Implicit Memory Management for SAC

Clemens Grelck and Kai Trojahner

University of Lübeck
Institute of Software Technology and Programming Languages
{grelck,trojahne}@isp.uni-luebeck.de

Abstract. While almost all functional languages rely on garbage collection for implicit memory management, the needs of efficient array processing are better suited by reference counting. The opportunities to reclaim unused memory immediately and to implement functionally sound array operations by destructive in-place updates turn out to be essential for approaching the performance achieved by imperative languages.
In this paper we outline the realization of implicit memory management in the functional array language SAC. Starting with basic schemes for the introduction of memory management and reference counting instructions into SAC code, the emphasis is laid on a range of optimizations that aim at reducing runtime overhead and exploiting memory reuse opportunities.

1 Introduction

Implicit memory management is an essential feature of functional programming languages. The two basic approachess to organize the automatic reclamation of memory — reference counting [2] and garbage collection [9] — were already invented in the early years of computer science. Since then they have intensively been studied and refined in many ways. For surveys see for example [11] and [1].

The basic idea behind reference counting is to augment any data object by a reference counter, which keeps track of the number of conceptual copies. The counter must be incremented and decremented as the data object is passed as argument to functions. As soon as the reference counter is decremented to zero, the data object can be de-allocated, and the associated storage may immediately be reclaimed for further usage.

With a garbage collector memory is allocated by simply pushing a pointer forward until all memory is exhausted or some limit is reached. If no further allocation is possible, normal program execution is interrupted by a garbage collection phase. The garbage collector identifies and tags all live data objects by systematically traversing the heap starting from certain root addresses, e.g. references on the runtime stack. Afterwards, all non-tagged data objects are considered garbage and removed.

Both methods have their advantages and disadvantages. Reference counting incurs space overhead for storing the counter and time overhead for its manipulation. De-allocation of a data object manipulates the counters of all referenced

data objects, which may lead to expensive recursive de-allocation cascades. Positive aspects of reference counting are that memory management overhead is evenly distributed over the execution time of a program and storage is reclaimed as soon as possible.

Garbage collection only reclaims storage if needed, but in that case may cause long disruptive pauses in the execution of the program itself. Data objects are only reclaimed after a potentially long zombie time increasing overall memory demand. The complexity of the collection process is in the order of the size of the life heap, not in the order of the space to be reclaimed, i.e., performance degrades with high memory pressure. However, if memory is not too short and data objects are frequently allocated and have short average life time, garbage collection offers very space and time efficient implicit memory management.

Functional programs usually fall into the latter category. They predominently process list- and tree-like data structures made up of large numbers of small data objects connected by references. For good reasons almost all functional programming languages rely on some kind of garbage collector for memory management. However, the situation changes completely if not lists and trees are the prevailing data structure but (large) arrays of numerical data. Efficient application of garbage collection relies on keeping program interrupts as infrequent as possible. This inevitably leads to long zombie periods between the time a data object becomes garbage and the time its storage is actually reclaimed. If heap space is requested in large portions, zombie times may quickly become prohibitive.

In contrast, the disadvantages of reference counting are mostly avoided in the context of arrays. As soon as array elements are unboxed, which anyways is mandatory for performance reasons, expensive de-allocation cascades cannot occur. Memory overhead for storing the reference counter associated with each data object is negligible with respect to the average size of an array. Likewise, the runtime overhead for manipulating the reference counter is small with respect to the time required to compute the element values of an array.

Furthermore, knowing the exact number of conceptual copies of any data structure at runtime exposes a significant opportunity for optimization. If an argument to some operation is known to become garbage afterwards, the associated storage may be reclaimed immediately to accomodate the result, provided that the operation is suitable for such a short cut. In addition to avoiding the overhead incurred by a de-allocation/allocation cycle, this has the potential to overcome or at least to mitigate the impact of the aggregate update problem [8]. Arrays may be updated *destructively* or *in-place* whenever the reference counter indicates that this is save to do without violating the functional semantics. This turns out to be a key feature for approaching the runtime performance of imperative code. Therefore, functional array languages like SISAL [3] or SAC [10] employ reference counting instead of garbage collection.

In contrast to garbage collectors, which are only loosely connected to an application program, reference counting requires tight integration into compiled code. Memory allocation and de-allocation requests as well as reference counter manipulating instructions must be inserted into compiled code with great care.

