A Compilation Scheme for a Hierarchy of Array Types

Dietmar Kreye

University of Kiel
Department of Computer Science and Applied Mathematics
D-24098 Kiel, Germany
E-mail: dkr@informatik.uni-kiel.de

Abstract. In order to achieve a high level of abstraction, array-oriented languages provide language constructs for defining array operations in a shape-invariant way. However, when trying to compile such generic array operations into efficiently executable code, static knowledge of exact shapes is essential. Therefore, modern compilers try to infer the shapes of all arrays used in a program.
Unfortunately, shape inference is generally undecidable. Therefore, most compilers either rule out all programs for which shape inference fails, or they perform no shape inference at all. In the first case the expressive power of the language is restricted, in the latter the generated code has a weak runtime performance.
This paper presents a new compilation scheme for the language SAC which combines these two approaches in order to avoid their individual shortcomings. A preliminary performance evaluation demonstrates the benefits of this compilation scheme.

1 Introduction

One of the key features of array-oriented languages such as APL [11], J [6], NIAL [13], FISH [12], ZPL [15], or SAC [19] is that they support so-called shape-invariant programming, i.e. all operations/functions can be defined in a way that allows arguments to have arbitrary extents in an arbitrary number of dimensions. This high level of abstraction gains the programmer a lot of benefits, among these are simplicity of program development, good readability and re-usability of programs.

However, the large semantical gap between such programming constructs and a given target architecture makes it difficult to develop a compiler that generates code with a high runtime performance. Sophisticated optimization techniques are needed that transform the generic program specifications into more specific ones, which in turn can be compiled into more efficiently executable code. A common method to achieve this is the so-called static shape inference, which tries to infer the shapes of all arrays used in a program as precisely as possible. Several case studies in the context of SAC [18,10] and ZPL [7,16] have shown that reasonably complex array operations can be compiled into code whose runtimes

are competitive with those obtained from rather low-level specifications in other languages such as SISAL or FORTRAN.

Unfortunately, inferring shapes statically is impossible in certain situations, e.g. if external module functions are compiled separately or input data have unknown shapes. Even worse, whenever a recursive function is applied to an argument whose shape is changing with each recursive call, shape inference may be undecidable.

There are basically two ways to handle this problem. The first approach, used for instance in the languages FISH, ZPL, and SAC, is to rule out all programs for which static shape inference fails. As a consequence, implementing certain algorithms in these languages requires some awkward code design or is even impossible. The second approach, which gives the language the full expressive power, is to refrain from static shape inference and to generate code for generic programs instead. Past experiences with such languages, for instance APL, J, or NIAL, have shown that this results in a rather weak runtime performance of the generated code.

The aim of this paper is to present a compilation scheme that combines the advantages of both approaches: generation of shape-specific code whenever exact shapes can be statically inferred and generation of more generic code, otherwise. The basic idea is to make use of a hierarchy of array types with different levels of shape information — the most specific array types specify an exact shape, whereas more general types prescribe an exact dimensionality only or contain no shape information at all — and to translate the typed programs into a corresponding hierarchy of array representations.

This compilation scheme has been developed in the context of the language SAC and has been implemented as an extension to the current SAC compiler. Therefore, Section 2 gives a brief introduction to the array concept of SAC and specifies a short SAC program that is used as a running example throughout the paper. Section 3 describes the compilation scheme for a hierarchy of array types, Section 4 presents a preliminary performance figure, and Section 5 sketches some related work. Finally, Section 6 concludes the paper and discusses future work.

2 Arrays in SAC

SAC [17] is a strict functional language based on C-syntax which primarily has been designed for numerical applications. This section gives a brief introduction to the array concept of SAC.

2.1 Representation of Arrays

SAC supports the notion of n-dimensional arrays and of high-level array operations as they are known from array languages such as APL, J, and NIAL. All arrays are represented by two vectors: a data vector containing all array elements in canonical order, and a shape vector which specifies the number of elements per

axis. For instance, a 2×3 matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ has the data vector `[1,2,3,4,5,6]` and the shape vector `[2,3]`. For reasons of uniformity scalars are considered arrays with empty shape.

2.2 Type System

For each base type (`int`, `float`, `double`, `bool`, `char`) SAC provides an entire hierarchy of array types. The most specific array types specify an exact shape, whereas more general types prescribe an exact dimensionality only or contain no shape information at all.

