Axis Control in SAC

Clemens Grelck[1] and Sven-Bodo Scholz[2]

[1] University of Lübeck
Institute of Software Technology and Programming Languages
grelck@isp.uni-luebeck.de
[2] University of Kiel
Institute of Computer Science and Applied Mathematics
sbs@informatik.uni-kiel.de

Abstract. High-level array processing is characterized by the composition of generic operations, which treat all array elements in a uniform way. This paper proposes a mechanism that allows programmers to direct effects of such array operations to non-scalar subarrays of argument arrays without sacrificing the high-level programming approach. A versatile notation for axis control is presented, and it is shown how the additional language constructs can be transformed into regular SaC code. Furthermore, an optimization technique is introduced which achieves the same runtime performance regardless of whether code is written using the new notation or in a substantially less elegant style employing conventional language features.

1 Introduction

SaC (Single Assignment C) [19] is a purely functional programming language, which allows for high-level array processing in a way similar to Apl [11]. Programmers are encouraged to construct application programs by composition of basic, generic, shape- and dimension-invariant array operations, typically via multiple intermediate levels of abstraction. As an example take a SaC implementation of the L2 norm:

```
double L2Norm( double[*] A)
{
  return( sqrt( sum( A * A)));
}
```

.

The argument type `double[*]` refers to double precision floating point number arrays of any shape, i.e., arguments to `L2Norm` can be vectors, matrices, higher-dimensional arrays, or even scalars, which in SaC like in Apl or J are considered 0-dimensional arrays. The same generality applies to the main building blocks `*`, `sum`, and `sqrt`. While `*` refers to the element-wise multiplication of arrays, `sum` computes the sum of all elements of an argument array. Although in the example `sqrt` is applied to a scalar only, `sqrt` in general is applicable to arbitrarily shaped arrays as well.

Such a composite programming style has several advantages. Programs are more concise because error-prone explicit specifications of array traversals are

R. Peña and T. Arts (Eds.): IFL 2002, LNCS 2670, pp. 182–198, 2003.

hidden from that level of abstraction. The applicability of operations to arrays of any shape in conjunction with the multitude of layers of abstraction allows for code reuse in a way that is not possible in scalar languages. However, when it comes to applying such universal operations to parts of an array only, a more sophisticated notation is required [19].

This paper is concerned with the special but frequently occurring situation where an operation is to be performed on certain axes of arrays only. As an example, Fig. 1 illustrates the various possible applications of `L2Norm` to different axes of a 3-dimensional array.

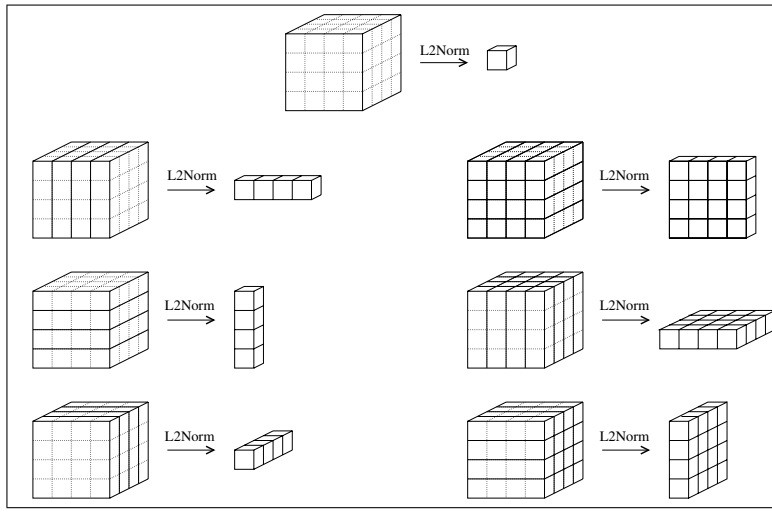

Fig. 1. Different views of an array.

In the standard case, as shown at the top of Fig. 1, `L2Norm` is applied to all elements and, hence, reduces the whole cube into a single scalar. However, the same cube may also be interpreted as a vector of matrices, as on the left hand side of Fig. 1. In this case, we would like to apply a reduction operation like `L2Norm` to each of the submatrices individually, yielding a vector of results. Similarly, the cube may also be regarded as a matrix of vectors. This view should result in applying `L2Norm` to individual subvectors yielding an entire matrix of results, as shown on the right hand side of Fig. 1. To add further complexity to the issue, the latter two views additionally offer the choice between three different orientations each.

In principle, such a mapping of an operation to parts of arrays in SAC can be specified by means of so-called WITH-loops, the central language construct for defining array operations in SAC. However, their expressiveness by far exceeds the functionality required in this particular situation because the design of WITH-loops aims at a much broader range of application scenarios. Rather cumbersome specifications may be the consequence when this generality is not needed, as for example in the cases shown in Fig. 1.

One approach to improve this situation without a language extension seems to be the creation of a large set of slightly different abstractions. However, continuously "re-inventing" minor variations of more general operations runs counter the idea of generic, high-level programming. Providing all potentially useful variations in a library is also not an option because this number explodes with an increasing number of axes and is unlimited in principle. Moreover, coverage of one operation still does not solve the problem for any other.

Another potential solution may be found in additional format parameters. Unfortunately, the drawbacks of this solution are manifold. Format arguments may have to be interpreted at runtime, which mostly prevents code optimizations. Many binary operations are preferably written in infix notation, which does not allow for an additional parameter. Last but not least, additional format parameters once again must be implemented for any operation concerned, although the problem itself is independent of individual operations.

What is needed instead is a more general mechanism that — independent of concrete operations — provides explicit control over the choice of axes of argument arrays to which an operation is actually applied. In this paper we propose such a mechanism, which fits well into the framework of generic, high-level array programming. It consists of a syntactical extension, called *axis control notation*, a compilation scheme, which transforms occurrences of the new notation into existing SaC code, and tailor-made code optimization facilities.