In the following, we present various compilation and optimization schemes for this purpose. Although they have been developed in the context of SaC, they are only specific to the setting of functional array processing, but do not rely on any specific features of the language.

Since development of a correct reference counting scheme is even more difficult than doing explicit memory allocation/de-allocation in the right way, we have split the process into several smaller tasks. In a first step, the purely functional world of values (*Val*) is extended by two new categories that make the notion of memory explicit: plain storage (*Mem*) and storage holding a value (*MemVal*). Specific instructions are introduced that request pieces of memory and connect values to storage. In a separate step, numbers of references are inferred, and reference counter manipulating instructions are inserted into the code. Both schemes are kept as simple as possible, deferring any kind of optimization to subsequent steps.

Two different kinds of optimizations can be distinguished. To begin with, superfluous reference counting instructions are eliminated in order to minimize the overhead associated with keeping track of numbers of references. Afterwards, potential candidates for immediate storage reuse are identified. In some cases, a candidate can be adopted at compile time, but generally the decision which candidate to take — if any — must be postponed until runtime. In addition, update-in-place opportunities are analysed, and tailor-made code is generated.

The remainder of the paper is organized as follows. In Section 2, we define a simplified core language of SaC called $\text{SaC}_{\mathsf{mini}}$, which exhibits all features relevant for memory management, but dispenses with any other details. Sections 3 and 4 introduce explicit memory allocation requests into $\text{SaC}_{\mathsf{mini}}$ programs and describe basic optimizations on them. A scheme for adding reference counting instructions is defined in Section 5. Sections 6 and 7 sketch out the range of optimizations. Eventually, Section 8 draws conclusions and outlines directions of future work.

2 $\text{SaC}_{\mathsf{mini}}$

SaC is a sophisticated functional array language. A comprehensive introduction can be found in [10]; case studies about programming style and runtime performance are presented in [4, 6, 5]. However, when reasoning about aspects of implicit memory management, many features of SaC turn out to be irrelevant or to superfluously complicate the situation. Therefore, we define a core language called $\text{SaC}_{\mathsf{mini}}$, which contains all features essential for memory management, but not more. Fig. 1 shows a pseudo syntax definition of $\text{SaC}_{\mathsf{mini}}$.

Basically, a $\text{SaC}_{\mathsf{mini}}$ program is a sequence of potentially mutually recursive function definitions. A function definition consists of a function name, a parameter list in brackets and a code block. In contrast to full SaC, the notion of types is entirely omitted. A code block is a sequence of assignments followed by a `return`-statement enclosed in curly brackets. This is equivalent to a nesting of let-expressions with a subsequent goal expression mainstream functional

```
Program     ⇒ [ FunDef ]*
FunDef      ⇒ Id ( [ Id [ , Id ]* ] ) Block
Block       ⇒ ExprBlock | CondBlock
ExprBlock   ⇒ { [ Assign ]* Return }
CondBlock   ⇒ { [ Assign ]* Cond }
Return      ⇒ return ( Id ) ;
Cond        ⇒ if ( Id ) Block else Block
Assign      ⇒ Id = Expr ;
Expr        ⇒ Const | Id | FunAp | PrfAp | With
FunAp       ⇒ Id ( [ Id [ , Id ]* ] )
PrfAp       ⇒ Prf ( [ Id [ , Id ]* ] )
With        ⇒ with ( Id ) [ Part ]+ genarray ( Id , Id )
Part        ⇒ Generator : ExprBlock
Generator   ⇒ ( Id <= Id < Id )
```

Fig. 1. Syntax of SAC$_{mini}$

languages. Likewise, conditionals are introduced. In addition to constants and identifiers, expressions may be applications of defined or of built-in functions. Since individual properties of built-in functions are irrelevant for our presentation of memory management, we do not elaborate on them. Furthermore, SAC$_{mini}$ comes without nested expressions, a property which can easily be achieved by simple preprocessing of SAC code.

