatives of a numerical model $F : X \mapsto Y : \mathbb{R}^n \mapsto \mathbb{R}^m$, given as a computer program P , is an important but also computation-intensive task. Automatic differentiation (AD) [4] in *adjoint* (or *reverse*) mode provides the means to obtain gradients and is used in many science and engineering contexts (refer to [4], the recent conference proceedings [3, 2], and the AD community website [1]). W.l.o.g we will assume $m = 1$, that is the computation of a gradient. Two major groups of AD tool implementations are operator overloading tools and source transformation tools. The latter are the focus of this paper. As a simplified rule, for each intrinsic floating-point operation ϕ (e.g., addition, multiplication, sine, cosine) that is exe-

Laurent Hascoët
Team Tropics, INRIA Sophia-Antipolis, France, laurent.hascoet@inria.fr

Jean Utke
Argonne National Laboratory / The University of Chicago, IL, USA, utke@mcs.anl.gov

cuted during run time in P as the sequence

$$[\dots, j : (u = \phi(v_1, \dots, v_k)), \dots], \quad j = 1, \dots, p, \quad (1)$$

of p such operations, the generated adjoint code has to implement the following sequence that reverses the original sequence in j :

$$[\dots, j : (\bar{v}_1 + = \frac{\partial \phi}{\partial v_1} \bar{u}, \dots, \bar{v}_k + = \frac{\partial \phi}{\partial v_k} \bar{u}), \dots], \quad j = p, \dots, 1, \quad (2)$$

with incremental assignments of adjoint variables \bar{v} for each argument v of the original operation ϕ . An AD source transformation tool creates \tilde{P} from P to compute the desired gradient \bar{X} . Making available the original variables required by the partials $\frac{\partial \phi}{\partial v_i}$ in (2) in the correct order leads to the use of schemes wherein sections of P are recomputed from checkpoints. Therefore \tilde{P} will not consist simply of (1) followed by (2). Program resources such as dynamic memory can be acquired and released at any point in (1). The appropriate action in (2), the handling of dynamic memory and pointers in general when splicing together subsequences of (1) and (2) are the subject of this paper.

2 Adjoint Data Association and Memory References

An implementation of (2) has to ensure that each original value v is associated with the adjoint \bar{v} . Several values v may be stored in a program variable v during its life span. Association with \bar{v} happens via the program variables and the two common approaches are *association by address* and *association by name*. The former pairs (v, \bar{v}) into a new *active type* for v while the latter assumes that the adjoint values \bar{v} are held in a variable with a generated name (e.g. here $v_$ for brevity) following a naming scheme that guarantees an unambiguous association with the original variable v . In either case, if a reference expression in (1) uses pointers then the same expression in (2) or its adjoint counterpart both use the same pointer mechanism, and therefore pointer values have to be recovered. Pointer values¹, however, generally are *offsets to base addresses* (e.g. the base address of a stack frame) and can change from (1) to (2). This may happen to memory on the stack or *stack variables*² which are automatically relinquished at the end of their scope in (1) and automatically reinstated – possibly at a different location – when their scope reopens in (2). It also happens for memory on the heap or *heap variables* that is deallocated explicitly during (1) and reallocated explicitly during (2) with no guarantee of invariance of the base address. The following subsections aim at categorizing the memory references according to restoration options.

¹ A pointer value may be more than a mere address. A Fortran90 pointer may define dimension slices, that must be recovered too. This question is only loosely related to our subject.

² sometimes, implying Fortran77, mischaracterized as statically allocated variables;

2.1 Categories of memory references

In the following we will see that similar mechanism are used to treat memory references to the heap and the stack. We therefore start with the basic categorization of stack variables.

C1.a: a named memory reference to a stack address that has a fixed offset from the frame base pointer; the memory reference to the adjoint value is done reusing the name; examples are scalar local/global variables, structure instances, references to local/global array variables with constant index, C++ references instantiated to refer to the above, and C99 variable-length arrays;

C1.b: memory referenced by items in C1.a, accessed via dereferencing a nested life-span quasi-constant instantiated³ pointer; analysis needs to establish that the pointer is quasi-constant (single assignment) and the scope of the pointee encloses the scope of the pointer; the memory reference to the adjoint value is done by replicating the single assignment at pointer instantiation during (2) followed by replicating the pointer dereference;

Both C1 sub-categories for phases (1) and (2) are illustrated in Fig. 1 using association by name. One can see that for the pointer parameter *p* of *f00* the scope condition with respect to *x* is trivially satisfied.

