Functional Array Programming in SaC

Sven-Bodo Scholz

Dept of Computer Science, University of Hertfordshire, United Kingdom
S.Scholz@herts.ac.uk

Abstract. These notes present an introduction into array-based programming from a functional, i.e., side-effect-free perspective.

The first part focuses on promoting arrays as predominant, stateless data structure. This leads to a programming style that favors compositions of generic array operations that manipulate entire arrays over specifications that are made in an element-wise fashion. An algebraicly consistent set of such operations is defined and several examples are given demonstrating the expressiveness of the proposed set of operations.

The second part shows how such a set of array operations can be defined within the first-order functional array language SaC. It does not only discuss the language design issues involved but it also tackles implementation issues that are crucial for achieving acceptable runtimes from such genericly specified array operations.

1 Introduction

Traditionally, binary lists and algebraic data types are the predominant data structures in functional programming languages. They fit nicely into the framework of recursive program specifications, lazy evaluation and demand driven garbage collection. Support for arrays in most languages is confined to a very resricted set of basic functionality similar to that found in imperative languages. Even if some languages do support more advanced specificational means such as array comprehensions, these usualy do not provide the same genericity as can be found in array programming languages such as Apl, J, or Nial.

Besides these specificational issues, typically, there is also a performance issue. Lazy evaluation and garbage collection pose a major challenge on an efficient implementation for arrays. A large body of work went into dealing with the performance issue [vG97, PW93, CK01]. Most of these approaches rely on explicit single-threading, either by using monads or by using uniqueness typing, as this allows all array modifications to be implemented destructively. The drawback of this approach is that it requires programmers to be aware of the states of the arrays they are dealing with. Arrays need to be allocated and copied whenever more than one versin of an array is needed. Although this copying to some extend can be hidden behind libraries, it does come for the price of potentially superfluous copying at the library interfaces [CK03].

In this paper, we present a radically different approach. We introduce a new functional programming language called SaC which is designed around the idea of runtime efficient support for high-level programming based on arrays

Z. Horváth (Ed.): CEFP 2005, LNCS 4164, pp. 62–99, 2006.

as predominant data structure. This facilitates an array-oriented programming style as it can be found in dedicated array programming languages such as Apl [Int84], J [Bur96], or Nial [JJ93]. In contrast to these languages, the absense of side-effects in SaC provides the ground for several advanced optimization techniques [Sch03] leading to runtimes competitive with those obtained from hand-optimized imperative programs [GS00, SSH+06].

The paper consists of two major parts. In the first part, an introduction to array-oriented programming is given. Starting from array essentials such as array representation, inspection, and creation in Section 2, Section 3 provides the core array operators. The set of high-level operators defined in this part are prototypical for the basic operations found in any other array programming language. Nevertheless, they reflect part of the functionality provided by the standard library of SaC. Section 4 describes a SaC-specific language feature called **Axis Control Notation** which, when used jointly with the basic operators, allows many array algorithms to be specified in a combinator style. Several examples to this effect round-off the first part of the paper.

The second part of this paper focuses on the language SaC itself. After a brief introduction to the basic design issues of SaC in Section 5, Section 6 gives a detailled desciption of the central language construct of SaC called with-**loop**. Several examples and exercises are given to demonstrate how the generic aray operations defined in the foirst part can be implemented in SaC itself. How such program specifications can be compiled into efficiently executable code is sketched in Section 7.

2 Array Basics

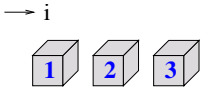
All arrays are represented by two vectors: a **data vector** which contains its elements, and a **shape vector** which defines its structure. Fig. 1 shows a few example arrays. As can be seen from the examples, the length of the shape vector corresponds to the dimensionality (also referred to as **rank**) of the array and the individual entities of it define the extent of the array with respect to the individual axes. The data vector enumerates all elements with increasing indices. For arrays with more than one axis, index increment proceeds with indices from right to left. From this relation between data and shape vector we obtain:

Lemma 1. *Let* $[d_0, \ldots, d_{q-1}]$ *denote the data vector of an array and let* $[s_0, \ldots, s_{n-1}]$ *denote its shape vector. Then we have* $q = \prod_{i=0}^{n-1} s_i$.

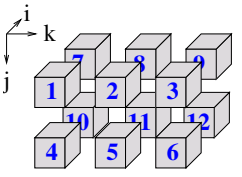
The bottom of Fig. 1 shows that scalar values can be considered 0-dimensional arrays with empty shape vector. Note here, that Lemma 1 for scalars still holds.

2.1 Specifying Arrays

Given the representation of n-dimensional arrays by two vectors, we use the following notation for arrays:



shape vector: [3]
data vector: [1, 2, 3]



shape vector: [2, 2, 3]
data vector: [1, 2, 3, ..., 11, 12]

shape vector: [ ]
data vector: [42]

Fig. 1. Array representations

$$\texttt{reshape(}\ [s_0, ..., s_{n-1}]\texttt{,}\ [d_0, ..., d_{q-1}]\texttt{)}$$

where $q = \prod_{i=0}^{n-1} s_i$. The special cases of scalars and vectors may be denoted as

$$s \equiv \texttt{reshape([], [}s\texttt{])} \quad \text{, and}$$
$$[v_0, ..., v_{n-1}] \equiv \texttt{reshape([}n\texttt{], [}v_0, ..., v_{n-1}\texttt{])} \quad .$$

This shortcut notation gives rise to considering `reshape` a built-in array operator which holds the following property:

$$\begin{aligned}&\texttt{reshape(shp\_vec, reshape(shp\_vec\_2, data\_vec))}\\ &\quad\texttt{== reshape(shp\_vec, data\_vec)}\end{aligned} \tag{1}$$

provided that Lemma 1 holds for the resulting array.

2.2 Inspecting Arrays

Alongside the array constructing operator `reshape`, functions for extracting shape and data information are required. We introduce two operations for retrieving shape information:

`shape` returns an array's shape vector, and
`dim` returns an array's dimensionality

For example, we have:

```
shape( 42) == []
shape( [1, 2, 3]) == [3]
shape( reshape( [2, 3], [1, 2, 3, 4, 5, 6])) == [2, 3]
```

Formally, `shape` is defined by

$$\texttt{shape(reshape(shp\_vec, data\_vec))} == \texttt{shp\_vec} \tag{2}$$

and `dim` is defined by

$$\texttt{dim}(\texttt{reshape}(\texttt{shp\_vec}, \texttt{data\_vec})) == \texttt{shape}(\texttt{shp\_vec})[0] \tag{3}$$

where the square brackets denote element selection. Note here that the element selection is well defined as `shp_vec` denotes an n-element vector and, thus, `shape( shp_vec)` is a one element vector.

From these definitions, we can derive

$$\forall \texttt{a} : \texttt{dim}(\texttt{a}) == \texttt{shape}(\texttt{shape}(\texttt{a}))[0] \tag{4}$$

as

dim(reshape(shp_vec, data_vec))
$\overset{(3)}{==}$shape(shp_vec)[0]
$\overset{(2)}{==}$shape(shape(reshape(shp_vec, data_vec)))[0] □

So far, we have used square brackets to denote selection within vectors. However, we want to introduce a more versatile definition for array selections. It is supposed to work for n-dimensional arrays in general. As one index per axis is required, such a definition requires an n-element vector as index argument rather than n separate scalar index arguments. Hence, we define an operation

`sel( idx_vect, array)`

which selects that element of `array` that is located at the index position `index_vect`. For example:

```
sel( [1], [1, 2, 3]) == 2
sel( [1, 0], reshape( [2, 3], [1, 2, 3, 4, 5, 6])) == 4
```

As we can see from the examples, we always have `shape( idx_vect)[0] == dim(array)`. This leads to the formal definition

```
shape( sel( idx_vec, reshape( shp_vec, data_vec))) == 0
```
provided that `shape( idx_vec) == shape( shp_vec)` (5)

From it, we obtain for scalars `s`:

`sel( [], s) == s`

In order to extend this property to non-scalar arrays (5) is generalized into

```
shape( sel( idx_vec, reshape( shp_vec, data_vec)))
       == shape( shp_vec) - shape( idx_vec)
```
provided that `shape( idx_vec) <= shape( shp_vec)` (6)

This extension enables the selection of entire subarrays whenever the index vector is shorter than the rank of the array to be selected from. For example, we have

```
sel( [1, 0], reshape( [2, 3], [1, 2, 3, 4, 5, 6]))
    == 4

sel( [1], reshape( [2, 3], [1, 2, 3, 4, 5, 6]))
    == [4, 5, 6]

sel( [], reshape( [2, 3], [1, 2, 3, 4, 5, 6]))
    == reshape( [2, 3], [1, 2, 3, 4, 5, 6])
```
.

2.3 Modifying Arrays

Similar to the generalized form of array element selection, we introduce an array modification operation which is not restricted to modifications of individual elements but permits modifications of entire subarrays. It takes the general form

```
modarray( array, idx_vect, val)
```

and results in an array which is identical to `array` but whose subarray at the index position `idx_vect` is changed into `val`. Hence, we have

$$\texttt{shape(modarray(reshape(shp\_vec, data\_vec), idx, val))} \texttt{ == shp\_vec} \tag{7}$$

and

$$\texttt{modarray(array, idx, val)[idx2]} == \begin{cases} \texttt{val} & \text{iff} \texttt{idx2} == \texttt{idx} \\ \texttt{sel(idx2, array)} & \text{otherwise} \end{cases} \tag{8}$$

It should be noted here that this "modification" conceptually requires the creation of a copy of the array. This is a consequence of the functional setting which requires the arguments of a function application to be unaffected by the evaluation of the application. However, static analysis often can determine that an array provided as argument is not referred to anywhere else. In these situations, `modarray` can be implemented in a destructive manner.