Up to now, SAC$_{mini}$ is just a straightforward functional toy language. The only array-specific construct we introduce here is a substantially simplified version of SAC's array comprehensions, called WITH-loops. A WITH-loop of the form

```
with (iv) ... genarray( shp, default)
```

defines an array whose shape is given by ***shp***. Hence, ***shp*** must refer to an integer vector. Note that in SAC$_{mini}$ (just as in SAC) any expression denotes an array. Arrays are represented by two vectors: a *shape vector* and a *data vector*. The length of the shape vector defines the dimensionality or rank of the array while each element specifies the array's extent along the corresponding axis. The data vector solely contains the array's elements without any structural information. In this sense, scalars are rank zero arrays with an empty shape vector and a 1-element data vector. Arrays can be nested as long as the whole array remains representable by shape and data vector, i.e., all elements of an array must have the same shape.

The elements of a WITH-loop-defined array are either set to the ***default*** value or computed according to the specification given in one of the *parts*, depending on its index position. Each part consists of a *generator*, which defines a set of

index positions, and an associated expression block, which determines the values of array elements at index positions covered by the generator. In its simplest form a generator `( lb <= iv < ub )` defines a rectangular index range delimited by a lower bound vector `lb` and an upper bound vector `ub`. The WITH-loop-variable `iv` refers to the current index position; its scope is restricted to the corresponding expression block. For reasons of simplification all generators of a single WITH-loop must use the same WITH-loop-variable. This restriction is emphasized by introducing the WITH-loop-variable right behind the key word `with`.

Multiple parts allow to define different array elements according to different specifications. In order to ensure deterministic results, the index sets defined by the various generators of an individual WITH-loop must be pairwise disjoint.

3 Explicit memory allocation

While functional programs evaluate expressions in order to derive a value, machines compute results by executing sequences of state transforming operations. Hence, a compiler for a functional programming language must introduce a notion of memory during the compilation process.

In SAC$_{\mathsf{mini}}$, memory is introduced via a special memory language called SAC$_{\mathsf{mem}}$. SAC$_{\mathsf{mem}}$ distinguishes between the functionally pure values used in SAC$_{\mathsf{mini}}$ so far and the memory that is used to store them. This leads to three basic categories of variables. *Val* subsumes the values of SAC$_{\mathsf{mini}}$ expressions which are merely abstract description of arrays. *Mem* describes chunks of memory that can be used to store values. Finally, *MemVal* denotes the category of memory containing a value.

`alloc`:	$Val \longrightarrow Mem$
`fill`:	$Val \times Mem \longrightarrow MemVal$
`suballoc`:	$Mem \times Val \longrightarrow Mem$
`copy`:	$MemVal \longrightarrow Val$

Fig. 2. The four operations of SAC$_{\mathsf{mem}}$

As depicted in Fig. 2, SAC$_{\mathsf{mem}}$ consists of four built-in functions defining the operational behaviour of SAC$_{\mathsf{mini}}$ programs. The operation `alloc(` *shp*) allocates memory, where *shp* is an integer vector value describing the memory's shape. The shape may be known at compile time, but in general it is an expression that can be used to compute the shape at runtime. The `fill` operation initializes memory with a value of the appropriate shape, yielding a *MemVal*. In the context of SAC$_{\mathsf{mem}}$, the WITH-loop can be understood as closely related to the `fill` operation. The WITH-loop fills an array by filling each of its elements individually. This is expressed by the `suballoc` operation which yields the piece of memory at the position of the index vector from the WITH-loop's result array.

A compilation scheme for memory allocation in SAC$_{\mathsf{mini}}$ is shown in Fig. 3. As SAC$_{\mathsf{mini}}$ has eager semantics, all right hand side expressions are evaluated before being assigned to a left hand side identifier. Straightforwardly, memory allocation

```
ALLOC [[ v = f( a1, ..., an);
         Rest                 ]] = { v = f( a1, ..., an);
                                     ALLOC [ Rest]

ALLOC [[ v = a;
         Rest   ]] = { v = a;
                       ALLOC [ Rest]

ALLOC [[ v = c;
         Rest   ]] = { v' = alloc( []);
                       v = fill( c, v');
                       ALLOC [ Rest]

ALLOC [[ v = prf( a1, ..., an);
         Rest                   ]]
    = { v' = alloc( shape(prf(a1,...,an)));
        v = fill( prf( a1, ..., an), v');
        ALLOC [ Rest]

ALLOC [[ A = with ( iv)
               ( lb1 <= iv < ub1) {
                 assigns1
                 return( res1);
               }
               ⋮
               ( lbn <= iv < ubn) {
                 assignsn
                 return( resn);
               }
              genarray( shp, def);
         Rest                          ]]
    = { iv = alloc( shape(lb1));
        A' = alloc( shp ++shape(def));
        A = with ( iv)
              ( lb1 <= iv < ub1) {
                ALLOC [ assigns1]
                sub1' = suballoc( A', iv);
                sub1 = fill( copy( res1), sub1');
                return( sub1);
              }
              ⋮
              ( lbn <= iv < ubn) {
                ALLOC [ assignsn]
                subn' = suballoc( A', iv);
                subn = fill( copy( resn), subn);
                return( subn);
              }
             genarray( shp, def, A');
        ALLOC [ Rest]
```