C2: memory references as in C1 with an additional non-constant offset; the memory reference to the adjoint value is done by recovering the non-constant offset and the actions indicated for C1.a or C1.b, respectively; see Fig. 2 (left half).

C3: a named memory reference to an address in the heap with an optional non-constant offset; the memory reference to the adjoint value is done by restoring the size information for the heap memory, reallocating to the corresponding name; restoring the non-constant offset and replicating the memory reference expression; The most important example is the Fortran *allocatable* array illustrated in Fig. 2 (right half). Note that this Fortran construct has syntactic scope and its base address

real :: x(10)	
...	
... x(3)... !reference	C1.a:(1)
real :: x_(10)	
...	
... x_(3)... !adjoint	C1.a:(2)
float f00(float* p)	
{ return (*p)*2; }	
float x,y;	
y=f00(&x);	C1.b:(1)
void f00_(float* p_,float r_)	
{ (*p_)+=(r_)*2; }	
float x_,y_;	
f00_(&x_,y_);	C1.b:(2)

Fig. 1 Example illustrations for categories C1.a in Fortran and C1.b in C++.

void f00(float* a,int j) { float r[10]; ...; i=f(j); ... *(a+i)=r[i];}	void f00_(float* a_,int j) { float r_[10]; // recover i r_[i]+=(a_+i); ...}	subroutine f00(n) real, allocatable :: x(:) ...; allocate(x(n));... ... x(j)... !reference	subroutine f00_(n) real, allocatable :: x_(: ...; !recover n allocate(x_(n)) ...; !recover j ... x_(j)...
C2:(1)	C2:(2)	C3:(1)	C3:(2)

Fig. 2 C++ example for C2 memory references **(a+i)* and *r[i]* (left half) and a Fortran example for a C3 memory reference *x(j)* (right half) for the respective phases (1) and (2).

³ initialized at declaration time

(in the heap) is always accessible via the given name⁴.

C4: a memory reference via a generic computed stack address (i.e. C2 without the nested scope and pointer instantiation requirement); the memory reference to the adjoint value is done (i) under the optimistic assumption of invariant offsets in the stack between executing (1) and (2) by restoring the absolute address value or (ii) under the pessimistic assumption of varying offsets in the stack by Alg. 2;

Even when we can assert that the path in the call tree when running (2) replicates the call tree path when running (1), the optimistic assumption (i) does not hold true for the general case because of possible compiler optimizations such as inlining and slicing, because only Fortran(77) has the potential to keep the base address of a stack frame invariant with respect to control flow, and because of implementation choices like separate adjoints, see also Sect. 2.2. The overhead for Alg. 2, however, may warrant the additional testing to ascertain the validity of the optimistic assumption for certain Fortran codes in conjunction with the encapsulated adjoint approach.

C5: a pointer reference to unnamed heap memory or via a generic computed address that cannot be restricted to one of the previous categories; the memory reference to the adjoint is done via Alg. 2. This is the typical category for memory allocated with `malloc` or `new` or Fortran’s pointer `allocate` where no syntactic scope is enforced on the allocated memory chunk and therefore, in contrast to C3, no guarantee can be given that it always be accessible by a single name.

2.2 Options to recover addresses

As indicated in Sect. 2.1 in C4 one may make optimistic or pessimistic assumptions regarding invariance of stack offsets. More generally, recording only an absolute address for memory references C4 and C5 is insufficient when the base address of the underlying memory changes between (1) and (2). If they vary, base addresses may be recorded along with the absolute addresses, into an *address map* during (1), and used in (2) for a run-time conversion discussed in Sect. 4. Its overhead is non-negligible, and therefore it is important to filter out all cases of the benign categories C1-3 (mostly by checking syntactic properties) and further identifying cases with invariant bases among categories C4 and C5. For categories C1-3 the recovery of addresses can simply be characterized as a recomputation that may involve taking the address of a stack variable or calling the Fortran `allocate` intrinsic.

Whether or not base addresses vary between (1) and (2) depends on the memory scope and implementation choices for the AD transformation. Certainly, any heap memory not deallocated before the last instruction of (1) will still be available in the relevant section of (2). Likewise, all stack variables (in particular global ones) at or above the stack frame of the “driver” subroutine that implements (without returning) the execution of (1) and (2) will not change base addresses. The latter can

⁴ The semantically closest C++ construct is a class whose constructor allocates memory and deallocates that memory in the destructor thereby giving the memory the syntactic scope of the class instance.

be identified by syntactic scope checking. In some cases heap memory can be made invariant simply by skipping over deallocation statements in (1). However, this has the obvious risk of memory leaks.