2.4 Generating Arrays

While an explicit specification of arrays is convenient for arrays with few elements only, large arrays require operator support. To this effect, we introduce an operation

```
genarray( shape, val)
```

which creates an array of shape `shape` with elements `val`. Similar to the operations introduced in the previous sections, `val` is not restricted to scalar values but can be an arbitrary array. For example, we have

```
genarray( [2], 42)
    == reshape( [2], [42, 42])

genarray( [2], [1, 2, 3])
    == reshape( [2, 3], [1, 2, 3, 1, 2, 3])

genarray( [2, 3], 1)
    == reshape( [2, 3], [1, 1, 1, 1, 1, 1])        .
```

Formally, we obtain for `genarray`

$$\texttt{shape(genarray(shp\_vec, reshape(shp\_vec2, data\_vec)))} \texttt{ == shp\_vec ++ shp\_vec2} \tag{9}$$

where `++` denotes vector concatenation and

$$\texttt{genarray(shp\_vec, a)[idx]} == \texttt{a} \quad \text{provided that} \quad \texttt{shape(shp\_vec)} == \texttt{shape(idx)} \tag{10}$$

2.5 Exercises

Exercise 1. Given the language constructs introduced so far, can you define the following array of shape `[5,2,2]`

```
[ [[ 0, 0 ], [ 0, 0 ]],
  [[ 1, 0 ], [ 0, 0 ]],
  [[ 0, 1 ], [ 0, 0 ]],
  [[ 0, 0 ], [ 1, 0 ]],
  [[ 0, 0 ], [ 0, 1 ]] ]
```

in a way so that the letter `1` is not used more than once?

Exercise 2. What are the results of the following expressions, if we assume `a` to be defined as `[1,2,3,4]`, and `b` to be defined as `[a,a]`?

- `modarray( modarray( a, [0], 0), [1], 0)`
- `modarray( b, [0], [0,0,0])`
- `modarray( b, [0], modarray( a, [0], 0))`

3 Array Operations

The operations introduced in the previous section pertain to very basic functionality only: array creation, inspection, and element/subarray replacement.

In this section, we introduce operations that compute arrays from other arrays in a more elaborate manner. The design of these operations is inspired by those available in array languages such as Apl, j, or Nial. However, several aspects - in particular wrt. special case treatment - have been adjusted to allow for a more runtime favorable compilation. The operations can be divided into three different categories: structural operations, that predominantly change the shapes of the argument arrays or the placement of the individual elements within them; element-wise operations which are mappings of scalar operations such as `+`, `-`, $<$, etc. into the domain of arrays; and so-called reductions which by means of dyadic operations successively fold all elements of an array into a scalar value.

3.1 Structural Operations

Concatenation of vectors, denoted by `++`, has been used in Section 2. Here, we provide a more generic definition of concatenation. The basic idea is to consider n-dimensional arrays vectors of $n-1$-dimensional subarrays. This leads to the following definitions

```
          shape( reshape( shp_vec, data_vec)
           ++ reshape( shp_vec2, data_vec2)
== modarray( shp_vec, [0], shp_vec[[0]] + shp_vec2[[0]])
provided that shape( shp_vec) == shape( shp_vec2) otherwise
```
(11)

and

```
        reshape( shp_vec, data_vec)
     ++ reshape( shp_vec2, data_vec2)                    (12)
== reshape( shp_res, data_vec ++ data_vec2)
```

where `shp_res` is defined as specified in (11). This definition realizes concatenation wrt. the left-most axis. Thus, we have

```
reshape( [2,2], [1,2,3,4]) ++ reshape( [2,2], [5,6,7,8])
    == reshape( [4,2], [1,2,3,4,5,6,7,8])                .
```

Besides concatenation, we introduce two operations for cutting off parts of an array:

`take` that selects a prespecified portion of an array, and
`drop` that cuts off a prespecified portion of an array.

In their simplest form we have for example

```
take( 2, [1,2,3,4,5]) == [1,2]
drop( 2, [1,2,3,4,5]) == [3,4,5]     .
```

Formally, we define `take` by

```
shape( take( n, reshape( shp_vec, data_vec)))
       == modarray( shp_vec, [0], n)                     (13)
           provided that      n ≥ 0
```

and

```
take( n, reshape( shp_vec, data_vec))[idx]
   == reshape( shp_vec, data_vec)[idx]                   (14)
           provided that      n ≥ 0
```

The symmetry between `take` and `drop` can be captured by defining `drop` through

$$\forall \mathtt{a} : \mathtt{drop}(\mathtt{n}, \mathtt{a}) == \mathtt{take}(\mathtt{n} - \mathtt{shape}(\mathtt{a})[[0]], \mathtt{a}) \tag{15}$$

and by extending `take` for negative arguments:

```
shape( take( n, reshape( shp_vec, data_vec)))
      == modarray( shp_vec, [0], | n | )                 (16)
```

and

$$\begin{array}{l}\mathtt{take(\ n,\ reshape(\ shp\_vec,\ data\_vec))[idx]}\\ == \begin{cases}\mathtt{reshape(\ shp\_vec,\ data\_vec)[idx]} & \text{iff } \mathtt{n} \geq 0\\ \mathtt{reshape(\ shp\_vec,\ data\_vec)[idx2]} & \text{otherwise}\end{cases}\\ \text{where } \mathtt{idx2 == modarray(\ idx,\ 0,\ idx[[0]] + offset\ )}\\ \qquad \text{where } \mathtt{offset == shp\_vec[[0]] - \ |n|}\end{array} \tag{17}$$

Applying the above definitions of `take`, we obtain

```
take( -2, [1,2,3,4,5]) == [4,5]
take( -1, reshape( [3,2], [1,2,3,4,5,6]))
    == reshape( [1,2], [5,6])                .
```

We also obtain

```
take( 0, [1,2,3,4,5]) == reshape( [0], [])
take( 0, reshape( [3,2], [1,2,3,4,5,6]))
    == reshape( [0,2], [])                    .
```

From these examples, we can observe that our array calculus requires us to distinguish between infinitely many differently shaped empty arrays. As numbers in square brackets are used as a shortcut notation for vectors, `[]` in fact denotes the array `reshape( [0], [])`.

To further extend the expressiveness of `take`, we use vectors instead of scalars as first argument and map the vector's components to the individual axes of the second argument. For example, we have

```
take( [2,1], reshape( [3,2], [1,2,3,4,5,6]))
    == reshape( [2,1], [1,3])

take( [2], reshape( [3,2], [1,2,3,4,5,6]))
    == reshape( [2,2], [1,2,3,4])

take( [], reshape( [3,2], [1,2,3,4,5,6]))
    == reshape( [3,2], [1,2,3,4,5,6])          .
```

A formal definition of this extended version of `take` is left as an exercise.

Two further structural operations are useful when it comes to operating on arrays in a cyclic fashion: `shift` and `rotate`. Both move all array elements towards increasing or decreasing indices. They only differ in the way they handle the border elements. While `shift` ignores the element(s) that is(are) moved out of the array and inserts a pre-specified one on the other end of the index range, `rotate` reuses the moved-out elements. An extension to n-dimensional arrays yields:

```
shift( [1], 0, [1,2,3]) == [0,1,2]

shift( [-1], 0, [1,2,3]) == [2,3,0]

shift( [1,1], 0, reshape( [3,3], [1,2,...,9]))
    == shift( [1], reshape( [3,3], [0,1,2,0,4,5,0,7,8]))
    == reshape( [3,3], [0,0,0,0,1,2,0,4,5])

rotate( [1], [1,2,3]) == [3,1,2]

rotate( [-1], [1,2,3]) == [2,3,1]

rotate( [1,1], reshape( [3,3], [1,2,...,9]))
    == rotate( [1], reshape( [3,3], [3,1,2,6,4,5,9,7,8]))
    == reshape( [3,3], [9,7,8,3,1,2,6,4,5])                .
```

Again, formal definitions are left as an exercise.

3.2 Element-Wise Operations

Most of the operations that in non-array languages are provided for scalars can be easily extended for usage on entire arrays by mapping them on an element-wise

basis. In the context of this lecture, we assume that all binary arithmetic, logic and relational operations of C can be applied to n-dimensional arrays. The only restriction we impose on these operations is shape conformity. More precisely, we demand that the shapes of the two argument arrays are either identical or at least one of them needs to be a scalar.
Here, a few examples:

```
[1,2,3] + [2,3,4] == [3,5,7]
reshape( [2,2], [1,2,3,4] ) * 2 == reshape( [2,2], [2,4,6,8] )
[1,2,3] < [0,4,5] == [false,true,true]                          .
```

3.3 Reductions

Reduction operations fold the scalar elements of an array into a single one by folding the data vector according to a binary operation. Again, most of the standard C operators can be used to that effect. In the context of this lecture, we focus on the following 4:

`sum` derives from `+`;
`prod` derives from `*`;
`all` derives from `&&`;
`any` derives from ||;

A few example applications of the above reduction operations:

```
sum( [1,2,3]) == 6
prod( reshape( [2,2], [1,2,3,4] )) = 24
all( [1,2,3] < [0,4,5]) == all( [false,true,true]) == false
any( [1,2,3] < [0,4,5]) == any( [false,true,true]) == true      .
```

3.4 Examples

The operations introduced so far suffice to conveniently express many operations on arrays without being required to write explicit loops over index ranges. This section provides a few examples as they typically occur when writing numerical codes.

Our first example relates to situations where all boundary elements of an array need to be cut off, i.e., we create an array that contains only those elements of a given argument array `a`, whose indices are all neither zero nor maximal. This can be achieved by a combination of the generic versions of `take` and `drop`:

```
take( shape( a)-2, drop( shape( a)*0+1 , a))
```

The most challenging aspect in this expression is the subexpression `shape (a)*0+1` which computes a vector of ones whose length matches the dimensionality of the array `a`.

In numerical methods, approximations are usually applied repetitively until a certain convergence criterion is met. Assuming `approx` to be an array that holds the current approximation and `solution` to be an identically shaped array of the solution to be approximated, the quality of the approximation can be computed by

```
sum( abs(approx - solution))
```

This example shows how the array notation helps in denoting expressions in a rather abstract style close to a mathematical notation.