Fig. 3. Compilation scheme for explicit memory allocation in SAC$_{\mathsf{mini}}$.

instructions must be inserted before an assignment whenever the evaluation of the right hand side requires additional memory.

Assigning a constant scalar value to a variable needs one memory element. Its shape is represented by the empty shape vector and hence, `alloc([])` yields memory of the required size. In all other cases an expression describing the desired shape must be formed from the assignment's right hand side. For all built-in SAC$_{\mathsf{mini}}$ functions this expression can be deduced by wrapping a copy of the right hand side into an application of the shape function. In case of the WITH-loop the results shape can be computed by concatenating its shape parameter to the shape of the default element. Simply for technical reasons to facilitate specifications of subsequent transformation schemes, the identifier of the WITH-loop's result memory is annotated as another parameter of the WITH-loop. In addition to the result array, memory for the index vector must be allocated. By definition, its shape equals the boundary vectors' shape. The WITH-loop's *ExprBlock*s are compiled by recursively applying the allocation scheme. Afterwards, each block is appended by an application of `suballoc` and a combination of `fill` and `copy` which initializes the elements memory with the computed value.

No allocation instructions are needed in front of applications of user defined functions as these allocate memory an their own. The assignment of variables does not require allocation either as it describes the identity of these two variables, meaning they actually share the same memory.

4 Optimizing SAC$_{\mathsf{mem}}$

Because of its local scope, the allocation scheme for WITH-loops introduces a combination of `fill`/`copy` operations at the end of each of the WITH-loop's *ExprBlock*s. However, when the allocation of the copied array takes place inside of the same expression block, copying can be avoided.

An optimization called IN-PLACE-COMPUTATION aims at computing the array element's value in the correct piece of the WITH-loop's result memory. Its compilation scheme can be seen in Fig. 4. The optimization consists in replacing the initial allocation with the suballocation and eliminating the original `suballoc`, `fill` and `copy` combination.

5 Introducing reference counting instructions

The idea of reference counting is to keep track of the number of conceptual copies of a data object by means of a special object attribute, the so called *reference counter*. Once it becomes zero, the data object is no longer used and can be cleared from memory.

Reference counting of SAC$_{\mathsf{mem}}$ is done by enriching the program with a special reference counting language depicted in Fig. 5. The `set_rc` operation is used to initialize an object's reference counter. `inc_rc` increases a given data object's reference counter by a certain number. `dec_rc(` x`,` n`)` frees an object's memory if its reference counter equals n. Otherwise the counter is decremented

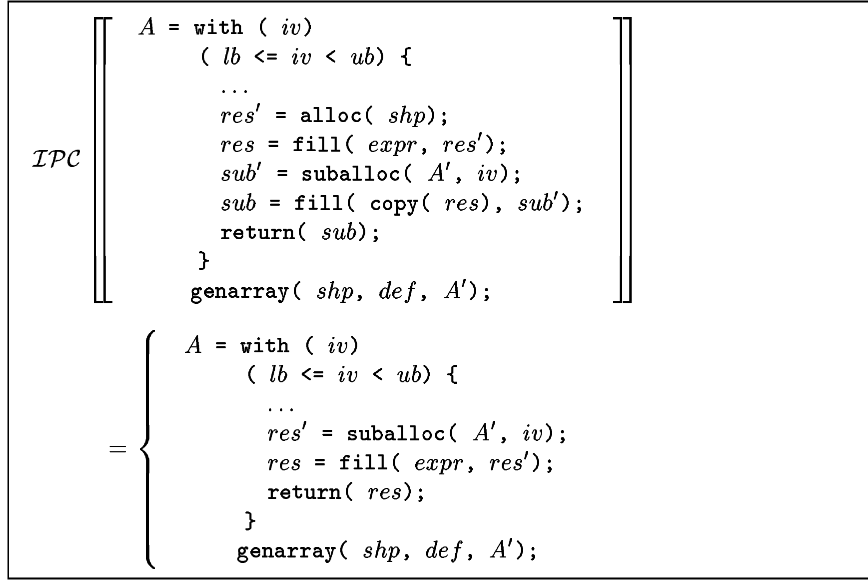

Fig. 4. Optimization scheme for In-Place Computation

by n. The pseudo operation `adjust_rc` is used to simplify the notation of the reference counting scheme.