Otherwise, base addresses of memory references to the heap or stack can generally not be considered invariant. The reasons briefly mentioned in Sect. 2.1 under C4 are as follows. Typically, P consists of a set of subroutines s each implementing sections of statements in (1). An AD source transformation may elect to place the adjoint statements into a *separate adjoint* routine \bar{s} . This implies different stack offsets between variables in s and \bar{s} and therefore varying base addresses already for the Fortran77 memory model. Alternatively, the tool may create a modified \tilde{s} containing the original statements together with the *encapsulated adjoint* statements and use some control flow structure to decide which sequence of statements to execute. Assuming no stack changes are implied by compiler optimizations (such as inlining) this can yield invariant stack bases in the Fortran77 model. In C/C++ programs, variables declared in nested basic block scopes imply possibly different declaration scopes for the adjoint code compared to the original code thereby effectively shifting the base address even for encapsulated adjoints. Using pointer analysis the following information can be statically determined.

- $\text{Dests}_i(\mathfrak{p})$: the set of possible destinations (aka pointees) of pointer variable \mathfrak{p} at instruction i ;
- VaryingBase_i : for instruction i , an overestimated set of pointer variables with at least one target whose base address may change.

Thus, we can determine \mathfrak{p} at instruction i to have an invariant base if all $d \in \text{Dests}_i(\mathfrak{p})$ have an invariant base and can store the address. Instead of storing the address, similar to categories C1-3 we may under certain circumstances be able to recompute the value if there is a unique defining instruction. Using the above sets we present a data-flow analysis in Sect. 3 that enables an adjoint AD tool distinguish these scenarios and determine recomputation when possible.

3 An algorithm for Address Recovery

We propose an algorithm to detect the possibility of recovering addresses by exact repetition in (2) of some of the statements that compute the addresses in (1). This data-flow algorithm, shown in Alg. 1 for a Basic Block $I_{1:N}$, tracks uses of pointer variables in the source code of the original program P . Loosely following [6] the model for tracking dependencies is to consider for each instruction i the pairs (m, D) of a memory reference (expression) m occurring in i and the set D of possible defining instructions for the value held by m . This makes a suitable connection between program variables and the values they may hold. In turn, the different m each are mapped to a set of possible memory locations $l(m)$ such that m and m' may alias if $l(m) \cap l(m') \neq \emptyset$. The alias and reaching definitions analyses are not subject of

this paper and here just added for the occasional illustration and to achieve closure for nested indirection. Typically a pointer variable p is used in sequences as follows:

1. In instruction i the pointer variable p is assigned a value p , an address, computed from some variables in an expression using address arithmetic or other intrinsics such as an address-of operator or an `allocate` call. We set the pairs $(p, \{i\})$ and $(*p, \delta)$, δ being the set of the defining instructions of the dereferenced right-hand side if applicable. The right-hand side r of the assignment in i determines $l(p)$ as $l(r)$, e.g. $l(r) = \emptyset$ for $p = \text{NULL}$. We drop pairs (p, \cdot) $(*p, \cdot)$ representing any previous value held by p from propagation.
2. The pointer variable p is used
 - a. by address arithmetic, where the address value p is used but not dereferenced; the analysis refers to $(p, \{i\})$
 - b. by dereferencing p , e.g. $*p$, in computations that will be differentiated, making it required for (2); The analysis will also refer to $(*p, \delta)$.
3. For tracked pairs (p, \cdot) , $(*p, \cdot)$, an instruction i that implies any of the following scenarios may trigger the need to recover the pointer value at the corresponding adjoint of i :
 - a. p is overwritten (see item 1);
 - b. p goes out of scope (the scope exit is marked by a placeholder instruction);
 - c. the value held in $*p$ becomes "adjoint-dead", which means that none of the further uses, if any, are required for (2);
 - d. $*p$ becomes inaccessible, e.g. upon leaving the scope of a local stack variable whose address was assigned to p being pointer variable in an enclosing scope (cf. the scope condition in C2 vs. C4).

Considering these scenarios, the data flow algorithm in Alg. 1 propagates forward the following information:

ARI: (for "address recomputation instructions") a set of (m, D) pairs for pointer uses (see item 2 above), where D contains a single defining instruction i and that instruction's right-hand side value can be "cheaply" recomputed at the adjoint of the instructions mentioned in item 3. What is considered "cheap" depends on the implementation. Certainly, addresses computed for memory references of category C1 are considered cheap. This does not include allocation statements (C3 and C5) because they have to be handled by Alg. 2.