Basically, an array type consists of a base type followed by a shape vector. If the shape is not completely defined, some or all components of the shape vector may be replaced by wildcards (`.`, `*`, `+`). The wildcard `.` means that the extent of a certain dimension is unknown. The wildcards `+` and `*` represent arrays with at least dimensionality 1 or completely unknown dimensionality, respectively.

The hierarchy of array types can be classified into three major categories:

- arrays with known dimensionality and known extent, e.g. τ`[3,2]`, $\tau \equiv \tau$`[]`, where τ denotes a base type;
- arrays with known dimensionality but unknown extent, e.g. τ`[.]`, τ`[.,.]`;
- arrays with unknown dimensionality and unknown extent, e.g. τ`[+]`, τ`[*]`.

Figure 1 depicts the directed graph of the (reflexive, transitive and antisymmetrical) subtype relation. Vertices of the graph represent types and an edge $(\tau \to \sigma)$ means that σ is a subtype of τ. The dashed horizontal lines separate the three categories of array types.

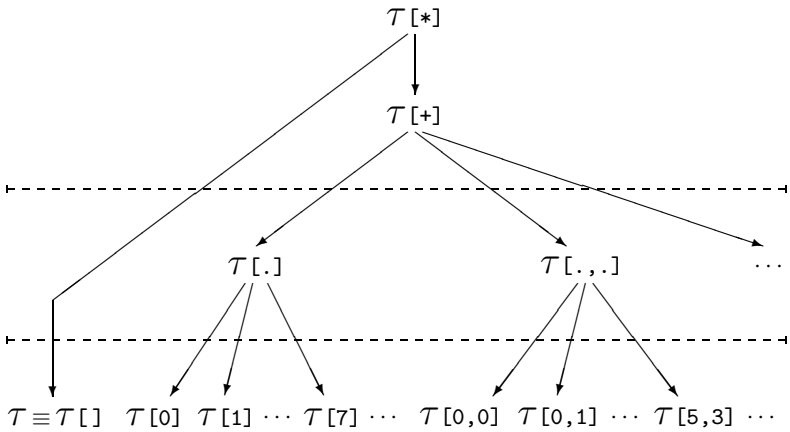

Fig. 1. Hierarchy of array types in SAC.

2.3 User-defined Array Operations

User-defined functions may have arbitrary numbers of parameters and return values. Moreover, functions can be overloaded with respect to almost[1] arbitrary argument type constellations. The semantics of SAC prescribes that for each function application the most specific instance suitable for the arguments must be used.

Consider as an example computing the determinant of a two-dimensional array. For arrays with shape `[2,2]` the operation is very simple:

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc \quad . \tag{1}$$

Higher-order determinants may be computed recursively using the Laplace expansion (along the first column):

$$\det(A) = \sum_{i=0}^{n-1} (-1)^i \cdot A_{i0} \cdot \det(\mathcal{A}_{i0}) \quad , \tag{2}$$

where A is an array of shape `[`n`,`n`]` and $\mathcal{A}_{ij}$ denotes the array A without the i-th row and j-th column. This mathematical specification of the algorithm can be translated almost literally into a SAC implementation, defining a separate function for each of the equations (1) and (2), as depicted in Fig. 2.

This example of a user-defined function is well-suited to explain the basic problems that arise when trying to compile generic SAC code into efficiently executable C code:

It is important to note that the function `Det()` is overloaded. The first instance is suitable for arrays of shape `[2,2]` only, the second one applies for all two-dimensional arrays with bigger shapes. It is the compiler's duty to resolve this overloading correctly.

Note also that the second instance of `Det()` has a non-shape-specific argument. In order to generate code with best possible runtime performance, the compiler has to specialize this instance for all required argument shapes. Moreover, these additional instances must be taken into account during resolution of function overloading.

Whenever the compiler succeeds in inferring all array shapes statically, both tasks — function specialization and resolution of overloading — are quite simple, because functions are specialized for concrete shapes only and overloading can be resolved statically then. But in general things are not that easy.