Reference counting inference can be done using the deduction system shown in Fig. 6. Reference counting annotations are introduced for each function independently. The deduction system translates into a bottom-up traversal of a function body. In order to gather the necessary information needed to annotate the appropriate reference counting instructions, an environment is used which associatively maps identifiers to integers.

Deduction starts with with the return statement of an *ExprBlock*. Here the returned variable's environment is initialized with one as by definition a function returns an object with a reference counter of one. Consequently, the deduction rule for the application of a user defined function adjusts the reference counter of the returned variable v by $Env(v) - 1$. The importance of assuming each returned variable's reference counter to be one becomes evident in case of $Env(v)$ being zero, meaning v is not used in $Rest'$. In this case, `adjust_rc` will transform into a `dec_rc` statement which will possibly remove v from memory. As each parameter

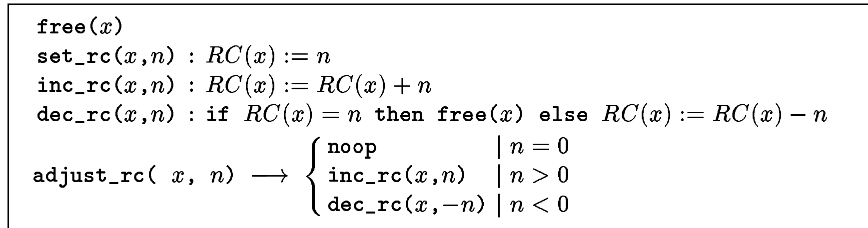

Fig. 5. Reference counting instructions

of a function application consumes one conceptual copy of the argument, the environment of each argument is increased by one. The deduction rule for the application of `fill` mimics this behaviour. Primitive function don't consume their arguments, they must consumed explicitly by means of a `dec_rc` statement. Additionally, the filled memory's reference counter is initialized with one in order

$$\frac{}{\texttt{return(}\ a\texttt{);} \longrightarrow \texttt{return(}\ a\texttt{);}\ | Env[a \leftarrow 1]}$$

$$\frac{Rest \longrightarrow Rest' | Env}{\begin{array}{l} v\ \texttt{=}\ f\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{);} \\ Rest \end{array} \longrightarrow \begin{array}{l} v\ \texttt{=}\ f\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{);} \\ \texttt{adjust_rc(}v\texttt{,}Env(v) - 1\texttt{);} \\ Rest' \end{array} \left| \begin{array}{l} Env[v \leftarrow 0] \\ {[a_i \leftarrow Env(a_i) + 1]} \end{array} \right.}$$

$$\frac{Rest \longrightarrow Rest' | Env}{\begin{array}{l} v\ \texttt{=}\ \texttt{fill(}c\texttt{,}m\texttt{);} \\ Rest \end{array} \longrightarrow \begin{array}{l} v\ \texttt{=}\ \texttt{fill(}c\texttt{,}m\texttt{);} \\ \texttt{adjust_rc(}v\texttt{,}Env(v){-}1\texttt{);} \\ Rest' \end{array} \left| \begin{array}{l} Env[v \leftarrow 0] \\ {[m \leftarrow 1]} \end{array} \right.}$$

$$\frac{Rest \longrightarrow Rest' | Env}{\begin{array}{l} v\ \texttt{=}\ \texttt{fill(}prf\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{),}m\texttt{);} \\ Rest \\ \quad \longrightarrow \begin{array}{l} v\ \texttt{=}\ \texttt{fill(}prf\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{),}m\texttt{);} \\ \texttt{adjust_rc(}v\texttt{,}Env(v) - 1\texttt{);} \\ \texttt{dec_rc(}\ a_i\texttt{,}\ 1\texttt{);} \\ Rest' \end{array} \left| \begin{array}{l} Env[v \leftarrow 0] \\ {[a_i \leftarrow Env(a_i) + 1]} \\ {[m \leftarrow 1]} \end{array} \right. \end{array}}$$