The remainder of the paper is organized as follows. Section 2 provides a very brief introduction into SaC for those who are not yet familiar with the language. In Section 3, we present the axis control notation. The compilation of axis control constructs into existing SaC code is outlined in Section 4, while optimization issues specific to the new mechanism are discussed in Sections 5 and 6. Finally, some related work is sketched out in Section 7, and Section 8 draws conclusions.

2 SAC

The core language of SaC is a functional subset of C, extended by n-dimensional arrays as first class objects. Despite the different semantics, a rule of thumb for SaC code is that everything that looks like C also behaves as in C. Arrays are represented by two vectors, a shape vector that specifies an array's extent wrt. each of its axes, and a data vector that contains all its elements. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays with a fixed number of dimensions, e.g. `int[.,.]`, and arrays with any number of dimensions, i.e. `int[*]`.

In contrast to other array languages, e.g. Fortran-95, Apl, or later versions of Sisal [7], SaC provides only a very small set of built-in operations on arrays. Basically, they are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's number of dimensions (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements of an array (*array*`[`*index-vector*`]`), where the length of *index-vector* is supposed to meet the number of dimensions or axes of *array*.

All basic aggregate array operations which are typically built-in in other array languages in SaC are specified in the language itself using powerful mapping and folding operations, the so-called WITH-loops. As a simple example take the definition of the element-wise `sqrt` function:

```
double[*] sqrt ( double[*] A)
{
  res = with (. <= iv <= .)
        genarray( shape( A), sqrt( A[iv]) );
  return( res);
}                                                    .
```

This function takes an array `A` of any shape as argument and computes a new array `res` by means of a simple WITH-loop. The WITH-loop consists of two parts, a so-called *generator* (preceded by the keyword `with`) and an *operation* (preceded by the keyword `genarray`). The basic functionality is defined in the operation part. In the given example, an array of the same shape as the array `A` is to be generated (first expression within the operation part), and an element at index position `iv` is computed by applying `sqrt`[1] to the corresponding element of `A` (second expression within the operation part). The generator part specifies an index set to which the given element computation actually applies. The dot symbols used within the generator part of the example are a shortcut notation for the lowest and for the highest legal index vector, respectively. Hence, the generator in fact covers the entire index range of `A`.

WithLoopExpr	⇒	**with** (*Generator*) ⌈ *AssignBlock* ⌋ *Operation*
Generator	⇒	*Expr RelOp IdVec RelOp Expr* ⌈ *Filter* ⌋
RelOp	⇒	< | <=
Operation	⇒	**genarray** (*Expr* , *Expr*) | ...

Fig. 2. Syntax of with-loop expressions.

As indicated by the (simplified) syntax of WITH-loops presented in Fig. 2, WITH-loops in general are more flexible. The generator set can be refined to rectangular index ranges specified by arbitrary lower and upper bounds, which in turn can be further restricted by optional filters. This inherently introduces the notion of a default definition for all those elements of the result array that are not covered by the generator. Furthermore, several variants of mapping and folding are available as operation parts, and an optional assignment block between the two parts allows more complex element definitions within the operation part to be abstracted out into local variables. However, in the context of this paper this flexibility is not required. A more detailed introduction into SaC can be found in [19]; a case study on a non-trivial problem investigating both the programming style and the resulting runtime performance is presented in [8].

[1] This seeming recursion is resolved by the type system of SaC; cf. [19].

3 Axis Control Notation

Having a closer look at the L2 norm example used to motivate the need for axis control, it turns out that the desired behaviour basically is a 3-step process.

1. Split the argument array along selected axes into uniform subarrays.
2. Apply the operation, e.g. `L2Norm`, to each subarray individually.
3. Laminate the array of subresults to form the overall result.

Fig. 3 illustrates this 3-step process for the L2 norm example and a 1-dimensional (top) as well as a 2-dimensional (bottom) splitting operation.

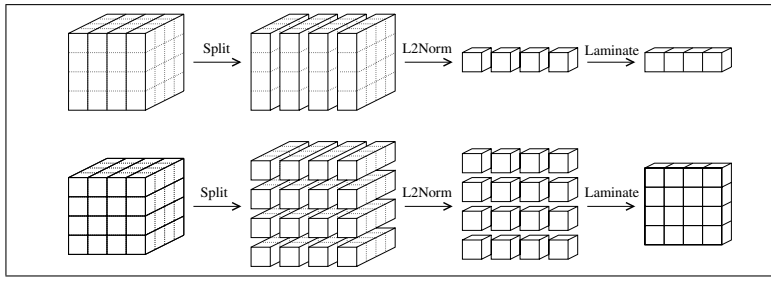

Fig. 3. Axis control as a 3-step process.

As a first step towards notational support for axis control we introduce a generalized selection facility. As outlined in the previous section, array element selection in SaC is specified as *array*`[`*index-vector*`]`, where the length of *index-vector* is supposed to meet the number of dimensions or axes of *array*. This selection facility is generalized by allowing index values in one or several dimensions to be left unspecified. Substituting elements of *index-vector* by single dots allows for selection of all elements of *array* along the corresponding axes. As illustrated in Fig. 4, the number of dimensions of the resulting value is identical to the number of dots in *index-vector*. Leaving all dimensions unspecific makes the selection facility an identity function. Of course, dots are only permitted in array selections, not in expressions in general. These syntactical extensions and their limitations are formally defined in Fig. 5.

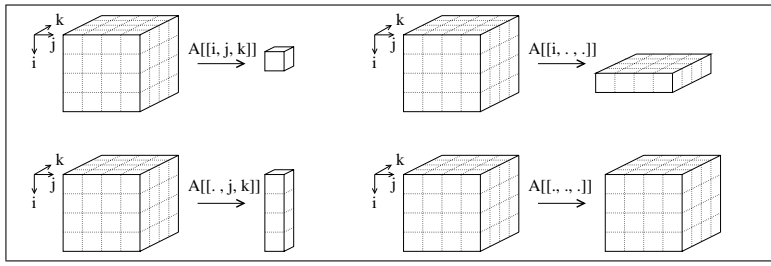

Fig. 4. Generalized array selection facility.