Another example for the similarity to mathematical notation is the scalar product of two vectors `v1` and `v2` which can be specified as

```
sum( v1 * v2)
```

Even more complex expressions lend themselves to a specification in terms of generic array operations. As an example, consider the kernel of Danielson-Lanczos's FFT algorithm. Essentially, it computes a vector `v`'s fast furier transform by recursively applying FFT to subvectors which consist of those elements of `v` that are located at even or odd index positions, respectively. The results of the recursive calls are then combined into a single vector which constitutes the overall result. Assuming the availability of a function compress which allows us to extract every second element from a vector, this kernel can be specified as

```
fft( compress( [2], v)) ++ fft( compress( [2], drop( [1], v)))
```

3.5 Exercises

Exercise 3. What results do you expect from the following expressions:

- `reshape([3,0,5], [])[[]]`?
- `reshape([3,0,5], [])[[1]]`?
- `reshape([3,0,5], [])[[1,0]]`?
- `reshape([3,0,5], []) + reshape([3,0,5], [])`?
- `reshape([1,1], [1]) + reshape([1], [1])`?

Exercise 4. Give a formal definition of the extended version of `take`. Derive the dual definition for `drop` from your definition for `take` and equation (15).

Exercise 5. Give a formal definition of the structural operations `shift` and `rotate`.

Exercise 6. Reformulate the following expressions in terms of `take`, `++`, and the basic operations defined in the previous section. Try to sketch a correctness proof of your solution by using the formal definitions of the individual operations.

- `drop( v, a)`?
- `shift( [n], e, a)`?
- `shift( [m,n], e, a)`?
- `rotate( [n], a)`?
- `rotate( [m,n], a)`?

Can we define the general versions of `shift` and `rotate` as well?

Exercise 7. All operations introduced in this part apply to **all** elements of the array they are applied to. Given the array operations introduced so far, can you specify row-wise or column-wise summations for matrices? Try to specify these operations for a 2 by 3 matrix first.

3.6 Further Reading

There are several possible approaches for defining a consistent representation for n-dimensional arrays and the basic operations on them. The representation presented here, to a large extent, is based on that of APL as suggested in [Ive62]. Several variants have been proposed such as the Mathematics of Arrays [Mul88], APL2 [Bro85], or SHARP APL [Ass87]. An alternative approach is the Array Theory of T. Moore [JF99]. It is based on the idea of nested arrays and serves as basis for NIAL [JG89]. In [MJ91], this approach is contrasted to the Mathematics of Arrays. A more general discussion of the design space can be found in [JF99].

4 Axis Control Notation

As can be seen from the Exercise 7, without further language support, it is rather difficult to apply an array operation to certain axes of an array only. This section introduces two language constructs of SaC which, when taken together, can be used to that effect. While **Generalized Selections** are convenient for separating individual axes of an array, **Set Notations** allow to recombine such axes into a result array after applying arbitrary operations to them. However, as the two constructs in principle are orthogonal, we introduce them separately before showing how they can be combined into an instrument for **Axis Control**.

4.1 Generalized Selections

The selection operation introduced in Section 2.2 does not only allow scalar elements but entire subarrays of an array to be selected. However, the selection of (non-scalar) subarrays always assumes the given indices to refer to the leftmost axes, i.e., all elements wrt. the rightmost axes are actually selected. So far, a selection of arbitrary axes is not possible. As an example consider the selection of rows and columns of a matrix. While the former can be done easily, the latter requires the array to be transposed first.

To avoid clumsy notations, we introduce special syntactical support for selecting arbitrary subarrays called **Generalized Selections**. The basic idea is to indicate the axes whose elements are to be selected entirely by using dot-symbols instead of numerical values within the index vectors of a selection operation.

Note here, that vectors containing dot-symbols are not first class citizens of the language, i.e., they can exclusively be specified within selection operations directly!

There are two kinds of dot-symbols, single-dots which refer to a single axis and triple-dots which refer to as many axes as they are left unspecified within a selection. In order to avoid ambiguities, a maximum of one triple-dot symbol per selection expression is allowed.

Fig. 2 shows a few examples of generalized selections. The examples in the upper half demonstrate how arbitrary axes can be selected by using dot symbols in the selection vector. The lower examples feature the triple-dot notation. While in the left example the triple-dot refers to the two leftmost axes of the array `A`,

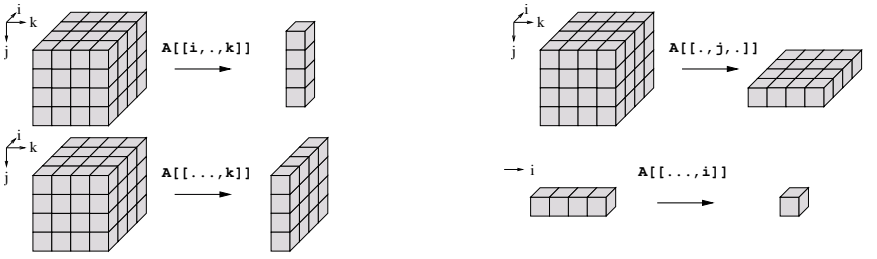


Fig. 2. Generalized Selections

the right example shows that in fact the triple-dot notation may refer to zero axes if the selection vector provides as many indices as the array to be selected from has axes.

4.2 Set Notation

The means for composing arrays that have been described so far are rather restricted. Apart from element-wise definitions all other operations treat all elements uniformly. As a consequence, it is difficult to define arrays whose elements differ depending on their position within the array. The so-called **set notation** facilitates such position dependent array definitions. Essentially, it consists of a mapping from index vectors to elements, taking the general form

{ idx_vect -> expr }

where `idx_vect` either is a variable or a vector of variables and `expr` is an expression that refers to the index vector or its components and defines the individual array elements. The range of indices this mapping operation is applied to usually can be determined by the expression given and, thus, it is not specified explicitly. Fig. 3 provides a few examples. The first example constitutes an element-wise

```
{ idx_vec -> a[idx_vec] + 1 } == a + 1
{ [i,j] -> mat[[j,i]] }
{ [i,j] -> ( i == j ? mat[[i,j]] : 0) }
```

Fig. 3. Set Notation examples

increment of a matrix `a`, the second example implements the transposition of a matrix `mat`, and the last example replaces all elements that are not located on the main diagonal of a matrix `mat` by the value 0. In all these examples, the ranges for `idx_vec`, `i`, or `j` are inferred from the selection operations on the right hand side, or, more precisely, from the shapes of the arrays the selections are applied to. It should be mentioned here, that the requirement to be able to infer the index ranges restricts the range of legal set notations. Examples for non-legal set notations are

```
{ idx_vec -> 1 }
{ idx_vec -> a[[ foo( idx_vec)]] }
```

where `foo` may be any user defined function. This restriction may seem rather prohibitive at first glance. However, in practice, it turns out that most examples suffice this restriction and that the readability of the examples benefits vastly from the implicit range inference.

As the following examples demonstrate, range inference is not limited to single occurences of the index variables.

```
{ idx_vec -> a[[ foo( idx_vec)]] + b[[ idx_vec]]}
{ [i,j] -> a[[i]] + a[[j]] }
{ [i,j] -> a[[i]] + a[[j]] + [1][[j]] }
```

In case of more than one use within a selection operation, the element-wise minimum of all selection shapes is used. Furthermore, the index variables may occur in non-selection related contexts as well. Wrt. the range inference, these occurences are ignored.

It should be noted here, that explicit indices on the left hand side do not necessarily have to match the rank of the array they select from. For example, we have

```
{ [i] -> reshape( [2,2], [1,2,3,4])[[i]] }
    == reshape( [2,2], [1,2,3,4])        .
```

From the definition (6) we obtain that `reshape( [2,2], [1,2,3,4])[[i]]` yields either `[1,2]` or `[3,4]` depending on `i` being `0` or `1`, respectively. The set notation combines these subarrays in a way that ensures that non-scalar right hand side expressions per default constitute the inner axes of the result array. This can be changed by using `.`-symbols for indicating those axes that should constitute the result axis. Applying this to our last example, we obtain

```
{ [.,i] -> reshape( [2,2], [1,2,3,4])[[i]] }
    == reshape( [2,2], [1,3,2,4])        .
```

4.3 Axis Control

Although generalized selections and the set notation per se can be useful their real potential shows when they are used in combination. Together, they constitute means to control the axes a given operation is applied to.

The basic idea is to use generalized selections to extract the axes of interest, apply the desired operation to the extracted subarrays and then recombine the results to the overall array.

Fig. 4 shows how axis control can be used to sum up different hyperplanes of a rank 3 array. The first example shows how the `sum` operation is mapped on all vectors within the rightmost axis of the rank 3 array which results in a matrix

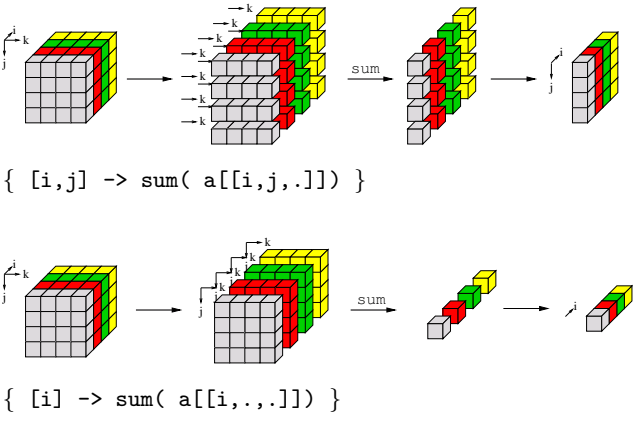


Fig. 4. Axis Control for summation of hyperplanes

of sums as indicated on the right hand side of the figure. The second example demonstrates how the summation can by applied to submatrices which results in a vector of sums.

Reduction operations, in general, are prone to axis control as they often need to be applied to one or several particular axes rather than an entire array. Other popular examples are the maximum (`max`) and minimum (`min`) operations which can now be used to compute local maxima or minima within selected hyperplanes.

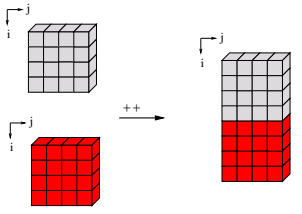
Further demand for axis control arises in the context of array operations that are dedicated to one fixed axis (usually the outermost one) and that need to be applied to another one. An example for this situation is the concatenation operation (`++`). Fig. 5 shows how axis control can be used to drive the concatenation into non-leftmost axes. Essentially, the idea is to first split the matrices into vectors. Pairs of these vectors are then concatenated before the results of that operation are combined into the final result.