$$\frac{Rest \longrightarrow Rest' | Env}{\begin{array}{l} m\ \texttt{=}\ \texttt{alloc(}shape\texttt{);} \\ Rest \end{array} \longrightarrow \begin{array}{l} m\ \texttt{=}\ \texttt{alloc(}shape\texttt{);} \\ \texttt{set_rc(}m\texttt{,}\ 1\texttt{);} \\ \texttt{adjust_rc(}m\texttt{,}Env(m) - 1\texttt{);} \\ Rest' \end{array} \left| \ Env[m \leftarrow 0] \right.}$$

$$\frac{Rest \longrightarrow Rest' | Env}{\begin{array}{l} v\ \texttt{=}\ a \\ Rest \end{array} \longrightarrow \begin{array}{l} v\ \texttt{=}\ a \\ Rest' \end{array} \left| \ Env[a \leftarrow Env(a) + Env(v)] \right.}$$

$$\frac{B \longrightarrow B' | Env}{\begin{array}{l} f\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)}\ \texttt{\{} \\ \quad B \\ \texttt{\}} \end{array} \longrightarrow \begin{array}{l} f\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)}\ \texttt{\{} \\ \quad \texttt{adjust_rc(}\ a_i\texttt{,}\ Env(a_1) - 1\texttt{);} \\ \quad B' \\ \texttt{\}} \end{array} \left| \ Env \right.}$$

Fig. 6. Deduction system for reference counting inference.
(continued on next page)

$$\frac{B_{then} \longrightarrow B'_{then}|Env_{then} \qquad B_{else} \longrightarrow B'_{else}|Env_{else}}{\texttt{if } (p) \; \{ \; B_{then} \; \} \; \texttt{else} \; \{ \; B_{else} \; \} \longrightarrow \texttt{if } (p) \; \{ \; \texttt{dec_rc}(p,\,1); \; \texttt{adjust_rc}(v,\, Env_{then} - Env); \; B'_{then} \; \} \; \texttt{else} \; \{ \; \texttt{dec_rc}(p,\,1); \; \texttt{adjust_rc}(v,\, Env_{else} - Env); \; B'_{else} \; \} \;\Big|\; Env[p \leftarrow Env(p)+1]}$$

```
if (p)                 if (p) {
{                        dec_rc(p, 1);
  B_then                 adjust_rc(v, Env_then - Env);
}                        B'_then
else {          -->    }
  B_else               else {
}                        dec_rc(p, 1);
                         adjust_rc(v, Env_else - Env);
                         B'_else
                       }                                   | Env[p <- Env(p)+1]
```

where
$Env[v \leftarrow Max(Env_{then}(v), Env_{else}(v))|Env_{then}(v) = 0 \vee Env_{else} = 0]$
$[v \leftarrow Min(Env_{then}(v), Env_{else}(v))|Env_{then}(v) \neq 0 \wedge Env_{else} \neq 0]$

$$\frac{B_i \longrightarrow B'_i|Env_{with_i} \qquad Rest \longrightarrow Rest'|Env}{}$$

```
A = with ( iv)
      ( lb_1 <= iv < ub_1) {
        B_1
      }
      ...
      ( lb_n <= iv < ub_n) {
        B_n
      }
     genarray( shp, def, M);
Rest
```

```
      A = with ( iv)
            ( lb_1 <= iv < ub_1) {
              adjust_rc( v, Env_with_1);
              B_1
            }
            ...
            ( lb_n <= iv < ub_n) {
              adjust_rc( v, Env_with_n);
-->           B_n
            }
           genarray( shp, def, M);
      dec_rc( lb_i, 1);
      dec_rc( ub_i, 1);
      dec_rc( shp, 1);
      dec_rc( def, 1);
      dec_rc( iv, 1);
      dec_rc( v, 1); |∃i : Env_with_i(v) ≠ 0
      Rest'
```

$Env[lb_i \leftarrow Env(lb_i) + 1]$
$[ub_i \leftarrow Env(ub_i) + 1]$
$[shp \leftarrow Env(shp) + 1]$
$[def \leftarrow Env(def) + 1]$
$[iv \leftarrow Env(iv) + 1]$
$[v \leftarrow Env(v) + 1$
$\quad |\exists i : Env_{with_i}(v) \neq 0]$