Consider, for instance, applications of the `Det()` function to arguments of type `int[3,3]`, `int[.,.]`, and `int[+]`. For the first application the compiler builds a specialization of the second instance with argument shape `[3,3]` and resolves the overloading statically. For the second application no specialization is needed, but it is not statically decidable which instance must be used. Each of the

[1] In the next section it will be shown that the signatures of overloaded functions have to meet some side conditions.

```
(1):  int Det( int[2,2] A)
(2):  {
(3):    return( A[[0,0]] * A[[1,1]] - A[[0,1]] * A[[1,0]]);
(4):  }

(5):  int Det( int[.,.] A)
(6):  {
(7):    shp = shape( A);
(8):    if (shp[[0]] == shp[[1]]) {
(9):      ret = with ([0] <= [i] < [shp[[0]]]) {
(10):             B = Elim( A, [i,0]);
(11):             det = Det( B);
(12):             val = (-1)^i * A[[i,0]] * det;
(13):           } fold( +, val);
(14):   } else {
(15):     ret = ERROR( "array is not quadratic");
(16):   }
(17):   return( ret);
(18): }
```

Fig. 2. Computing the determinant of a two-dimensional array.

three instances available — the two given by the programmer and the one built by the compiler — may be applicable. Thus, the compiler must generate additional code that chooses the matching instance at runtime. For the third application the situation is basically the same, but here an additional dynamic type check is needed to ensure that the argument represents indeed a two-dimensional array.

So, in general it can be a rather complicated and time consuming task to determine which functions have to be specialized for which argument shapes, and to build all the tailor-made code fragments that resolve the overloading.

Another problem arises in the backend of the compiler. The compiler must generate code for three different categories of array types, which include scalars. With respect to runtime efficiency it is obviously a good idea to use different representations for scalars and non-scalars. But in fact even for non-scalar arrays a hierarchy of different representations should be used. However, these different representations have to interact with each other, e.g. formal and actual arguments of a function may have different types. Therefore, they must be designed in a way that allows for cheap conversion from one representation into another.

3 Compilation

The compilation of user-defined array operations into C code is divided into four major phases:

- type inference and function specialization,
- resolution of function overloading,
- high-level code optimization,

– code generation.

In this section, these phases are described in more detail.

Note, that the following subsection about type inference and function specialization is just an excerpt from [20] and is presented here as background information only. The topic of this paper is code generation. It is demonstrated how the partial shape information provided by the type system can be exploited to generate efficiently executable code even in case of failing static shape inference. So, the innovative parts of the compilation scheme are described in the subsections 3.2 and 3.4.

3.1 Type Inference and Function Specialization

The first important task of compilation is to infer the types of all local variables. In order to achieve best possible potential for code optimizations, these types are inferred as shape-specific as possible.

Basically, the inference algorithm works as follows: Starting from the `main()` function, the type inference system traverses all function bodies from outermost to innermost, propagating shapes as far as possible. Whenever a function application is encountered, it has to be determined which function definitions are relevant for it, i.e. which function definitions are possibly needed to compute the result of the application. Then, the type of the application equals the most specific common supertype (short: MSCS) of the return types of all these function definitions. Furthermore, if only a single relevant definition is found which has not yet been specialized for the actual argument shapes, the specialization is enforced.[2]

Take as an example the two definitions of `Det()` in Fig. 2. If `Det()` is applied to an argument of type `int[2,2]`, only the first definition is relevant. However, if the argument is of type `int[3,3]`, only the second definition is relevant. Moreover, the type inference system generates a specialized `int[3,3]` version of this definition, which afterwards is the only relevant one. If the argument is of type `int[.,.]`, both definitions are relevant.

The inference algorithm sketched above has some important implications which will be formalized in the following. Consider a function application

$$f\texttt{(}\, x_1 : \tau_1 \texttt{,} \ldots \texttt{,} x_m : \tau_m \texttt{)} \quad .$$

(The notation $x : \tau$ means that for the variable x the type τ has been inferred.) Let

$$\sigma_1^k \texttt{,} \ldots \texttt{,} \sigma_n^k \; f^{(k)} \texttt{(}\, \tau_1^k \; a_1 \texttt{,} \ldots \texttt{,} \tau_m^k \; a_m \texttt{)} \; \texttt{\{ ... \}} \quad , \quad k \in \{1, ..., M\}$$

[2] To avoid non-termination, the number of possible function specializations is limited to a pre-specified number of instances. If this number is exceeded, the generic version of the function is used instead.