4.4 Examples

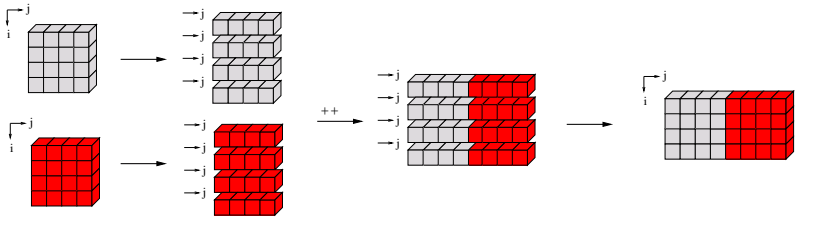
The array operations presented so far constitute a substantial subset of the functionality that is provided by array programming languages such as APL. When orchestrated properly, these suffice to express rather complex array operations very concisely. In the sequel, we present two examples that make use of this combined expressive power: matrix product and relaxation.

Matrix Product. The matrix product of two matrices A and B (denoted by $A\,B$) is defined as follows:

Provided A has as many columns as B has rows, the result of $A\,B$ has as many rows as A and as many columns as B. Each element $(A\,B)_{i,j}$ is defined as the scalar product of the i-th row of A and the j-th column of B, i.e., we have $(AB)_{i,j} = \sum_k A_{i,k} * B_{k,j}$.

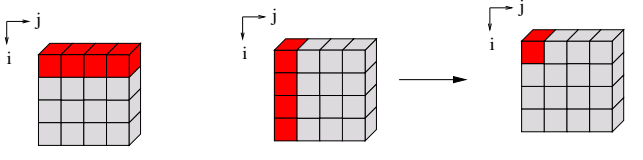



```
a ++ b
```




```
{ [i] -> a[[i]] ++ b[[i]] }
```

Fig. 5. Axis Control for concatenation on inner axes

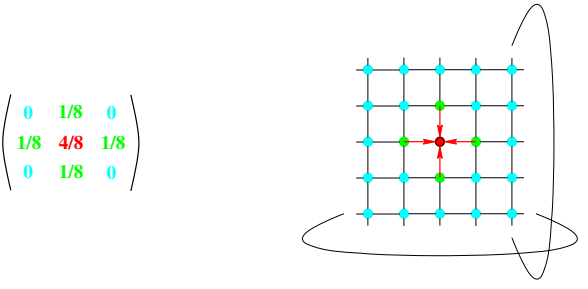



```
{ [i,j] -> sum( A[[i,.]] * B[[.,j]]) }
```

Fig. 6. Matrix product

Fig 6 shows how the matrix product can be defined in terms of axis control. The expression `A[[i,.]]` selects the `i`-th row of `A`, and `B[[.,j]]` refers to the `j`-th column of `B`. The index ranges for `i` and `j` are deduced from the accesses into `A` and `B`, respectively. A variable k as used in the mathematical specification is not required as we can make use of the array operations `*` and `sum`.

Relaxation. Numerical approximations to the solution of partial differential equations are often made by applying so-called **relaxation methods**. These require large arrays to be iteratively modified by so-called **stencil operations** until a certain convergence criterion is met. Fig. 7 illustrates such a stencil operation. A stencil operation re-computes all elements of an array by computing a weighted sum of all neighbor elements. The weights that are used solely depend on the positions relative to the element to be computed rather than the position in the result array. Therefore, we can conveniently specify these weights by a single matrix of weights as shown on the left side in the top of Fig. 7.




```
weights = [ [0d, 1d, 0d], [1d, 4d, 1d], [ 0d, 1d, 0d]] / 8d;
mat = ...
res = { [i,j] -> sum(
        { iv -> weights[iv] * rotate( iv-1, mat)}
                      [[...,i,j]] ) };
```

Fig. 7. A 5-point-stencil relaxation with cyclic boundaries

In this example, only 4 direct neighbor elements and the old value itself are taken into account for computing a new value. (Hence its name: **5-point-stencil operation**). As can be seen from the weights, a new value is computed from old ones by adding an eight-th each of the values of the upper, lower, left, and right neighbors to half of the old value.

As demonstrated on the right side in the top of Fig. 7 our example assumes so-called **cyclic boundary conditions**. This means that the missing neighbor elements at the boundaries of the matrix are taken from the opposite sides as indicated by the elliptic curves.

The code shown in the bottom of Fig. 7 shows the relevant part for computing a single relaxation step, i.e., the code for one re-computation of the entire array. At its core, all elements are re-computed by operations on the entire array rather than individual elements. This is achieved by applying `rotate` for each legal index position `iv` into the array of weights `weights`. Since the expression { `iv -> weights[iv] * rotate( iv-1, mat)`} computes a 3 by 3 array of matrices (!) the reduction operation `sum` needs to be directed towards the leftmost two axes of that expression only. This is achieved through axis control using a selection index `[...,i,j]` within a set notation over `i` and `j`.

4.5 Exercises

Exercise 8. How can a selection of all elements of a rank 3 array `mat` be specified using generalized selections? Try to find all 9 possible solutions!

Exercise 9. Referring to Exercise 3, can generalized selections be used for selecting "over" empty axis? For example, can you specify a selection vector $< vec >$, so that `reshape([3,0,5], [])[`$< vec >$`] == reshape( [3,0], [])` holds?

Exercise 10. Which of the examples in Fig. 3 can be expressed in terms of the array operations defined in the previous sections?

Exercise 11. What results do you expect from the expressions in Fig. 3 if `a` or `mat` turn out to be empty matrices, e.g., the turn out to be identical to `reshape( [10,0], [])`?

Exercise 12. The `.`-symbol in the set notation allows us to direct a computation to any axes of the result. This is identical to first putting the result into the innermost axes and then transposing the result. Can you come up with a general scheme that translates set notations containing `.`-symbols into set notations that do without?

Exercise 13. The operation `take` is defined in a way that ensures inner axes to be taken completely in case the take vector does not provide enough entities for all axes. How can `take` be applied to an array so that the outermost axis remains untouched and the selections are applied to inner axes, starting at the second one? (You may assume, that the take vector has fewer elements than the array axes!) Can you specify a term that - according to a take vector of length 1 - takes from the innermost axis only?

Exercise 14. Can you merge two vectors of identical length element-wise? Extend your solution in a way that permits merging n-dimensional arrays on the leftmost axis.

Exercise 15. Another variant of relaxation problems requires the boundary elements to have a fixed value. Can you modify the above solution in a way that causes all boundary elements to be 0? [**Hint:** You may consider the boundary elements to actualy be located **outside** the matrix]

4.6 Further Reading

A more detailled definition of Axis Control Notation can be found in [GS03]. APL [Int84] does provide the notion of explicit array nesting as alternative means for controlling the function applications wrt. certain axes. The notion of nesting is introduced in [Ben91, Ben92]. Yet another alternative to the same effect is the rank operator as proposed in [Ber88, Hui95]. It is implemented as part of J [Ive91, Ive95].

5 SAC Basics

SAC (for Single Assignment C) is a functional language whose design targets array-intensive applications as they for example can be found in the areas of scientific applications or image processing.

5.1 Overall Design

The fundamental idea in the design of the language is to keep the language as close as possible to C but to nevertheless base the language on the principle

of context free substitutions. While the former may attract application programmers with an imperative background, the latter ensures the Church-Rosser property which is crucial for extensive compile-time optimizations as well as for non-sequential executions. Another key objective in the design of SaC is to provide support for high-level array programming as introduced in the previous sections.

Fig. 8 sketches the overall design of SaC. As can be seen in the middle of the figure, a large part of standard C such as basic types and the fundamental language constructs is adopted in SaC. Only a few language constructs of C such as pointers and global variables need to be excluded in order to be able to guarantee a side-effect-free setting. Instead, some new language constructs are added pertaining to array programming. Besides a basic integration of n-dimensional arrays as first-class citizens, the most important addition are the so-called WITH-**loops**. They are versatile language constructs that allow all the array operations as introduced in the previous sections to be defined within the language itself.[1]

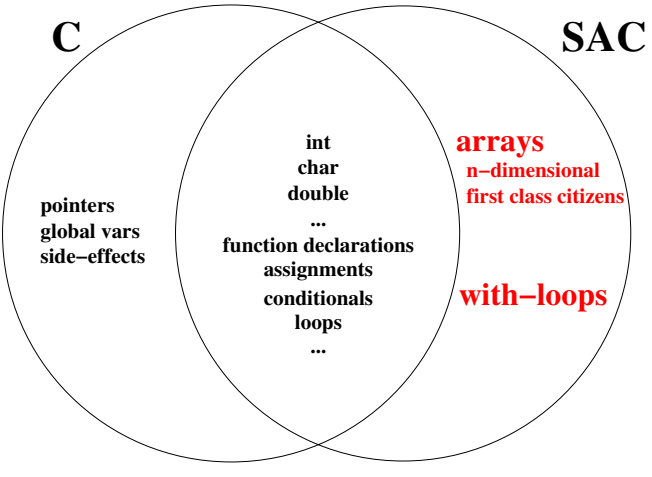


Fig. 8. The overall design of SaC

5.2 To Be and Not to Be Functional

The incorporation of most of the fundamental language constructs of C such as loops, conditionals, and assignments into the functional setting of SaC allows the programmer to stick to his preferred model of computation. To illustrate this effect, let us consider the following function `foo`:

[1] In fact, all examples from the previous sections can be used in SaC without modification. They are implemented within the standard library of SaC.

```
int foo( int v, int w) {
   int r;

   r = v + w;
   r = r + 1;
   return(r);
}
```

It takes two arguments `v` and `w`, adds them up, and increments the result by one which yields the return value of `foo`.