As a second step, we introduce a notation for lamination of subarrays, which includes the replicated application of operations prior to the lamination itself.

The new notation is based on expressions of the form { *idvec* `->` *expr*(*idvec*) }. Basically, such set expressions define a function from indices, represented by a so-called *frame vector* of identifiers (*idvec*), to values defined by the subsequent expression. In some sense, the set notation resembles a ZF-expression without a range specification or — in terms of SaC — a WITH-loop without a generator. This observation raises the question of how the range of indices is determined in the absence of an explicit specification. In fact, it is implicitly derived from the (mandatory) occurrence of each element of the frame vector in a selection operation within the subsequent expression. The shape of the array involved then defines the range for this particular index.

Associating the frame vector with these ranges yields a set of index vectors. For each element of this set the expression is evaluated and the resulting values are laminated according to the frame vector. Hence, the overall result and value of the entire set expression is characterized by a rank which equals the sum of the the frame vector's length and the rank of the expression.

Generalized selection facility and set notation form our *axis control notation* since in conjunction they offer a concise solution to the problem of axis control. For example, applications of `L2Norm` to submatrices of a cube can be written as simple as { `[j] -> L2Norm( A[[.,j,.]])` }; applying `L2Norm` to subvectors is as straightforward as { `[i,k] -> L2Norm( A[[i,.,k]])` }.

In both examples, the elements of the frame vector naturally occur in a selection operation within the expression. Hence, the range can easily be derived from the shape of the argument array `A`. The observation also illustrates why we use the term *notation* in this context. The axis control notation allows for shorter, more concise specifications in all those simple though frequent cases where a range *can* be derived from corresponding selections and, hence, the notational power of a full-fledged WITH-loop is not required.

As a reduction operation `L2Norm` reduces any subarray to a scalar. In general, any relationship between the shapes of argument and result subarrays may occur. The only restriction to the choice of operations here is uniformity, i.e., suitable operations must map all argument subarrays to result subarrays of identical shape. Otherwise, the subsequent lamination step would have to create an array with non-rectangular shape, which is not supported by SaC.

Examples which benefit from the new axis control notation are manifold, e.g., matrix transposition may be written as { `[i,k] -> Matrix[[k,i]]` }, matrix multiplication as { `[i,k] -> sum( MatrixM[[i,.]] * MatrixN[[.,k]])` }. Also purely structural array operations often benefit from axis control, e.g., the row-wise concatenation of a matrix with a vector can simply be written as { `[i] -> Matrix[[i,.]] ++ Vector` } based on the vector concatenation operator `++`.

If row-wise matrix-vector concatenation can be written easily, what about column-wise matrix-vector concatenation? Here, a limitation of axis control, as described so far, becomes apparent. In all cases examined so far, lamination used to be along the leftmost axis. However, column-wise matrix-vector concatenation requires lamination along the second axis. To cover this and similar cases we

further extend our axis control notation by the free choice of lamination axes. This is accomplished by allowing dot symbols to be used in the frame vector of the set notation. As a consequence, column-wise matrix-vector concatenation can be specified as { `[., j] -> Matrix[[., j]] ++ Vector` }.

$$
\begin{array}{lll}
Expr & \Rightarrow & \ldots \\
 & & \mid\ [\ Expr\ \lceil\ ,\ Expr\ \rfloor^{*}\ ] \\
 & & \mid\ Expr\ [\ Expr\ ] \\
 & & \mid\ Expr\ [\ SelVec\ ] \\
 & & \mid\ \{\ FrameVec\ ->\ Expr\ \} \\
 & & \\
SelVec & \Rightarrow & [\ DotOrExpr\ \lceil\ ,\ DotOrExpr\ \rfloor^{*}\ ] \\
DotOrExpr & \Rightarrow & .\ \mid Expr \\
 & & \\
FrameVec & \Rightarrow & [\ DotOrId\ \lceil\ ,\ DotOrId\ \rfloor^{*}\ ] \\
DotOrId & \Rightarrow & .\ \mid Id
\end{array}
$$

Fig. 5. Syntactical extensions for axis control notation.

Fig. 5 summarizes the various syntactical extensions introduced with the axis control notation and clarifies where dots may occur in program texts and where not. However, there are restrictions on the use of set notations which cannot elegantly be expressed by means of syntax. In all the examples presented so far, each identifier introduced by the frame vector occurs exactly once within the expression, which is directly in a selection operation. Consequently, ranges can be determined without ambiguity. In general, restrictions are less severe. First, the emphasis is on *direct* occurrence in a selection operation. For example, in the set expression { `[i, j] -> M[[i, N[[j]]]]` } ranges for `i` and `j` are clearly determined by the shapes of `M` and `N`, respectively. The indirect occurrence of `j` in the selection on `M` is not considered harmful. Similarly, additional occurrences outside of selection operations as in { `[i, j] -> M[[i, j]] + i * j` } are ignored. Multiple direct occurrences in selections on different arrays as in { `[i, j] -> M[[i, j]] + N[[j, i]]` } can easily be resolved by taking the minimum over all potential ranges. This ensures legality of selection indices with respect to the shapes of all arrays involved.

Only those set expressions in which elements of the frame vector do not directly occur in any selection operation have to be rejected. For example, in { `[i] -> M[[ fun( i)]]` } deriving a range specification from the shape of `M` would require to compute the inverse of `fun`, which usually is not feasible. Even simpler set expressions like { `[i] -> M[[ i - 1]]` } are ruled out because their meaning is not obvious: Legal values for `i` would be in the range from `1` up to and including `shape(M)[[0]]`. However, this contradicts to the rule that indexing in SaC always starts at zero.