RecA $_{i,j}$: the j -th scheduled recomputation d for a pair $(m, \{d\})$ triggered by instruction i , to be prepended before \bar{i} . It is used in Alg. 2.

StoA $_i$: the set of pairs (m, D) scheduled for store/restore at instruction i/\bar{i} . The use is shown in Alg. 2.

StackTargets: a global set of the stack variables that must be managed by address mapping, see also Alg. 2.

Alg. 1 also uses the following data-flow information, precomputed backwards:

AdjLive $_i$: the set of pairs (m, D) whose value after instruction i is eventually needed for the derivative computation.

AdjOut_i : the set of pairs (m, D) that may be overwritten by the instructions following i in (1).

We rely on the “to be recorded” algorithm [5] integrated as lines 01, 02, 08, and 13 in Alg. 1. In line 02, before each instruction i , the algorithm adds the adjoint required pairs (i.e., $(m, D) \in \text{use}(\bar{i})$) to TBR. All pairs in TBR that i may overwrite (i.e. $(m, D) \in \text{out}(i)$) are added to StoA_i , see line 08, and all pairs completely overwritten by i are removed from TBR, see line 13. All pairs (m, D) required in (2) will be in TBR at some moment. An address recomputation $(m, \{d\})$ remains in ARI during

Algorithm 1 Address Recomputation algorithm through a Basic Block

Given $I_{1:N}$ and the sets TBR, ARI either from previous basic block or \emptyset

```

01 for  $i = I_1$  to  $I_N$ 
02   TBR := TBR  $\cup$  use( $\bar{i}$ )
    // Schedule appropriate address recomputation before  $\bar{i}$ :
03    $j := 0$ 
04   while [  $\exists p \equiv (m, \{d\}) \in \text{ARI} \setminus \text{AdjLive}_i \mid$ 
    ( $\forall (m', \{d'\}) \in \text{ARI} : p \notin \text{use}(d')$ )  $\vee$  ( $\text{use}(d) \cup \{p\} \cap \text{out}(i) \neq \emptyset$ ) ]
05     RecA $_{i,j++} := d$ 
06     TBR := (TBR  $\setminus$  { $p$ })  $\cup$  use(ARI( $p$ ))
07     ARI := ARI  $\ominus$  { $p$ }
    // Use storage as last resort for pointers about to be overwritten:
08     StoA $_i := \text{TBR} \cap \text{out}(i)$ 
09      $\forall (m, D) \in \text{StoA}_i$  if  $m$  is a pointer then StackTargets  $\cup =$  Dests $_i(m)$ 
10     ARI := ARI  $\ominus$  out( $i$ )
    // Collect  $i$  as potential address recomputation candidate:
11     if [ ( $i$  assigns pointer  $m$ )  $\wedge$ 
    ( $m, \{i\} \in \text{AdjLive}_i \cap (\text{AdjOut}_i \cup \text{VaryingBase}_i) \setminus \text{use}(i)$ ) ]
12       ARI := ARI  $\cup$  { $(m, \{i\})$ }
13     TBR := TBR  $\setminus$  kill( $i$ )

```

propagation unless it is invalidated by i , i.e. i overwrites m or something used by d . Operation $\text{ARI} \ominus (m, D)$ effectively removes from ARI all recomputations invalidated by the instructions in D . Also, control flow merges (not shown in Alg. 1) imply forming the union of defining instruction sets but only pairs with single definitions remain in ARI.

The condition on line 04 triggers the recomputation as soon as p turns adjoint-dead ($p \notin \text{AdjLive}_i$) and still can be recomputed ($p \in \text{ARI}$); recomputation is scheduled at the corresponding step into (2) and is removed from ARI and from TBR. The condition also tests that the defining instruction d , because it is moved to instruction \bar{i} , can in fact still be executed. The condition also manages the order between recomputation of different pointers. Line 08 finally triggers the last resort store/restore mechanism. Alg. 2 covers the conversion of addresses from (1) to (2). Thus, Alg. 1 gives priority to recomputation over store/restore by the chosen order of actions on ARI and StoA.