An imperative interpretation of `foo` is shown in Fig. 9. In the imperative

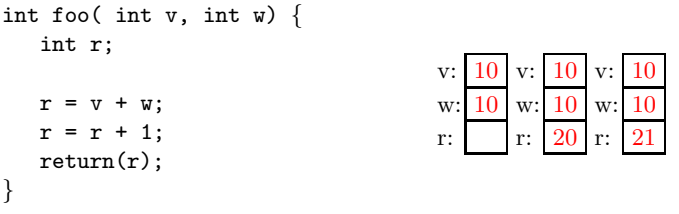


Fig. 9. An imperative look on `foo`

world, `v`, `w`, and `r` constitute names for box variables. During the execution of the body of `foo`, the content of these box variables is successively changed as indicated on the right hand side of Fig. 9 assuming an application to arguments `10` ans `10`. After the final "modification" of `r` the last value it contains, i.e., `21`, is returned as overall result of the function call of `foo`.

However, the definition of the function `foo` equally well can be interpreted as syntactic sugar for a `let`-based function definition as shown on the left-hand-side of Fig. 10. With this interpretation, `v`, `w`, and `r` become variables in a λ-calculus

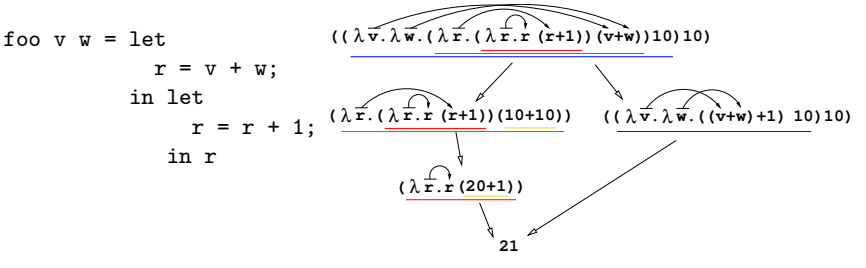


Fig. 10. A functional look on `foo`

sense. As we can see, the assignment to `r` has turned into two nested `let`-expression which effectively leads to two distinct variables `r` which are assigned to only once. A further transformation into an applied λ-calculus as shown on the

right-hand-side of Fig. 10 identifies the potential for independent evaluations of subexpressions. The arrows on top of the λ-expressions indicate the static scoping of the individual variables. The lines under the expressions indicate the β-redices that are present. As indicated by the different reduction sequences the λ-calculus representation thus eases the identification of legal program transformations part of which may be performed at compile-time.

This duality in program interpretation is achieved by the choice of a subset of C which can easily be mapped into an applied λ-calculus whose semantics reflects that of the corresponding C program. A formal definition of the semantics of SaC is beyond the scope of this lecture. In the sequel, it suffices to expect all language constructs adopted from C to adhere to their operational behaviour in C.

5.3 The Type System of SaC

As mentioned in Section 5.1, the elementary types of C are available in SaC too. However, they constitute only a small subset of the types of SaC. For each elementary type in C there exists an entire hierarchy of array types in SaC. As an example, Fig 11 shows the hierarchy for integer arrays. It consists of three layers

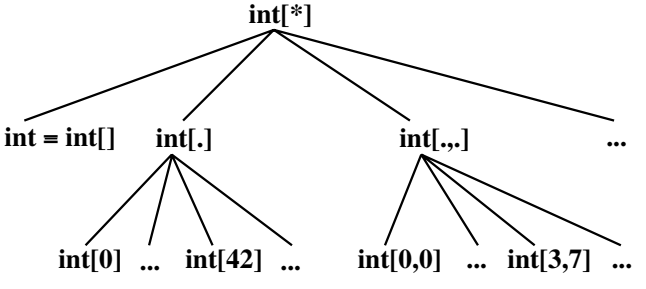


Fig. 11. A hierarchy of shapely types

of array types which differ wrt. the level of shape restrictions that is imposed on their constituents. On the top layer, we find `int[*]` which comprises all possible integer arrays. The second layer of types differentiates between arrays of different dimensionality. This layer comprises the standard type `int` which still denotes scalar integers only. All other types on this level are denoted by the elementary type followed by a vector of `.`-symbols. The number of `.`-symbols determines the rank of the arrays contained. On the bottom layer, the types are shape-specific. They are denoted by the elementary type followed by the shape. For example, `int[3,2]` denotes all integer matrices of shape [3,2].

Although a generic array programming style suggests a predominant use of the top layer types, the availability of the type hierarchy provides the programmer with additional expressiveness. Domain restrictions wrt. rank or shape of the arguments can be made explicit and support for function overloading eases rank/shape-specific implementations. Fig. 12 shows an example for such an overloading. Let us consider an implementation of a general solver for a set of linear

```
double[.] solve( double[.,.] A, double[.] b) {
  /* general solver */   ...
}

double[3] solve( double[3,3] A, double[3] b) {
  /* direct solver */   ...
}
```

Fig. 12. Overloading and function dispatch

equations `Ax = b` as indicated in the top of Fig. 12. For arrays of a certain shape it may be desirable to apply a different algorithm which has different runtime properties. The bottom of Fig. 12 shows how this functionality can be added in SaC by specifying a further instance of the function `solve` which is defined on a more restricted domain.

Besides the hierarchy of array types and function overloading it should be mentioned here that SaCin contrast to C does not require the programmer to declare array types for variables. Type specifications are only mandatory for argument and return types of all function instances.

5.4 Exercises

Exercise 16. Familiarize yourself with the SaC programming environment. Start your editor and type the following program:

Listing 1.1. Hello World

```
use StdIO: all;
use Array: all;

int main()
{
  printf( "Hello World!\n");
  return(0);
}
```

As you can see, it has a strong resemblance to C. The major difference are the module use declarations at the beginning of the program. Their exact behaviour is beyond the scope of this lecture. For now, it suffices to keep in mind, that these two declarations for most experiments will do the job. Details on the module system as well as further introductory material can be found at <http://www.sac-home.org/>.

5.5 Further Reading

A more extended introduction into the language constructs of SaC can be found in [Sch03]. Formal definitions are contained in [Sch96]. More elaboration on the type system of SaC is provided in [Sch01] and details of the module system can be found in [HS04].

6 WITH-loops

Besides basic support for n-dimensional arrays as described in Section 2 all array operations in SaC are defined in terms of a language construct called WITH-loop. There are three variants of WITH-loops: the genarray-WITH-loop, the modarray-WITH-loop, and the fold-WITH-loop.

6.1 Genarray-WITH-loop

In essence, the genarray-WITH-loop can be considered an advanced version of the set notation as introduced in Section 4. Apart from the syntax, there are two major differences: the result shape is specified explicitly, and besides the expression that is associated with the index vector there is a second expression, called **default expression**, which does not depend on the index vector. Which of these expressions actually is used for a given index position depends on a so called **generator** which denotes a subset of the index range. For all indices within the generator range the associated expression is chosen, and for all others the default expression is taken.

Fig. 13 shows an example WITH-loop and the array it represents. A genarray-WITH-loop starts with the keyword `with` followed by the index variable (`iv` in the example) followed by two parts: the so-called **generator part** (here: second line) and the **operator part** (here: last line). The operator part contains the result shape (here: `[4,5]`) and the default expression *def*. The generator part consists of a range specification followed by the expression *e(*`iv`*)* that is associated to it. As shown in the lower part of Fig. 13, the result array has shape `[4,5]`, all elements specified by the generator range i.e., those elements with indices between `[1,1]` and `[2,3]`, are computed from the associated expression with the index variable `iv` being replaced with the respective index, and all remaining elements are identical to the default expression.

```
A = with (iv)
      ([1,1] <= iv < [3,4]) : e(iv);
    genarray( [4,5],  def  );
```

$$\mathrm{A} = \begin{pmatrix} def & def & def & def & def \\ def & e([1,1]) & e([1,2]) & e([1,3]) & def \\ def & e([2,1]) & e([2,2]) & e([2,3]) & def \\ def & def & def & def & def \end{pmatrix}$$

Fig. 13. A genarray-WITH-loop

Range specifications always take the form

lower_bound rel_op variable rel_op upper_bound

where *lower_bound* and *upper_bound* are expressions that evaluate to vectors of the same length and *rel_op* is one of `<` and `<=`. This deviation from C-style was made in order to stress the fact that more general range restrictions are not supported. Although more general predicates do not cause any conceptual problems, they can have a substantial effect on the level of code optimization that can be achieved. In order to prevent from "spurious" performance degradations due to unfavourable generator specifications we rule out more general predicates here.

The generator variable can be replaced by a vector of variables which implicitly fixes the result shape. For example, we have:

```
with([i])
   ( [0] <= [i] < [n]) : i;
genarray( [n], 0);
```

which computes an `n`-element vector containing the values `0` up to `n-1`, i.e.,

```
reshape( [n], [0, 1, 2, ...., n-1])          .
```

Similarily, we can compute a 10 by 10 element unit matrix by

```
with([i,j])
   ( [0,0] <= [i,j] <= [9,9]) : ( i==j ? 1 : 0);
genarray( [10,10], 0);
```

Note here that an expression of the form

`(` *predicate* `?` *then_expr* `:` *else_expr* `)`

as in C denotes a conditional expression.

6.2 Modarray-WITH-loop

The difference between the modarray-WITH-loop and the genarray-WITH-loop lies in the way the result shape as well as the default expression are derived. Fig. 14 shows a prototypical example. As we can see, the only difference to a genarray-WITH-loop is the operator part. Rather than giving the result shape and the default expression explicitly, these are derived from an array *B*.

As a more concrete example, consider the following WITH-loop:

```
maxidx0 = shape( a)[[0]] - 1;
res = with([i])
        ( [0] <= [i] <= [maxidx0]): a[[maxidx0 - i]]
      modarray( a);
```

```
A = with (iv)
      ([1,1] <= iv < [3,4]) : e(iv)
    modarray( B );
```

$$\mathrm{A} = \begin{pmatrix} B[[0,0]] & B[[0,1]] & B[[0,2]] & B[[0,3]] & B[[0,4]] \\ B[[1,0]] & e([1,1]) & e([1,2]) & e([1,3]) & B[[1,4]] \\ B[[2,0]] & e([2,1]) & e([2,2]) & e([2,3]) & B[[2,4]] \\ B[[3,0]] & B[[3,1]] & B[[3,2]] & B[[3,3]] & B[[3,4]] \end{pmatrix}$$

Fig. 14. A modarray-WITH-loop

It computes an array `res` whose shape is identical to that of `a`. The generator specifies that all subarrays of the result wrt. the leftmost axis are reversed. Asuming `a` to be a 2 by 3 matrix of the form:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

we obtain `maxidx0 == 1` and, thus, `[a[[1]], a[[0]]] == [ [4, 5, 6], [1, 2, 3]]` as a result.