These observations lead to the rule: A set notation that is constructed according to the syntax presented in Fig. 5 is considered legal, iff each identifier of the frame vector occurs at least once directly within an array selection.

4 Translating Axis Control Notation into WITH-loops

The reason why the two new language features — generalized selection and set notation — are referred to as "notations" stems from the observation that they are hardly more than syntactic sugar for particular forms of WITH-loops. In fact, their translation into WITH-loops can be implemented as part of a preprocessing phase mapping full SAC into core SAC.

4.1 Translating Generalized Selections

Generalized selections directly correspond to WITH-loops over ordinary (dot-less) selections. For example, the selection of the third column of a two-dimensional array `A`, specified as `A[[.,2]]`, can be implemented as

```
with (. <= [ tmp_0] <= .)
genarray( [ shape(A)[[0]] ], A[[tmp_0,2]])                    .
```

The shape of the result equals the extent of `A` along the first axis, i.e., the number of rows of `A`, and the elements are selected from all rows of `A` at column position `2` which refers to the third element each.

In general, an expression of the form *expr*`[`*iv*`]` can be translated into a WITH-loop that ranges over as many axes as dot symbols are found in *iv*. The shape of the resulting array is determined by the corresponding components of the shape of the expression *expr*. A formalization of this transformation is presented in Fig. 6. The transformation of an expression *expr* into an expression *expr*′ is

$$\mathcal{AC}[\![expr\texttt{[}iv\texttt{]}]\!] = \begin{cases} \texttt{with(. <= } ds \texttt{ <= .)} \\ \texttt{genarray(} shp\texttt{,} expr\texttt{[}idx\texttt{])} \end{cases}$$

where

$< ds\ ,\ shp\ ,\ idx > = DeCon\ iv\ 0$

$DeCon\ \texttt{[]}\ i$
$= < \texttt{[]}\ ,\ \texttt{[]}\ ,\ \texttt{[]} >$

$DeCon\ \texttt{[}\ .\ ,\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{]}\ i$
$= < \texttt{[tmp\_}i\texttt{]}{+\!+}\ ds\ ,\ \texttt{[shape(}expr\texttt{)[[}i\texttt{]]]}{+\!+}\ shp\ ,\ \texttt{[tmp\_}i\texttt{]}{+\!+}\ idx >$
where
$< ds\ ,\ shp\ ,\ idx > = DeCon\ \texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{]}\ i+1$

$DeCon\ \texttt{[}\ expr_0\texttt{,}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{]}\ i$
$= < ds\ ,\ shp\ ,\ \texttt{[}expr_0\texttt{]}{+\!+}\ idx >$
where
$< ds\ ,\ shp\ ,\ idx > = DeCon\ \texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{]}\ i+1$

Fig. 6. Compiling array decomposition into WITH-loops.

denoted by $\mathcal{AC}[\![expr]\!] = expr'$. It is assumed that this transformation is applied to all subexpressions where axis control notation is used without explicitly applying $\mathcal{AC}$ recursively to all potential subexpressions. SAC program code and meta variables that represent arbitrary SAC expressions are distinguished by means of different fonts: `teletype` is used for explicit SAC code, whereas *italics* refer to arbitrary expressions.

The transformation rule is based on the computation of the three vectors *ds*, *shp*, and *idx*, which determine the generator variable, the shape vector, and the modified index vector of the generated WITH-loop, respectively. All three vectors are computed from the index vector *iv* by means of a recursive function *DeCon*. It traverses the given index vector and looks for dot symbols. Whenever a dot symbol is encountered, new components are inserted into the generator variable and the shape expression. Furthermore, the dot symbol of the index expression is replaced with the freshly generated generator variable component. The additional parameter i is needed for keeping track of the position within the original index vector *iv* only.

4.2 Translating Set Notations

As shown in Section 3, the multiplication of two matrices `M` and `N` can be specified as `{ [i,j] -> sum( M[[i,.]] * N[[.,j]]) }`. Basically, this expression can be translated into a WITH-loop by turning the frame vector, i.e. `[i,j]`, into an index generator variable and by turning the right hand side expression into the body of a WITH-loop:

```
with (. <= [i,j] <= .)
genarray( [ shape(M)[[0]], shape(N)[[1]] ],
          sum( M[[i,.]] * N[[.,j]]) );                    .
```

As explained in the previous section, the difficulty involved here is the determination of the result shape, i.e., `[ shape(M)[[0]], shape(N)[[1]] ]`. It has to be derived from the direct occurrences of `i` and `j` within array selections on the right hand side of the set notation. Since `M[[i,.]]` selects the $\mathtt{i}^{th}$ row of `M`, its maximum range is determined by the extent of `M` in the leftmost axis, i.e., `shape(M)[[0]]`. Likewise, the selection of the $\mathtt{j}^{th}$ column of `N` limits the range of `j` by `shape(N)[[1]]`.

A formalization of this approach towards the compilation of the set notation into WITH-loops is presented in Fig. 7. Two functions *FindSels* and *CompExt* are used for computing the components s_j of the result shape from the right hand side expression *expr*. *FindSels* expects two arguments: an expression *expr*

$$\mathcal{AC}[\![\{\ [var_0,\ \ldots,\ var_n]\ \texttt{->}\ expr\ \}]\!] = \begin{cases} \texttt{with(}\ .\ \texttt{<=}\ [var_0,\ldots,var_n]\ \texttt{<=}\ .\texttt{)} \\ \texttt{genarray(}\ [\ s_0,\ \ldots,\ s_n\texttt{]},\ expr\texttt{)} \end{cases}$$

where

$\forall j \in \{0,\ldots,n\} : s_j = CompExt\ (\ FindSels\ var_j\ expr\ )$

$FindSels\ var\ \dashv \ldots expr'\texttt{[[}e_0\texttt{,}\ldots\texttt{,}e_{i-1}\texttt{,}\ var\texttt{,}\ e_{i+1}\texttt{,}\ldots\texttt{,}e_m\texttt{]]}\ldots \vdash$
$\quad = \ [\texttt{shape(}\ expr'\texttt{)[[}i\texttt{]]}]$
$\qquad +\!\!+\ (\ FindSels\ var\ \dashv \ldots expr'\texttt{[[}e_0\texttt{,}\ldots\texttt{,}e_{i-1}\texttt{,}\ \texttt{0}\texttt{,}\ e_{i+1}\texttt{,}\ldots\texttt{,}e_m\texttt{]]}\ldots \vdash\ )$