Figure 6: Deduction system for reference counting inference.
(continued from previous page)

to signal that the memory is actually used. Before the reference counter of newly allocated memory can be adjusted (note that `adjust_rc` typically transforms into `noop` here), it must be initialized with one using `set_rc`. As pointed out in Section 3, an assignment like $v = a$ identifies objects v and a. This means all references to v are disguised references to a. Hence, $Env(a)$ must be increased by $Env(v)$. The final deduction rule of a function is about annotating counting instructions for the function parameters. Again, these are assumed to have an initial reference counter of one.

Reference counting for conditionals is somewhat more challenging as the numbers of references to variables in the two branches can differ. To overcome this inequality both branches are balanced out by inserting additional reference counting instruction at their beginning. Generally, it is more desirable to insert `inc_rc` instead of `dec_rc` operations as the latter contain an additional conditional. Hence, a good strategy is to perform reference counting inference in both branches and take the minimum of both environments as the global environment and put an `inc_rc` in front of the other branch. Unfortunately, this cannot be done if one of the environments equals zero as this would not result in the needed `dec_rc` instruction but in a `noop`. Thus, the maximum function must be used in these cases to determine the global environment of a variable.

Finally, the WITH-loop is reference counted in a way such that externally the parameters of the WITH-loop-construct and all the variables used inside the body are consumed. As no variable defined outside the WITH-loop must be freed from memory while evaluating the WITH-loop, the reference counters of all internally used variables must be increased at the beginning of each block.

6 Optimizations on reference counting code

The additional instructions needed for reference counting may introduce serious runtime overhead. In order to minimize this overhead, two techniques are applied that aim at stripping out superfluous reference counting instructions.

The first optimization exploits the fact that primitive functions and WITH-loops don't consume conceptual copies of variables on their own, but rely on `dec_rc` instructions to free these objects from memory. As an object cannot be freed from memory before the last reference to it, all the `dec_rc` instructions in between are redundant. These can be safely removed when the preceding `inc_rc` is adjusted accordingly.

The second step of optimization fuses `set_rc` instructions with subsequent `inc_rc` instructions such that reference counters are initialized with the correct value right from the start. Fig. 7 shows an illustrative example in which these techniques are applied.

7 Exploiting memory reuse

Besides allowing immediate deallocation of data objects, reference counting can help to overcome the aggregate update problem [8]. This is possible because the

```
a' = alloc( []);                     a' = alloc( []);
set_rc( a', 1);                      set_rc( a', 2);
a = fill( 1, a');                    a = fill( 1, a');
inc_rc( a, 3);
v' = alloc( [2]);                    v' = alloc( [2]);
set_rc( v', 1);              ==>     set_rc( v', 1);
v = fill( vec( a, a), v');           v = fill( vec( a, a), v');
dec_rc( a, 1);
dec_rc( a, 1);
r = f( v, a, a);                     r = f( v, a, a);
return(r);                           return(r);
```

Fig. 7. Example: Redundant reference counting instructions removal

reference counter always reflects the number of conceptual copies of an array. A reference counter of one indicates that the array will be no longer needed throughout the program and can be modified destructively. Again, support for memory reuse is introduced via a set of special instructions augmenting the syntax described so far. These instructions can be found in Fig. 8.

`alloc_or_reuse`:	$Val \times Val \times [MemVal]^{+}$	$\longrightarrow Mem$
`reuse`:	$MemVal \times Val$	$\longrightarrow Mem$
`isreused`:	$MemVal \times Mem$	$\longrightarrow Val$

Fig. 8. Operations introduced to support reuse

`alloc_or_reuse(` ***rc*, *shp*, *cand*** `)` is a replacement for an `alloc`/`set_rc` combination enabling dynamic reuse. It checks at runtime whether the reference counter of any of the reuse candidates given in ***cand*** equals one thus allowing the arrays content to be updated destructively. If a candidate has this desired property, it will be reused with its reference counter incremented by ***rc***. Otherwise, new memory of shape ***shp*** is allocated, and the associated reference counter is initialized to ***rc***.