6.3 Fold-WITH-loop

The fold-WITH-loop, again, is a variant in the operator part. As can be seen in Fig. 15, it denotes a binary folding operation $\oplus$ alongside with the neutral element *neutr* of that operation. The overall result of such a fold-WITH-loop stems from folding all the expressions associated with the generator. However, the order in which the folding eventually is done is intentionally left unspecified. To guarantee predictable results, the operation $\oplus$ needs to be associative and commutative.

With this construct, all reduction operations can be conveniently specified. For example, the `sum` operation as defined in Section 3 can be specified as

```
res = with(iv)
        ( 0*shape(a) <= iv < shape(a)): a[iv]
      fold( +, 0);
```

Assuming `a` to be of the form

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

we obtain for the lower bound of the genrator `0*shape(a) == [0, 0]` and for the overall result:

```
A = with (iv)
      ([1,1] <= iv < [3,4]) : e(iv)
    fold( ⊕, neutr );
```

$$A = neutr \oplus e([1,1]) \oplus e([1,2]) \oplus e([1,3]) \oplus e([2,1]) \oplus e([2,2]) \oplus e([2,3])$$

($\oplus$ denotes associative, commutative binary function.)

Fig. 15. A fold-WITH-loop

`0 + a[[0,0]] + a[[0,1]] + a[[0,2]] + a[[1,0]] + a[[1,1]] + a[[1,2]]`
which reduces to `0 + 1 + 2 + 3 + 4 + 5 + 6 == 21`.

6.4 Extensions

So far, the generator specification is rather limited. Only one dense range of indices can be specified. To provide more specificational flexibility, SAC provides a few optional extensions for the generator parts of WITH-loops.

Fig. 16 shows the first extension the so-called **step vectors**. They allow to specify grids of indices rather than dense index ranges. Each of the components of the step vector specified the stride wrt. one individual axis. As shown in the

```
A = with (iv)
      ([2,1] <= iv < [8,11] step [2,3]) : e(iv);
    genarray( [10,13], def );
```

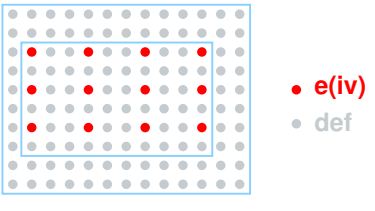


Fig. 16. Introducing step vectors

lower part of Fig. 16, in the example, this leads to a stride of 2 for the first axis and a stride 3 in the second.

These steps can be refined further by using so-called **width vectors**. Fig. 17 illustrates the use of width vectors. They allow the chosen grid elements not to

```
A = with (iv)
      ([2,1] <= iv < [8,11] ) step [3,4] width [2,3]) :  e(iv);
    genarray( [10,13],  def  );
```

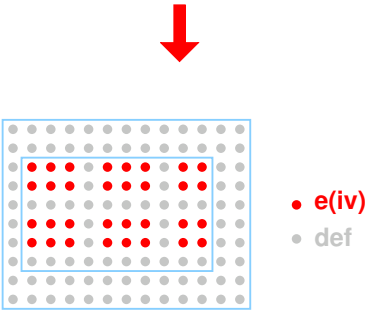


Fig. 17. Introducing width vectors

consist of one array element only but to consist of several adjacent elements. Again, the number of elements to be chosen can be specified on a per-axis basis.

Besides these potential extensions of individual generators, WITH-loops in SAC can contain more than just one generator part. In this case, the generators are subsequently executed in the order the are specified. Fig. 18 provides an example with two generators. The lower diagram shows how two separate generator ranges are computed according to the two (different) expressions *e1(*`iv`*)* and *e2(*`iv`*)*. All those elements not covered by any of the two generator ranges are copies of the default element.

6.5 Axis Control Revisited

As mentioned in Section 6.1, WITH-loops in SAC can be considered extended set notations. In fact, not only set notations but generalized selections as well can be defined in terms of WITH-loop.

For example, the generalized selection `a[[.,1]]` can be translated into

```
with( iv)
  ( 0 * shape( a)[[0]] <= iv < shape( a)[[0]])
  : a[ iv ++ [1]];
genarray( shape( a)[[0]], default);
```

which selects a vector of elements/subarrays of `a` as they are obtained when selecting with the second index being fixed to `1`. The challenge of this WITH-loop is a correct specification of the default expression `default`. It needs to be

```
A = with (iv)
      ([1,1] <= iv < [6,4]) : e1(iv)
      ([3,4] <= iv < [8,9]) : e2(iv)
    genarray( [10,13],  def  );
```

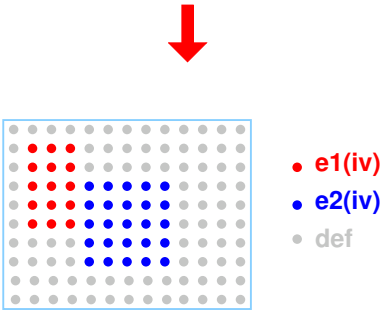


Fig. 18. A multi-generator WITH-loop

of the same shape as the subarrays of `a` are when selecting with two fixed indices. This can be specified by yet another WITH-loop as

```
with( iv)
  ( 0 * drop( [2], shape( a)) <= iv < drop( [2], shape( a))
  : zero( a);
genarray( drop( [2], shape( a)), zero( a));
```

where `zero( a)` denotes the scalar `0`.

Similarily, set notations can be translated into WITH-loops. Let us consider the increment operation `{ [iv] -> a[iv] + 1 }` . The range inference yields the shape of the result, which, in this example is identical to that of the array `a`:

```
with( iv)
  ( 0 * shape( a) <= iv < shape( a))
  : a[ iv ]
genarray( shape( a), default);
```

Again, the definition of the default expression, due to the lack of further information on the shape of `a`, requires another WITH-loop:

```
with( iv)
  ( 0 * drop( [dim(a)], shape( a)) <= iv
                          < drop( [dim(a)], shape( a))
  : zero( a);
genarray( drop( [dim(a)], shape( a)), zero( a));        .
```

6.6 Exercises

Exercise 17. Implement addition for arrays of arbitrary but identical shape as introduced in Section 3.2. Since this operation is already contained in the standard library you need to restrict the use of the modules from the standard library. You should import the scalar version of `+` by stating `import ScalarArith:{+}` and restrict the use of the library `Array` by excluding the array version for `+` contained in it. This can be achieved by a use statement of the form `use Array:all except{+}`. This will allow you to overload the scalar version of `+` by your own version of `+` for arrays.

Exercise 18. Implement a function `spread` that spreads an argument array `a` wrt. its first axis. It should insert elements / subarrays of value `0` between each two adjacent elements.

Modify your solution so that the "interim" values constitute the arithmetic mean of the formerly adjacent values.

Exercise 19. Extend your addition from Exercise 17 so that arrays of different shape but same dimensionality can be added. Find a consistent way in dealing with non-identical shapes!

Exercise 20. Extend the addition from Exercise 17 and Exercise 19 further so that mismatches in dimensionality can be handled as well. Do this by replicating the elements of the array that has fewer axes.

Exercise 21. Define an extended version of selection called `over_sel`. It should allow the selection vector to be an array of more than one axis. This index array should be considered an array of index vectors and the result should be an array of selected subarrays.

Examples:

$$\texttt{over\_sel}(\begin{pmatrix}1&0\\1&1\\1&1\end{pmatrix},\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}) == \begin{pmatrix}4\\5\\5\end{pmatrix}$$

$$\texttt{over\_sel}(\begin{pmatrix}1\\1\\0\end{pmatrix},\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}) == \begin{pmatrix}4&5&6\\4&5&6\\1&2&3\end{pmatrix}$$

6.7 Further Reading

Formal definitions of the WITH-loops as presented here can be found in [Gre01] and on the SaC home page <http://www.sac-home.org/>. A formal translation scheme for generalized selections and the set notation is contained in [GS03].

7 Compilation Issues

The natural choice of a target language for the compilation of SaC is C. Compilation to C can be liberated from all hardware-specific low-level optimizations

such as delay-slot utilization or register allocation, as this is taken care of by the C compiler for the target machine. Last not least, the strong syntactical similarity between the two languages allows the compilation efforts to be concentrated on adequate array representations and on optimizing the code for array operations. Other basic language constructs can be translated more or less one to one to their C counterparts.

The major phases of the actual SAC compiler are shown in Fig. 19. After scan-

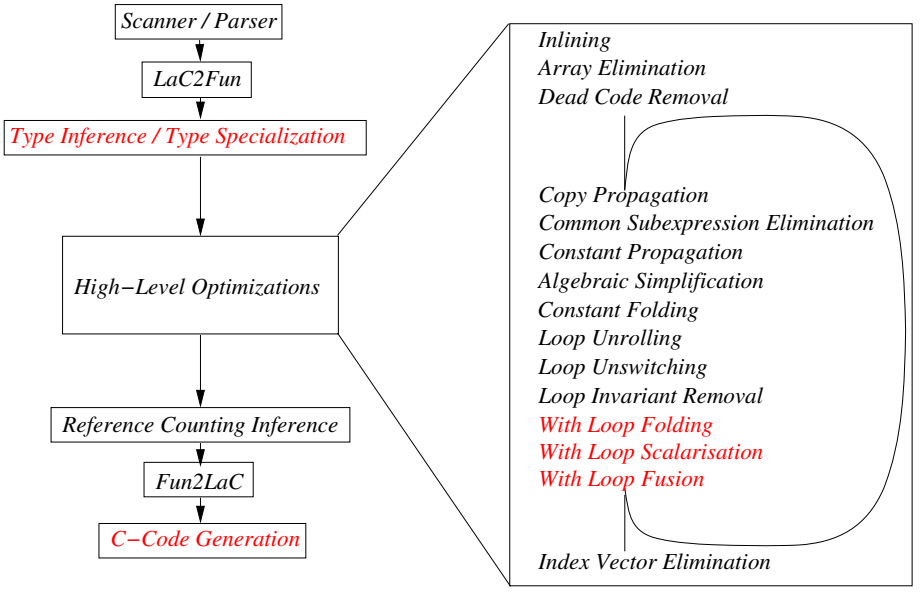


Fig. 19. Compiling SAC programs into C programs

ning and parsing the SAC-program to be compiled, its internal representation is simplified by a transformation called *LaC2Fun* which eliminates syntactical sugar such as loop constructs and (non-top-level) conditionals.