$FindSels\ var\ expr$
$\quad = \ []$

$CompExt\ [] \ = \ ERROR$

$CompExt\ [ext_0] \ = \ ext_0$

$CompExt\ [ext_0,\ldots,ext_k] \ = \ \texttt{min(}\ ext_0\texttt{,}\ CompExt\ [ext_1,\ldots,ext_k]\ \texttt{)}$

Fig. 7. Compiling array construction into WITH-loops.

and a variable name *var*. It locates subexpressions of *expr* that consist of array selections containing the given variable *var*. This is indicated by a pseudo pattern notation $\dashv \ldots expr'\,$`[`$expr''$`]`$\ldots \vdash$ which is meant to match arbitrary expressions that contain a subexpression of the form $expr'$`[`$expr''$`]`. For each array selection that contains the variable *var*, an according shape component selection is put into a resulting list of expressions. Note here, that for each component of the result shape such a list is computed. In the matrix multiplication example, all these lists do contain a single element only.

The function *CompExt* finally creates the expressions of the shape components from such lists. Empty lists indicate illegal programs as the corresponding variables are not used directly within array selections at all. If a list contains a single element only, this can be taken directly, as in the example. Multiple list entries require to guarantee that none of the corresponding selections violates array boundaries. To do so expressions are created that compute the minimum of all list components at runtime.

So far, it has been assumed that the frame vectors contain variables only. As a consequence, non-scalar right hand side expressions always constitute the rightmost axes of the result arrays. Now, the scheme has to be extended to cope with dot symbols in frame vectors. As these serve only one purpose, namely to place the right hand side expressions freely within the result, set expressions that contain dot symbols in their frame vectors can be transformed into nestings of two dot-free set expressions: one for computing the results and another one for accomplishing the intended transpose operation.

Applying this idea to the column-wise matrix-vector concatenation example, the original specification { `[.,i] -> M[[.,i]] ++ v` } first is transformed into { `[i] -> M[[.,i]] ++ v` } which inserts the prolongated column vectors as leftmost axis, i.e. as rows, of the result. Subsequently, the modified computation is embedded into a simple matrix transpose which leads to an expression of the form { `[tmp_0,i] -> { [i] -> M[[.,i]] ++ v } [[i,tmp_0]]` }.

The transformation of set notations that contain dot symbols in the frame vector into a nesting of dot-free ones can be formalized as shown in Fig. 8. As-

$\mathcal{AC}$`[{`$\ iv$ `->` $expr$ `}]` $=$ `{` lhs `->` `{` vs `->` $expr$ `}[` vs `++`ds `]` `}`
where
$<\ lhs\ ,\ vs\ ,\ ds\ >\ =\ Perm\ iv\ 0$

$Perm$ `[]` i
$= <$ `[]` , `[]` , `[]` $>$
$Perm$ `[` . , v_1, ..., v_n`]` i
$= <$ `[tmp_`i`]++` lhs , vs , `[tmp_`i`]++` ds $>$
where
$<\ lhs\ ,\ vs\ ,\ ds\ >\ =\ Perm$ `[`v_1, ..., v_n`]` $i+1$
$Perm$ `[` var, v_1, ..., v_n`]` i
$= <$ `[`var`]++` lhs , `[`var`]++` vs , ds $>$
where
$<\ lhs\ ,\ vs\ ,\ ds\ >\ =\ Perm$ `[`v_1, ..., v_n`]` $i+1$

Fig. 8. Resolving dot symbols on the left hand side of array constructions.

suming that the frame vector *iv* contains at least one dot symbol, a set notation { *iv* -> *expr* } first is turned into an expression { *vs* -> *expr* } where *vs* is obtained from *iv* by stripping off the dot symbol(s). This expression is embedded into a transpose operation { *lhs* -> { ...}[*vs* ++*ds*] }. The frame vector *lhs* in this set notation equals a version of *iv* whose dots have been replaced by temporary variables named `tmp_`*i* with *i* indicating the position of the temporary variable in *lhs*. The selection vector consists of a concatenation of the "dot stripped" version *vs* and a vector *ds* that contains a list of the temporary variables that have been inserted into the left hand side. This guarantees that all axes referred to by the dot symbols of *iv* are actually taken from the leftmost axes of { *vs* -> *expr* } and inserted correctly into the result.

5 Compilation Intricacies

As can be seen from applying the compilation scheme $\mathcal{AC}$ to the few examples on axis control notation given so far, intensive use of the new notation typically leads to deep nestings of WITH-loops. This contrasts strongly with the typical structure of SaC programs so far. The effect of this change in programming style can be observed when comparing the runtimes of direct specifications versus specifications that make use of axis control notation. A comparison of a direct specification of the row-wise matrix-vector concatenation with the axis control notation based solution on a Sun UltraSparc I for a 2000 x 2000 element matrix and a 500 element vector shows a slowdown by about 50%. For the column-wise matrix-vector concatenation (same extents) the slowdown even turns out to be a factor of 14! Since runtime performance is a key issue for SaC, this observation calls the entire approach in question. Performance figures, which have been found competitive even to low-level imperative languages [9,8,19], could only be achieved without using axis control notation.