4 An algorithm for Address Mapping

For a program symbol v let b_v denote the base address and e_v the end address of v (and v_- the associated adjoint symbol). If the program symbol p is a pointer, then let p denote the pointer value (i.e. the address) in the range test referenced in the following Alg. 2. Lines 18 and 20 of Alg. 2 assume that pointer arithmetic is available in the target language. If pointer arithmetic is unavailable one could push/pop along with (b, e) the number of allocated instances of the given type.

```
int foo(float*p, const int n) {
float *pr=p+n; int rc=0;
while (p<pr && *p<3.0) ++p;
if (p<pr) rc=1; // in range
return rc; }
```

Fig. 3 Marching Pointers go out of range

Algorithm 2 Code generation for runtime address mapping and heap memory handling for in-range pointers for subroutine s and its adjoint \bar{s}

Given StoA_i , StackTargets and global⁵ sets M, \bar{M} prepopulated with $(0, 0) \in M$ and $(0, 0, 0) \in \bar{M}$:

```
01  $\forall v \in \text{StackTargets}$ :
02   in  $s$  generate code
03     on entry: add  $(b_v, e_v)$  to  $M$ 
04     on exit: append  $\text{PUSHRANGE}(b_v, e_v)$ ; remove  $(b_v, \cdot)$  from  $M$ 
05   in  $\bar{s}$  generate code
06     on entry: prepend  $\text{POPRANGE}(b_v, e_v)$ ; add  $(b_v, e_v, b_{\bar{v}})$  to  $\bar{M}$ 
07     on exit: remove  $(\cdot, \cdot, b_{\bar{v}})$  from  $\bar{M}$ 
08  $\forall i \in [I_1, \dots, I_N]$ :
09   if  $i$  allocates memory range  $(b, e)$  :
10     in  $s$  generate code to add  $(b, e)$  to  $M$ 
11     in  $\bar{s}$  generate code to remove  $(\cdot, \cdot, \bar{b})$  from  $\bar{M}$ ; deallocate( $\bar{b}$ )
12   if  $i$  deallocates memory range  $(b, e)$  :
13     in  $s$  after  $i$  generate code to remove  $(b, e)$  from  $M$ ;  $\text{PUSHRANGE}(b, e)$ 
14     in  $\bar{s}$  generate code to  $\text{POPRANGE}(b, e)$ ;  $\bar{b} = \text{allocate}(e - b)$ ; add  $(b, e, \bar{b})$  to  $\bar{M}$ 
15   Prepend all recomputation instructions  $\text{RecA}_{i,j}$ , for  $j = J - 1 \rightarrow 0$ , before  $\bar{i}$  in  $\bar{s}$ 
16    $\forall (p, \cdot) \in \text{StoA}_i$ :
17     in  $s$  before  $i$  successively append generated code for:
18       if  $\exists (b^*, e^*) \in M : p \in [b^*, e^*]$  then  $\text{PUSH}(p)$  else abort
19     in  $\bar{s}$  before  $\bar{i}$  successively prepend generated code for:
20        $\text{POP}(p)$ ; find  $(b^*, e^*, \bar{b}^*) \in \bar{M} : p \in [b^*, e^*]$ ;  $p_- = \bar{b}^* + (p - b^*)$ 
```

As indicated in line 18, the algorithm will fail if the pointer value in question is not initialized to 0 and instead has some random value or is outside of any of the ranges registered in M . This last case can be constructed in C/C++, see Fig. 3, as can be a case where such a pointer then happens to fall into another (wrong) range. The Fortran pointer semantic is fortunately more restrictive and does not as easily permit a pointer out of range scenario. This scenario can, however, reliably be tackled only with run time base address and offset tracking shown in Alg. 3. Each useful pointer value in a program is computed by applying some offset either to a base address. The idea is simply to generate for each pointer variable p occurring in the StoA_i a pair (p_b, p_o) of base address and offset. The base address p_b can be either the address of

a named stack variable, the address yielded by dynamic memory allocation (e.g. via `new`) or 0 which is also the default initializer. Any computed address assigned to a pointer $p = q + o$ can be expressed as another address, the value of pointer q , plus an offset o . The pair (p_b, p_o) for the assignment $p = q + o$ is updated as follows:

$$(p_b, p_o) = \begin{cases} (0, 0) & \text{if } q_b \equiv 0 \quad (\text{to imply 0 offsets } q_o \equiv o \equiv 0) \\ (q_b, q_o + o) & \text{otherwise} \end{cases} \quad (3)$$

Using these pairs we can now formulate a modified algorithm that does not require pointer values to remain in a single valid range.

Algorithm 3 Code generation for runtime address mapping and heap memory handling with run time offset tracking.