denote all the instances of f that occur in the given SAC program where M is the number of these instances. A function definition $f^{(k)}$ is relevant for the above application iff two conditions hold:

$$\forall i \in \{1,...,m\} : (\tau_i^k \preceq \tau_i \ \vee \ \tau_i^k \succeq \tau_i) \quad ,$$

$$\neg \ \exists l \in \{1,...,M\} \setminus \{k\} : \forall i \in \{1,...,m\} : (\tau_i \preceq \tau_i^l \preceq \tau_i^k \ \vee \ \tau_i^l = \tau_i^k) \quad .$$

(The notation $\tau \preceq \sigma$ means that τ is a subtype of σ.) The first condition ensures that actual and formal parameters of the application have compatible types. The second condition excludes all instances that are under no circumstances needed, because it is guaranteed that always another instance with more specific argument shapes can be found.

Without loss of generality, let

$$\{f^{(k)} \mid k \in \{1,...,R\}\} \quad , \quad R \leq M$$

denote the set of relevant function definitions. As already mentioned, the type of the j-th return value of the given application is equal to

$$\text{MSCS}(\{\sigma_j^k \mid k \in \{1,...,R\}\}) \quad .$$

Obviously, such a supertype does not always exist. In order to ensure that the type of the application is well-defined, the type inference system has to check whether the return values of all relevant instances have pairwise a common supertype:

$$\forall k,l \in \{1,...,R\} : \forall j \in \{1,...,n\} : \exists \sigma : (\sigma \succeq \sigma_j^k \ \wedge \ \sigma \succeq \sigma_j^l) \quad . \tag{3}$$

Note here, that a common supertype of two arbitrary types exists iff the two types have an identical base type. Thus, the constraint (3) can be simplified to:

$$\forall k,l \in \{1,...,R\} : \forall j \in \{1,...,n\} : \text{Basetype}(\sigma_j^k) = \text{Basetype}(\sigma_j^l) \quad .$$

3.2 Resolution of Function Overloading

The type inference system of the compiler infers for each function application the set of relevant function definitions. If this set contains more than a single instance, the compiler must generate additional code for the function application, that dynamically chooses the matching instance (i.e. at runtime). Moreover, it may not be statically inferable whether or not a function application is type-correct, i.e. some actual arguments' types are proper supertypes of the formal arguments' types. In this case the compiler has to generate additional code for dynamic type checks as well.

Fortunately, it turns out that it is not necessary to generate individual code that performs the resolution of overloading and the dynamic type checks for

each function application explicitly. Instead, it suffices to generate it for the most general case only. This code is written in SAC itself and inserted into the SAC program via a high-level code transformation. Subsequently, individual and optimized code for each function application is obtained by means of the usual high-level code optimizations already integrated into the compiler, like function inlining, constant folding and constant propagation.

```
(1):  int Det__i_2_2( int[2,2] A) { ... }
(2):  int Det__i_X_X( int[.,.] A) { ... Det__i( B) ... }
(3):  int Det__i_3_3( int[3,3] A) { ... Det__i( B) ... }

(4):  inline int Det__i( int[*] A)         /* wrapper */
(5):  {
(6):    if (dim( A) == 2) {
(7):      if (shape( A) == [2,2]) {
(8):        ret = Det__i_2_2( A);
(9):      } else if (shape( A) == [3,3]) {
(10):       ret = Det__i_3_3( A);
(11):     } else {
(12):       ret = Det__i_X_X( A);
(13):     }
(14):   } else {
(15):     ret = ERROR( "type error");
(16):   }
(17):   return( ret);
(18): }

(19): int main()
(20): {
(21):   int[+] A;
(22):   int[.,.] B;
(23):   int[3,3] C;
(24):   ...
(25):   a = Det__i( A);
(26):   b = Det__i( B);
(27):   c = Det__i( C);
(28):   ...
(29): }
```

Fig. 3. After resolution of function overloading.