A closer examination of the compilation process shows that the nestings of WITH-loops generated by the transformation are not particularly apt to the optimizations incorporated into the SaC compiler implementation `sac2c`[2] so far. The problems involved can be observed nicely with the column-wise matrix-vector concatenation example. Starting with the expression

```
{ [.,i] -> M[[.,i]] ++ v }
```

the transformation scheme $\mathcal{AC}$ first eliminates the dots of the frame vector:

```
{ [tmp_0,i] -> { [i] -> M[[.,i]] ++ v }[[i,tmp_0]] }          .
```

Then, both set notations are transformed into WITH-loops:

```
with (. <= [tmp_0,i] <= .) {
  inner = with ( . <= [i] <= .)
          genarray( [ shape(M)[[1]] ], M[[.,i]] ++ v);
} genarray( ..., inner[[i, tmp_0]])                           .
```

[2] See <http://www.sac-home.org/>.

Note here, that the temporary variable `inner` is introduced for presentation purposes only. It represents the value of the inner set notation.

Finally, yet another WITH-loop is substituted for the column selection. For clarity of code, we again introduce a temporary variable `col` that holds the selected column(s):

```
with (. <= [tmp_0,i] <= .) {
  inner = with ( . <= [i] <= .) {
            col = with ( . <= [tmp_0] <= .)
                  genarray( [ shape(M)[[0]] ], M[[tmp_0,i]]);
          } genarray( [ shape(M)[[1]] ], col ++ vect);
} genarray( ..., inner[[i, tmp_0]])                          .
```

During optimization the WITH-loop-invariant computation of `inner` is lifted out of the body of the outer WITH-loop and the WITH-loop-based implementation of the concatenation operation `++` is inlined, which leads to a code structure of the form:

```
inner = with (... [i] ...) {
          col = with (... [tmp_0] ...)        // col = M[[.,i]]
                genarray( ... M[[tmp_0, i]] ...);
          res = with (... [j] ...)            // res = col ++ v
                genarray( ... col[[j]] ...  v[[j]] ...);
        } genarray( ... , res);
  res = with (... [tmp_0,i] ...)              // transpose
        genarray( ... inner[[i,tmp_0]] ...);                 .
```

At this stage, WITH-LOOP-FOLDING [18], a SAC-specific optimization that allows consecutive WITH-loops to be folded into single ones, is applied. It condenses the column selection and the concatenation operation into a single WITH-loop:

```
inner = with (... [i] ...) {
          res = with (... [j] ...)            // res = M[[.,i]] ++ v
                genarray( ... M[[j, i]] ... v[[j]] ...);
        } genarray( ... , res);
  res = with (... [tmp_0,i] ...)              // transpose
        genarray( ... inner[[i,tmp_0]] ...);                  .
```

Unfortunately, the remaining WITH-loops cannot be folded any further, as `[i]` is a 1-dimensional generator, whereas `[tmp_0,i]` is a 2-dimensional one. This leads to the generation of C code which copies all array elements three times. First, the individual vectors that represent the prolongated columns are built by the inner WITH-loop. Then, these vectors are copied into the transpose of the result, as represented by `inner`. Finally, the last WITH-loop realizes the transpose required.

The major hindrance of further optimizations is the nesting of WITH-loops as it resulted from the expression `M[[.,i]] ++ v` within the set notation. If this nesting was converted into a single WITH-loop that operates on scalars, all copying could be avoided. Rewriting the nesting as a single WITH-loop, we obtain

```
inner = with (... [i,j] ...)
        genarray( ... M[[j, i]] ... v[[j]] ... );
  res = with (... [tmp_0,i] ...)
        genarray( ... inner[[i,tmp_0]] ...);                  ,
```

which can be folded into

```
res = with (... [tmp_0,i] ...)
      genarray( ... M[[tmp_0, i]] ... v[[tmp_0]] ...);
```

As the resulting WITH-loop is identical to a direct specification, the runtime overhead inflicted by the use of axis control notation is eliminated entirely.

6 Scalarization of WITH-loops

The observation that WITH-loops operating on scalars are compiled into more efficient code than nested WITH-loops gives raise to a new optimization technique, called WITH-LOOP-SCALARIZATION. It systematically transforms nested WITH-loops into non-nested ones. Fig. 9 presents the basic transformation scheme $\mathcal{SC}$. The pattern which has to be looked for is a WITH-loop whose body is entirely

$$\mathcal{SC}\left[\!\!\left[\begin{array}{l}\texttt{with (}lb_1\texttt{ <= }iv_1\texttt{ < }ub_1\texttt{) \{}\\ \quad v_1\texttt{ = with (}lb_2\texttt{ <= }iv_2\texttt{ < }ub_2\texttt{) \{}\\ \qquad v_2\texttt{ = }expr\texttt{(}\,iv_1,iv_2\texttt{);}\\ \quad\texttt{\} genarray(}\,shp_2\texttt{, }v_2\texttt{);}\\ \texttt{\} genarray(}\,shp_1\texttt{, }v_1\texttt{)}\end{array}\right]\!\!\right] = \left\{\begin{array}{l}\texttt{with (}lb_1\texttt{++}lb_2\texttt{ <= }iv\texttt{ < }ub_1\texttt{++}ub_2\texttt{) \{}\\ \quad iv_1\texttt{ = take(}\,shape\texttt{(}\,lb_1\texttt{), }iv\,\texttt{);}\\ \quad iv_2\texttt{ = drop(}\,shape\texttt{(}\,lb_1\texttt{), }iv\,\texttt{);}\\ \quad v\texttt{ = }expr\texttt{(}\,iv_1,iv_2\texttt{);}\\ \texttt{\} genarray(}\,shp_1\texttt{++}shp_2\texttt{, }v\texttt{);}\end{array}\right.$$

$$\text{if } iv_1 \notin FV(lb_2) \;\wedge\; iv_1 \notin FV(ub_2)$$

Fig. 9. Simple WITH-LOOP-SCALARIZATION scheme.

made up of another WITH-loop. The transformation itself turns out to be rather simple: the vectors for the shape of the result and the bounds of the index generator have to be concatenated. The body of the resulting WITH-loop basically is identical to the body of the inner WITH-loop. It only requires the two index vectors of the original WITH-loop nesting (iv_1 and iv_2 in Fig. 9) to be derived from the new index generator variable by splitting it up accordingly.