The next compilation phase implements a type inference algorithm based on the hierarchy of array types described in Section 5.3. To achieve utmost code optimizations, the actual implementation tries to specialize all array types to specific shapes. Starting from the designated `main` function, it traverses function bodies from outermost to innermost, propagating exact shapes as far as possible. In order to avoid non-termination, the number of potential function specializations is limited by a pre-specified number of instances. If this number is exceeded, the generic version is used instead.

The fourth compilation phase implements all the optimizations that can be done on the level of SAC itself. Of particular interest in this context are three SAC-specific optimizations which try to combine WITH-loops for avoiding

the creation of arrays that hold intermediate results of the overall computation. These are

- WITH-LOOP-FOLDING eliminates intermediate arrays by folding consecutive WITH-loops into single ones. It constitutes the key optimization for achieving competitive runtimes.
- WITH-loop-fusion enables sharing of loop overhead between otherwise independent WITH-loops.
- WITH-loop-scalarization transform nested WITH-loops into non-nested ones which significantly improves the memory demands.

To improve the applicability of these optimizations, constants have to be propagated / inferred as far as possible, i.e., several standard optimizations have to be included in this compilation phase as well. It also turns out that on the SaC level these standard optimizations, due to the absence of side-effects, can be applied much more rigorously than in state-of-the-art C compilers. The standard optimizations implemented in the actual compiler include Function Inlining, Constant Folding, Constant Propagation, Dead Code Removal, etc. (cf. Fig. 19.).

Many of these optimizations interact with each other, e.g., constant folding may enable WITH-LOOP-FOLDING by inferring exact generator boundaries of WITH-loops which, in turn, may enable further constant folding within the body of the resulting WITH-loop. Therefore, the optimizations are applied in a cyclic fashion, as shown on the right hand side of Fig. 19. This cycle terminates if either there are no more code changes or if a pre-specified number of cycles has been performed.

The three final compilation phases transform the optimized SaC code step by step into a C program. The first phase, called *Reference Counting Inference*, adds for all non-scalar arr ays operations that handle the reference counters at runtime. The techniques used here are similar to those developed for Sisal.

The next phase, called *Fun2LaC*, is dual to *LaC2Fun*; it reverts tail-end recursive functions into loops and inlines functions that were created from non-top-level conditionals during *LaC2Fun*.

Finally, the SaC-specific language constructs are compiled into ANSI C code.

7.1 WITH-LOOP-FOLDING

Our first optimization technique, WITH-loop-folding, addresses the composition of WITH-loops that are used in a pipelined fashion. Consider, for example, a definition

```
res = (a + b) + c;
```

where `a`, `b`, and `c` are all matrices of shape `[ 10, 10]`. Inlining the definition of `+` leads to two subsequent WITH-loops of the form

```
tmp = with(iv)
        ( [0,0] <= iv < [10,10]) : a[iv] + b[iv];
      genarray( [10,10], 0.0);
res = with(iv)
```

```
        ( [0,0] <= iv < [10,10]) : tmp[iv] + c[iv];
      genarray( [10,10], 0.0);
```

which can be combined into a single one

```
res = with(iv)
        ( [0,0] <= iv < [10,10]) : a[iv] + b[iv] + c[iv];
      genarray( [10,10], 0.0);        .
```

Technically spoken, WITH-loop-folding aims at identifying array references within the generator-associated expressions in WITH-loops. If the index expression is an affine function of the WITH-loop's index variable and if the referenced array is itself defined by another WITH-loop, the array reference is replaced by the corresponding element computation. Instead of storing an intermediate result in a temporary data structure and taking the data from there when needed, we forward-substitute the computation of the intermediate value to the place where it is actually needed.

The challenge of WITH-loop-folding lies in the identification of the correct expression which is to be forward-substituted. Usually, the referenced WITH-loop has multiple generators each being associated with a different expression. Hence, we must decide which of the index sets defined by the generators is actually referenced. To make this decision we must take into account the entire generator sequence of the referenced WITH-loop, the generator of the referencing WITH-loop that is associated with the expression which contains the array reference under consideration, and the affine function defining the index. The top of Fig. 20 shows an example for a more general situation. The generator ranges of both WITH-loops do not cover the entire array. Instead, they overlap without one being included within the other. As a consequence, the result of the folding step requires the computation of the intersection of the generators. In order to be able to do this in a uniform way, we first introduce further generators that make the default expressions explicit. The result of this extension is shown in the middle part of Fig. 20. In case of the first WITH-loop, we obtain 4 further generators with `0` being the associated expression. Similarily, the second WITH-loop has to be extended by 4 generators as well. However, since the second WITH-loop is a modarray-WITH-loop, the associated expression needs to be a selection into the array `A`. After this transformation, the generators within the WITH-loops constitute partitions of the result arrays. This facilitates the computation of generator intersections which then can be folded naively leading to the overall result shown in the bottom of Fig. 20.

7.2 WITH-Loop-Fusion

WITH-loop-fusion is similar to conventional loop fusion. It is characterized by two a more WITH-loops without data dependences that iterate over the same index space. Consider for example a function body where both, the maximum element as well as the minimum element of a given array `A` is needed. This can be specified as

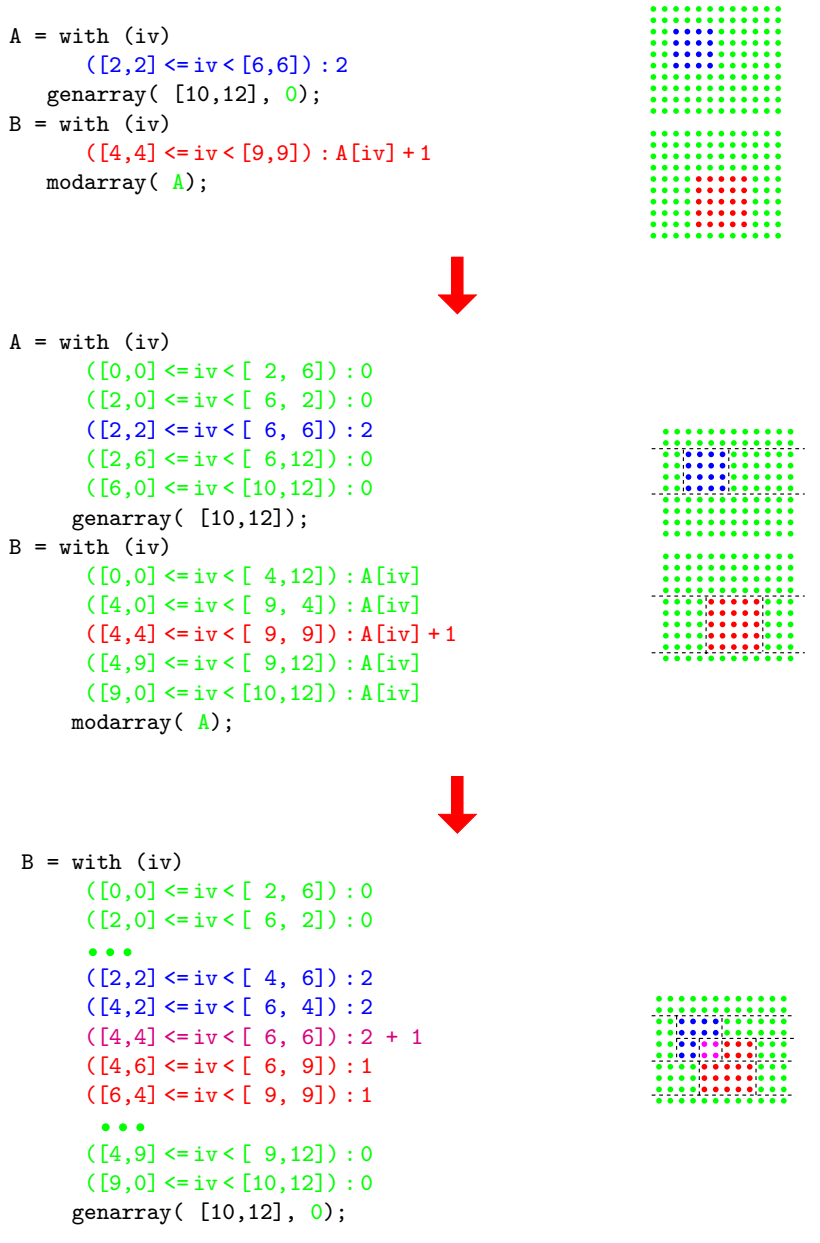


Fig. 20. WITH-loop-folding in the general case

```
minv = minval(A);
maxv = maxval(A);
```

Inlining the WITH-loop definitions for `minval` and `maxval` leads to

```
minv = with(iv)
          ( [0,0] <= iv < shape(A)): A[iv]
       fold( min, MaxInt());
maxv = with(iv)
          ( [0,0] <= iv < shape(A)): A[iv]
       fold( max, MinInt())
```

The idea of WITH-loop-fusion is to combine such WITH-loops into a more versatile internal representation named multi-operator WITH-loop. The major characteristic of multi-operator WITH-loops is their ability to define multiple array comprehensions and multiple reduction operations as well as mixtures thereof. For the example, we obtain:

```
minv, maxv = with(iv)
                ( [0,0] <= iv < shape(A)): A[iv], A[iv]
             fold( min, MaxInt())
             fold( max, MinInt())
```

As a consequence of the code transformation both values `minv` and `maxv` are computed in a single sweep. This allows us to share the overhead inflicted by the multi-dimensional loop nest. Furthermore, we change the order of array references at runtime. The intermediate code as shown above accesses large parts of array `A` in both WITH-loops. Assuming array sizes typical for numerical computing, elements of `A` are extremely likely not to reside in cache memory any more when they are needed for execution of the second WITH-loop. With the fused code both array references `A[iv]` occur in the same WITH-loop iteration and, hence, the second one always results in a cache hit.