In order for an array to be a reuse candidate, some conditions must be met. First of all, the candiate must have the same shape as the memory that otherwise would be allocated. Furthermore, the operation using the memory must only access the array in a pointwise manner, which means that each new value of an array element does only depend on the old value. Finally, it must only be tried to reuse an array before the last reference to it which because of the reference counting optimizations is always followed by a `dec_rc` operation.

Memory reuse can even be decided at compile time if in addition to all the above mentioned requirements the corresponding allocation is located in the current *ExprBlock* and the array has never been passed to a user defined function in between. This is expressed by `reuse(` ***a*, *rc*** `)`.

Once reuse candidates are annotated it is possible to create specialized code that takes reuse into account in order to exploit possible data reuse. The predicate `isreused` allows to dynamically choose the best code for a given reuse

situation. In the example given in Fig. 9 it is shown, how the knowledge about memory reuse can avoid substantial runtime overhead.

```
A'= alloc_or_reuse(1,shp,B);
A = with (iv)
     ( lb1 <= iv < ub1) {
       res'= suballoc(A',iv);
       res = fill(B[iv],res');
       return(res);
     }
     ( lb2 <= iv < ub2) {
       res'= suballoc(A',iv);
       res = fill(1,res');
       return(res);
     }
    genarray(shp,def,A');
```

⇒

```
A'= alloc_or_reuse(1,shp,B);
if (isreused( B, A') {
  A = with (iv)
       ( lb1 <= iv < ub1) {
         return();
       }
       ( lb2 <= iv < ub2) {
         res'= suballoc(A',iv);
         res = fill(1,res');
         return(res);
       }
      genarray(shp,def,A');
}
else {
  A = with (iv)
       ( lb1 <= iv < ub1) {
         res'= suballoc(A',iv);
         res = fill(B[iv],res');
         return(res)
       }
       ( lb2 <= iv < ub2) {
         res'= suballoc(A',iv);
         res = fill(1,res');
         return(res);
       }
      genarray(shp,def,A');
}
```

Fig. 9. Example for exploiting memory reuse

8 Conclusion and future work

In this paper we have described code transformation schemes that introduce the notion of memory into intermediate SAC code. Purely functional code is augmented with instructions for memory allocation and administration of reference counters. A range of optimization techniques is sketched out that aim at reducing reference counting overhead and at identifying opportunities for immediate memory reuse and destructive realizations of array operations.

The various optimizations succeed in a substantial reduction of reference counting overhead compared with an initial straightforward solution. Likewise, exploitation of reuse opportunities turns out to be a key performance issue in functional array processing. Unfortunately, only a few of the optimizations are incorporated into the SAC compiler for the time being. Hence, a quantitative

analysis of the various optimizations' impact on performance must be postponed to future work. In addition to immediate implementation and experimentation work, a long term goal is to combine the advantages of reference counting and garbage collection in a unified memory management framework.

References

1. Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Collection Schemes for Distributed Garbage. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management (IWMM'92), St. Malo, France*, volume 637 of *Lecture Notes in Computer Science*, pages 43–81. Springer-Verlag, 1992.
2. George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, 1960.
3. J.-L. Gaudiot, W. Böhm, T. DeBoni, J. Feo, P. Miller, and W. Najjar. The Sisal Model of Functional Programming and its Implementation. In *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs'97), Aizu-Wakamatsu, Japan, March 1997.*, mar 1997.
4. C. Grelck. Implementing the NAS Benchmark MG in SAC. In Viktor K. Prasanna and George Westrom, editors, *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.
5. C. Grelck and S.-B. Scholz. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.
6. C. Grelck and S.-B. Scholz. Towards an Efficient Functional Implementation of the NAS Benchmark FT. In V. Malyshkin, editor, *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia*, volume 2763 of *Lecture Notes in Computer Science*, pages 230–235. Springer-Verlag, Berlin, Germany, 2003.
7. C. Grelck, S.-B. Scholz, and K. Trojahner. With-Loop Scalarization: Merging Nested Array Operations. In P. Trinder and G. Michaelson, editors, *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2004. Accepted for publication.
8. P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL'85), New Orleans, Louisiana, USA*, pages 300–313. ACM Press, 1985.
9. John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Communications of the ACM*, 3(4):184–195, 1960.
10. S.-B. Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.
11. Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management (IWMM'92), St. Malo, France*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer-Verlag, 1992.