However, an application of the transformation is not appropriate for all kinds of WITH-loop nestings that match the given pattern. The problem involved here is the fact that the bounds of the inner WITH-loop, i.e. lb_2 and ub_2, are lifted out of the scope of iv_1. Therefore, the transformation can only be applied if neither lb_2 nor ub_2 depends on iv_1.

Code generated from applications of our axis control notation typically match the nesting pattern of Fig. 9. For example, both row-wise as well as column-wise matrix-vector concatenation, as discussed in Section 3, benefit tremendously from WITH-LOOP-SCALARIZATION. In both cases, WITH-LOOP-SCALARIZATION is the key to compiling specifications based on axis control notation into codes which are equivalent to direct implementations of the problems. As a consequence, the performance degradations caused by using the axis control notation reported in Section 5 — factors of 1.5 and 14 — are eliminated entirely.

Although the new axis control notation is a major source for nested WITH-loops, these or similar intermediate code representations may occur for many

reasons. Hence, WITH-LOOP-SCALARIZATION as an optimization technique is independent of axis control. However, hand-coded WITH-loop nestings often have slightly different forms. To enhance the applicability of this transformation, the

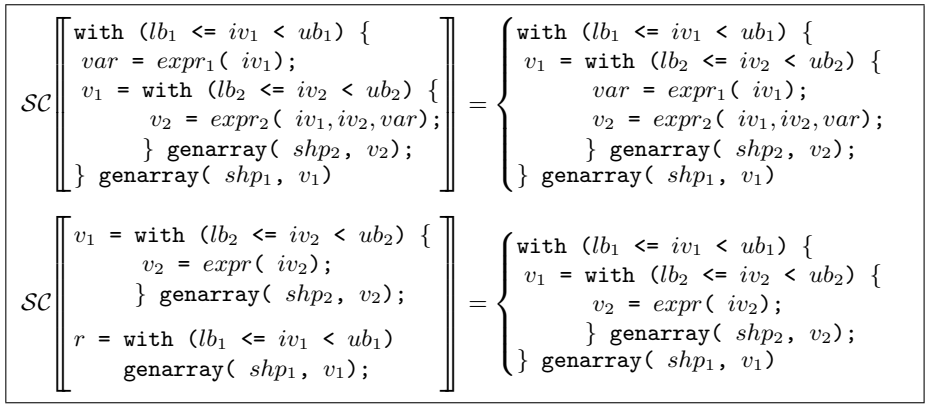

Fig. 10. Enhancing the applicability of WITH-LOOP-SCALARIZATION.

$\mathcal{SC}$ scheme is accompanied by additional rules for deriving the desired nesting pattern from others. Two transformations to this effect are shown in Fig. 10. The upper transformation rule moves assignments that precede the inner WITH-loop into its body. The lower part demonstrates how entire WITH-loops can be moved into others for generating WITH-loop nestings that can be scalarized.

In contrast to the basic scheme, which guarantees an improvement of the code generated, these two transformations may introduce considerable overhead as the computation of the expressions that are moved into the WITH-loop bodies is duplicated. Whether or not this overhead actually leads to any runtime degradation depends on the concrete code it is applied to. If the transformation does trigger further optimizations such as WITH-LOOP-FOLDING, the amount of overhead may be easily amortized by the effect of these optimizations. Otherwise, if the code remains almost unmodified, the back-end of the compiler may detect the loop-invariant portions of the code and lift them back out again during the final code generation phase.

7 Related Work

APL [11], the origin of all array languages, addresses the issue of axis control only in a very restricted way. Certain built-in operators provide an additional optional parameter which allows selection of exactly one axis. For example, the reduction operator `/` by default reduces the rightmost axis of an argument array `A` using an appropriate binary built-in operation α: α`/A`. Reduction along the second axis, provided that `A` is of suitable rank, can be written as α`/[2]A`. Although this language feature of APL is sometimes erroneously called *dimension operator*, it clearly lacks the desired generality as it is limited to certain built-in operators as well as to the selection of exactly one axis.

These shortcomings have been addressed in the further development of APL. IBM's APL-2, which largely influenced the current APL standard [12], introduced the notion of *nested arrays* [4]. Whereas arrays in APL originally were multidimensional data structures based on scalar elements, nested arrays impose additional structure. Entire arrays can be "wrapped" by means of the new *enclose* operator and behave just as scalars afterwards, i.e., they hide their internal structure. A complementary *disclose* operator allows for "unwrapping" previously enclosed arrays.

The array language NIAL [14,15] also uses the notion of nested arrays, but comes without an explicit *enclose*/*disclose* mechanism. Instead the nesting of arrays simply follows their construction. Full support for recursion allows for elegantly traversing multiple nesting levels of arrays. Since the effect of normal operations is limited to the outermost level, careful manipulation of nesting levels may achieve similar effects as our axis control notation. However, repeated re-organization of data structures for this purpose may be tedious and time-consuming both in terms of programmer time as well as in terms of execution time.

As an alternative to nested arrays, Sharp-APL [2] and later J [13] proposed the idea of *function rank* [5,10]. Rather than extending the data structure of arrays, they introduced the *rank operator* (or *rank conjunction* in J terminology). Basically, the rank conjunction is a built-in higher-order function, denoted by the infix operator "`"`", which provides a uniform and general concept for directing effects of any operation to a given number of either leading or trailing dimensions. For example, `L2Norm"2 A` would apply `L2Norm` to each 2-dimensional subarray of `A` individually and laminate the results. Provided that `L2Norm` is defined as its SAC counterpart, this operation would be equivalent to the SAC axis control expression `{[i] -> L2Norm( A[[i,.,.]])}`. Compared with our approach the rank conjunction is limited in two aspects. First, it only allows to address consecutive leading and trailing axes of argument arrays. Any other choice of axes requires explicit transposition of arguments beforehand. Second, it does not allow for permutation of axes as axes are not identified by names.