Technically, WITH-loop-fusion requires systematically computing intersections of generators in a way similar to WITH-loop-folding. After identification of suitable WITH-loops, we compute the intersections of all pairs of generators. Whereas this leads to a quadratic increase in the number of generators for the worst case, many of the new generators turn out to be empty in practice.

7.3 WITH-Loop-Scalarization

So far, we have not paid any attention to the element types of the arrays involved. In SAC, complex numbers are not built-in, but they are defined as vectors of two elements of type double. As a consequence, an addition of two vectors of complex numbers such as

```
cv = [ Cplx(1.0,1.0), Cplx(-1.0,-1.0), Cplx(-1.0,0.0)];
res = cv + cv;
```

in fact is an addition of two matrices of doubles. However, since addition for complex arrays is defined in terms of a WITH-loop as is the scalar addition of complex numbers, after inlining we obtain

```
res = with(iv)
        ( [0] <= iv < [3]) :
        with(jv)
          ( [0] <= jv < [2]) :(cv[iv])[jv]+(cv[iv])[jv];
        genarray( [2], 0.0);
      genarray( [3], [ 0.0, 0.0]);
```

The idea of WITH-loop-scalarization is to get rid of these nestings of withloops and to transform them into WITH-loops that operate on scalar values. This is achieved by concatenating the bound and shape expressions of the WITH-loops involved and by adjusting the generator variables accordingly. For our example we obtain

```
res = with(iv)
        ( [0,0] <= iv < [3,2]) : cv[iv] + cv[iv]
      genarray( [3,2], [[ 0.0, 0.0], ...]);
```

When comparing this code against the non-scalarized version above we can observe several benefits. There are no more two-element vectors which results in less memory allocations and deallocations at runtime. Furthermore, the individual values are directly written into the result arrays without any copying from temporary vectors.

7.4 Further Reading

Material on the basic compilation scheme can be found in [Sch96, Sch03]. Cache related aspects of the compilation of WITH-loops are covered in [GKS00]. Elaboration on how to make use of various levels of shape information for generating efficient code is provided in [Kre03]. Issues around the heap management as well as compilation into concurrently executable code are presented in [Gre01]. Formal descriptions of the individual optimizations can be found in [Sch98, GST04, Sch03]. Several further papers on performance comparisons can be found on the SaC home page <http::/www.sac-home.org/>.

Besides the SaC-specific publications, there is a large body of literature on optimizing array computations in general. Good starting points are books such as [ZC91, Wol95, AK01]. In the context of functional languages, papers on optimizations towards high-performance array computations can also be found in the context of the programming language Sisal [Feo91, Can92, Can93, Can89]. Optimizations for n-dimensional array operations similar to WITH-loop-fusion in SaC can be found in the context of Zpl [Lin96, LLS98].

WITH-loop-folding and WITH-loop-fusion are based on principles that can be found in optimization techniques for algebraic data types as well. The corresponding optimization techniques are referred to as fusion and as tupling, respectively. Papers such as [Chi93, Chi94, Chi95, Gil96, HI97, NP98, Chi99, vAvGS03] contain work on these optimizations.

References

[AK01] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0.

[Ass87] I.P Sharp & Associates. *SHARP APL Release 19.0 Guide for APL Programmers*. I.P Sharp & Associates, Ltd., 1987.

[Ben91] J.P. Benkard. Extending Structure, Type, and Expression in APL-2. In *Proceedings of the International Conference on Array Processing Languages (APL'91), Palo Alto, California, USA*, volume 21 of *APL Quote Quad*, pages 20–29. ACM Press, 1991.

[Ben92] 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.

[Ber88] 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.

[Bro85] J. Brown. Inside the APL2 Workspace. *SIGAPL Quote Quad*, 15:277–282, 1985.

[Bur96] C. Burke. *J and APL*. Iverson Software Inc., Toronto, Canada, 1996.

[Can89] D.C. Cann. Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1989.

[Can92] D.C. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[Can93] D.C. Cann. *The Optimizing SISAL Compiler: Version 12.0*. Lawrence Livermore National Laboratory, LLNL, Livermore California, 1993. Part of the SISAL distribution.

[Chi93] W.N. Chin. Towards an Automated Tupling Strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'97), Copenhagen, Denmark*, pages 119–132. ACM Press, 1993.

[Chi94] W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. *Journal of Functional Programming*, 4(4):515–550, 1994.

[Chi95] W.N. Chin. Fusion and Tupling Transformations: Synergies and Conflicts. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming, Susono, Japan*, pages 106–125. World Scientific Publishing, 1995.

[Chi99] O. Chitil. Type Inference Builds a Short Cut to Deforestation. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, pages 249–160. ACM Press, 1999. ACM Sigplan Notices, 34(9).

[CK01] M.M.T. Chakravarty and G. Keller. Functional Array Fusion. In X. Leroy, editor, *Proceedings of ICFP'01*. ACM-Press, 2001.

[CK03] Manuel M.T. Chakravarty and Gabriele Keller. An Approach to Fast Arrays in Haskell. In Johan Jeuring and Simon Peyton Jones, editors, *Summer School and Workshop on Advanced Functional Programming, Oxford, England, UK, 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer-Verlag, Berlin, Germany, 2003.

[Feo91] J.T. Feo. *Arrays in Sisal*, chapter 5, pages 93–106. Arrays, Functional Languages, and Parallel Systems. Kluwer Academic Publishers, 1991.

[Gil96] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, 1996.

[GKS00] C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proc. of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, Selected Papers*, volume 1868 of *LNCS*, pages 77–95. Springer, 2000.

[Gre01] C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC - Single Assignment C*. PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2001.

[GS00] C. Grelck and S.B. Scholz. HPF vs. SAC – A Case Study. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 620–624. Springer, 2000.

[GS03] C. Grelck and S.B. Scholz. Axis Control in SaC. In T. Arts and R. Peña, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Selected Papers*, volume 2670 of *LNCS*, pages 182–198. Springer-Verlag, Berlin, Germany, 2003.

[GST04] C. Grelck, S.-B. Scholz, and K. Trojahner. WITH-Loop Scalarization – Merging Nested Array Operations. In G. Michaelson and P. Trinder, editors, *Proc. of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, UK, Selected Papers*, volume 3145 of *LNCS*, pages 118–134. Springer, 2004.

[HI97] Z. Hu and H. Iwasaki. Tupling Calculation Eliminates Multiple Data Traversals. In *Proceedings of the 2nd ICFP*. ACM-Press, 1997.

[HS04] S. Herhut and S.-B. Scholz. Towards Fully Controlled Overloading Across Module Boundaries. In c. Grelck and F. Huch, editors, *Proceedings of the 16th International Workshop on the Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany*, pages 395–408. University of Kiel, 2004.

[Hui95] R. Hui. Rank and Uniformity. *APL Quote Quad*, 25(4):83–90, 1995.

[Int84] International Standards Organization. International Standard for Programming Language APL. Iso n8485, ISO, 1984.

[Ive62] K.E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[Ive91] K.E. Iverson. *Programming in J*. Iverson Software Inc., Toronto, Canada, 1991.

[Ive95] K.E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.

[JF99] M.A. Jenkins and P. Falster. Array Theory and NIAL. Technical Report 157, Technical University of Denmark, ELTEK, Lyngby, Denmark, 1999.

[JG89] M.A. Jenkins and J.I. Glasgow. A Logical Basis for Nested Array Data Structures. *Computer Languages Journal*, 14(1):35–51, 1989.

[JJ93] M.A. Jenkins and W.H. Jenkins. *The Q'Nial Language and Reference Manuals*. Nial Systems Ltd., Ottawa, Canada, 1993.

[Kre03] D.J. Kreye. *A Compiler Backend for Generic Programming with Arrays*. PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2003.

[Lin96] C. Lin. ZPL Language Reference Manual. UW-CSE-TR 94-10-06, University of Washington, 1996.

[LLS98] E.C. Lewis, C. Lin, and L. Snyder. The Implementation and Evaluation of Fusion and Contraction in Array Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. ACM, 1998.

[MJ91] L.M. Restifo Mullin and M. Jenkins. A Comparison of Array Theory and a Mathematics of Arrays. In *Arrays, Functional Languages and Parallel Systems*, pages 237–269. Kluwer Academic Publishers, 1991.

[Mul88] L.M. Restifo Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, 1988.

[NP98] L. Nemeth and S. Peyton Jones. A Design for Warm Fusion. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages*, pages 381–393. University College, London, 1998.

[PW93] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL '93, New Orleans*. ACM Press, 1993.

[Sch96] S.-B. Scholz. **S***ingle* **A***ssignment* **C** – *Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.

[Sch98] S.-B. Scholz. With-loop-folding in SAC–Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.

[Sch01] S.-B. Scholz. A Type System for Inferring Array Shapes . In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, Sweden*, pages 65–82. Ericsson Computer Science Laboratory, 2001.

[Sch03] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[SSH+06] A. Shafarenko, S.-B. Scholz, S. Herhut, C. Grelck, and K. Trojahner. Implementing a numerical solution for the KPI equation using Single Assignment C: lessons and experience. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05*, volume ??? of *LNCS*. Springer, 2006. to appear.

[vAvGS03] D. van Arkel, J. van Groningen, and S. Smetsers. Fusion in Practice. In R. Peña and T. Arts, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, Berlin, Germany, 2003.

[vG97] J. van Groningen. The Implementation and Efficiency of Arrays in Clean 1.1. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, volume 1268 of *LNCS*, pages 105–124. Springer, 1997.

[Wol95] M.J. Wolfe. *High-Performance Compilers for Parallel Computing.* Addison-Wesley, 1995. ISBN 0-8053-2730-4.

[ZC91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.