Given StoA_i , StackTargets and global⁶ sets M, \bar{M} the algorithm is the same as Alg. 2 except for the following changes. Within the loop implied by line 08 add:

```
08+1   if  $i$  assigns to a pointer  $p$ :
08+2   in  $s$  before  $i$  prepend code to update  $p_b$  and  $p_o$  according to (3)
```

and change lines 18 and 20 as follows:

```
18*   find  $(b^*, e^*) \in M : p_b = b^*$  then PUSHPAIR  $(p_b, p_o)$ 
20*   POPPAIR  $(p_b, p_o)$ ; find  $(b^*, e^*, \bar{b}^*) \in \bar{M} : p_b = b^*$ ;  $p_- = \bar{b}^* + p_o$ 
```

5 Bookkeeping for Checkpoints

For handling the checkpointing of pointers and heap memory one might start with the simple criterion that the purpose of checkpointing must enable recovery not only of all values and but also of pointers to start computation from said checkpoint. As opposed to checkpoints for fault tolerance the reverse mode tradeoff requires to keep around as many checkpoints as one can afford. It is imperative to minimize their size which typically excludes system-level checkpointing and the tools geared toward it. Instead, one uses incremental (application-level) checkpoints created either manually by the user or automatically by the AD tool. Restoring an incremental checkpoint implies it is done inside of a running process. This implies solving problems similar to the forward to reverse sequence mapping of stack and heap addresses. In particular, we can attach a given checkpoint to a (subroutine call) instruction i for which side effect analysis cumulatively determines all the data modified by the instruction and thereby cumulatively determines StoA_i . Then, instead of generating stack push and pop calls referenced in Alg. 2 and Alg. 3 lines 18, 18*, 20, and 20*, respectively, the system generates checkpoint write and read calls. In addition, the checkpoint will also have to contain the current state of the global map M at the checkpoint instruction. To facilitate address mapping for stack addresses during checkpoint restoration we also require the top-down cumulative StackTargets_c and

an additional field δ in M that discriminates between heap and stack memory and, for Fortran, may contain an `allocatable` variable name where applicable. Then, upon restoring state from the checkpoint, one first retrieves the checkpointed map into M_c . The difference in the memory state is reflected in the difference between the current state of M and the restored copy M_c . From this difference Alg. 4 populates M' . Note that for lines 4 and 6, if δ indicates an `allocatable` variable then the

Algorithm 4 Code generation for checkpoint restoration.

Given M, M_c , and StackTargets_c , start with $M' = \emptyset$ and generate code for:

```

1   $\forall (b, e, \delta) \in M \cap M_c:$ 
2   $M' := M' \cup \{(b, e, b, \delta)\}$ 
3   $\forall (b_c, e_c, \delta) \in M_c \setminus M:$ 
4  if ( $\delta$  is heap) then  $b' := \text{allocate}(e_c - b_c); M' := M' \cup \{(b_c, e_c, b', \delta)\}$ 
5   $\forall (b, \cdot, \delta) \in M \setminus M_c:$ 
6  if ( $\delta$  is heap)  $\wedge \exists (\cdot, \cdot, b) \in \bar{M}$  then deallocate( $b$ )
```

Fortran `allocate` and `deallocate` statements will have to be all generated at compile time for symbols in question and their execution be made dependent on the condition in lines 3 and 5, respectively. Given M' , Alg. 2 can be easily adapted to produce a modified recomputation routine s' with M' in place of \bar{M} and the adjustments for forward instead of reverse statement order.

An implementation of the algorithms discussed in this paper is not yet available but will be undertaken by the authors in the AD tools Tapenade and OpenAD/ADIC.

Acknowledgements This work was supported by the U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

1. AD community website: <http://www.autodiff.org>
2. Bischof, C.H., Bücker, H.M., Hovland, P.D., Naumann, U., Utke, J. (eds.): Advances in Automatic Differentiation, *Lecture Notes in Computational Science and Engineering*, vol. 64. Springer, Berlin (2008). DOI 10.1007/978-3-540-68942-3
3. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50. Springer, New York, NY (2005). DOI 10.1007/3-540-28438-9
4. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd edn. No. 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA (2008). URL <http://www.ec-securehost.com/SIAM/OT105.html>
5. Hascoët, L., Naumann, U., Pascual, V.: “To be recorded” analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems* **21**(8), 1401–1417 (2005). DOI 10.1016/j.future.2004.11.009
6. OpenAnalysis: <http://openanalysis.berlios.de/>