So far, we have only sketched out work related to axis control. WITH-LOOP-SCALARIZATION does not find its counterpart in conventional loop optimizations (For surveys see [3,1].) as the setting is rather different. Conventional loops correspond to a single axis of an array each, whereas the whole issue discussed in Section 5 arises because by means of WITH-loops SAC does provide an inherently multi-dimensional loop construct. Only this feature provides the opportunity to merge nested loops into a single construct, whereas conventional languages do not offer means to express multi-dimensional loops other than by nesting.

An example of a multidimensional loop construct other than WITH-loops are the FOR-loops of SISAL [16]. However, according to [6] no optimizations similar to WITH-LOOP-SCALARIZATION are performed by the SISAL compiler. One reason may be the fact that SISAL 1.2 represents multidimensional arrays as nested vectors. Although this data representation has its flaws [17], it helps here because it avoids data copying of subarrays to a large extent.

8 Conclusions

This paper presents axis control notation as a general means for controlling the application of generic array operations in a dimension-specific manner. Axis control notation gives explicit control over the axes to which operations are applied as well as allowing the programmer to choose arbitrary dimensions for placing the results of such applications. The advantages of this approach are demonstrated in the context of the array programming language SaC. It is shown, that despite the enhanced flexibility – when compared to well-known concepts such as the rank conjunction in J – it can be implemented as a simple preprocessing step rather than requiring support for a built-in higher-order operator.

Unfortunately, these appealing properties do not come for free. Both notations, generalized selection and set notation, impose some syntactical restrictions which may be considered not very intuitive. The dot symbols used for generalized selection are put within the index vectors rather than being attached to the selection operator. Although this elegantly allows for indicating the axes to be selected, it may wrongly insinuate that dot symbols are legal vector entities. In a similar fashion, liberating the programmer from the burden to specify the index range of set notations leads to the restriction that the identifiers of frame vectors have to be used literally within array selections. However, in the context of axis control, these restrictions do not become apparent. Only if the axis control notation is "misused" for specifying more sophisticated functionalities, these restrictions may force the programmer to use WITH-loops instead.

As an offspring of the implementation of axis control notation, a new compiler optimization called WITH-LOOP-SCALARIZATION is proposed. It transforms nested WITH-loops into non-nested ones, which allows programs that make use of the new notation to be compiled into code that is identical to direct specifications that do without. This discloses another benefit of the proposed approach. Since the new notation is transformed into ordinary WITH-loops, WITH-LOOP-SCALARIZATION as an optimization technique is not specific to axis control notation, but it improves arbitrary SaC programs that contain nested WITH-loops.

Acknowledgements

We would like to thank Sébastien de Menten de Horne who inspired the development of the axis control notation by his ideas on active and passive indices. Furthermore, we are grateful to the people who helped improving this paper, in particular to Robert Bernecky for sharing his APL expertise, and to the four anonymous referees.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0.

2. I.P Sharp & Associates. *SHARP APL Release 19.0 Guide for APL Programmers*. I.P Sharp & Associates, Ltd., 1987.
3. D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
4. J.P. Benkard. Nested Arrays and Operators — Some Issues in Depth. In *Proceedings of the International Conference on Array Processing Languages (APL'92), St.Petersburg, Russia*, APL Quote Quad, pages 7–21. ACM Press, 1992.
5. R. Bernecky. An Introduction to Function Rank. In *Proceedings of the International Conference on Array Processing Languages (APL'88), Sydney, Australia*, volume 18 of *APL Quote Quad*, pages 39–43. ACM Press, 1988.
6. D.C. Cann. *The Optimizing SISAL Compiler: Version 12.0*. Lawrence Livermore National Laboratory, Livermore, California, 1993. part of the SISAL distribution.
7. J.T. Feo, P.J. Miller, S.K.Skedzielewski, S.M. Denton, and C.J. Solomon. Sisal 90. In A.P.W. Böhm and J.T. Feo, editors, *Proceedings of the Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA*, pages 35–47. Lawrence Livermore National Laboratory, Livermore, California, USA, 1995.
8. C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.
9. C. Grelck and S.-B. Scholz. HPF vs. SAC — A Case Study. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of the 6th European Conference on Parallel Processing (Euro-Par'00), Munich, Germany*, volume 1900 of *Lecture Notes in Computer Science*, pages 620–624. Springer-Verlag, Berlin, Germany, 2000.
10. R.K.W. Hui. Rank and Uniformity. In *Proceedings of the International Conference on Array Processing Languages (APL'95), San Antonio, Texas, USA*, APL Quote Quad, pages 83–90. ACM Press, 1995.
11. International Standards Organization. International Standard for Programming Language APL. ISO N8485, ISO, 1984.
12. International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.
13. K.E. Iverson. *Programming in J*. Iverson Software Inc., Toronto, Canada, 1991.
14. M.A. Jenkins and J.I. Glagow. A Logical Basis for Nested Array Data Structures. *Computer Languages Journal*, 14(1):35–51, 1989.
15. M.A. Jenkins and W.H. Jenkins. *The Q'Nial Language and Reference Manual*. Nial Systems Ltd., Ottawa, Canada, 1993.
16. J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al. Sisal: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146, Lawrence Livermore National Laboratory, Livermore, California, USA, 1985.
17. R.R. Oldehoeft. Implementing Arrays in SISAL 2.0. In *Proceedings of the 2nd SISAL Users Conference, San Diego, California, USA*, pages 209–222. Lawrence Livermore National Laboratory, 1992.
18. S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In *Proc. 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, selected papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.
19. S.-B. Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, 2003. Accepted for publication.