As an example consider a `main()` function that contains applications of the familiar `Det()` function to arguments of type `int[+]`, `int[.,.]`, and `int[3,3]`. Figure 3 depicts the SAC code with resolved function overloading and explicit type checks. There exist three instances of the function `Det()`: Two of them have been given by the programmer (see lines 1, 2), the third one with argument shape `[3,3]` (line 3) is built by the type inference system via specialization of the `[.,.]` version. All three instances of `Det()` have unique names now. This is

done by adding suffixes representing the types of the arguments (e.g. `__i_X_X` for `int[.,.]`). The resolution code is implemented as a wrapper function `Det__i()` with the most general argument type `int[*]` (lines 4–18). All applications of the function `Det()` in the SAC source code are replaced by applications of this wrapper (lines 2, 3, 25–27). The wrapper selects the appropriate instance with respect to the actual shape of the argument (lines 8, 10, 12), or causes a runtime error if no appropriate instance has been found (line 15). In order to minimize on average the number of comparisons needed to choose the correct function, the choice is narrowed down by first checking the argument's dimensionality (line 6). The keyword `inline` in front of the definition of the wrapper function (line 4) directs the compiler to perform function inlining on it.

The interesting part of this code transformation is the generation of the wrapper function(s). Let again

$$\sigma_1^k\texttt{,}\ldots\texttt{,}\sigma_n^k\ f^{(k)}\texttt{(}\,\tau_1^k\ a_1\texttt{,}\ldots\texttt{,}\tau_m^k\ a_m\texttt{)\ \{ ... \}}\quad,\quad k\in\{1,...,M\}$$

denote all the instances of a function f that occur in the given SAC program, and define the set $\mathcal{I}(f)$ of these instances:

$$\mathcal{I}(f) := \{f^{(k)} \mid k \in \{1,...,M\}\} \quad .$$

Then, the following defines an equivalence relation $\sim$ on $\mathcal{I}(f)$:

$$f^{(k)} \sim f^{(l)} \quad :\Longleftrightarrow \quad \forall i \in \{1,...,m\} : \text{Basetype}(\tau_i^k) = \text{Basetype}(\tau_i^l) \quad .$$

For each equivalence class of this relation a wrapper function must be generated.

Without loss of generality, let

$$\mathcal{C} := \{f^{(k)} \mid k \in \{1,...,N\}\} \quad , \quad N \leq M$$

denote an arbitrary equivalence class of $\sim$. Then, the i-th argument of the wrapper function has the type

$$\text{Basetype}(\tau_i^k)\texttt{[*]} \quad , \quad \text{identical for all } k \in \{1,...,N\} \quad ,$$

and the j-th return value has the type

$$\text{MSCS}(\{\sigma_j^k \mid k \in \{1,...,N\}\}) \quad .$$

Again, such a supertype exists only if a constraint analogous to (3) is met:

$$\forall k,l \in \{1,...,N\} : \forall j \in \{1,...,n\} : \text{Basetype}(\sigma_j^k) = \text{Basetype}(\sigma_j^l) \quad . \qquad (4)$$

In order to generate the body of the wrapper function, the instances in $\mathcal{C}$ must be sorted according to their argument types. Unfortunately, in case of multiple arguments the subtype relation on its own is not a proper ordering relation. Consider as an example the following two instances of a function `fun`:

```
int fun( int[.] A, int[2] B) { ... }
int fun( int[2] A, int[.] B) { ... }
```

Regarding the argument `A` the second instance has a more specific type than the first one, regarding the argument `B` it is the other way round. Consider an application of `fun`, where for both arguments the type `int[2]` has been inferred. Without additional criteria it is undecidable which instance has to be chosen. This problem could be solved by introducing different priorities for different argument positions, but then, the semantics of SAC programs would depend on the order of the function arguments. To avoid confusion about the overloading mechanism, instances with such argument types are ruled out:

$$\forall k,l \in \{1,...,N\} : ((\exists i : \mathcal{T}_i^k \prec \mathcal{T}_i^l) \Rightarrow (\forall i : \mathcal{T}_i^k \not\succ \mathcal{T}_i^l)) \quad . \tag{5}$$

(The notion $\mathcal{T} \prec \mathcal{S}$ means $(\mathcal{T} \preceq \mathcal{S} \ \wedge \ \mathcal{T} \neq \mathcal{S})$.) Besides, all pairwise distinct instances must differ in their argument signatures:

$$\forall k,l \in \{1,...,N\} : \exists i \in \{1,...,m\} : \mathcal{T}_i^k \neq \mathcal{T}_i^l \tag{6}$$

Now, an ordering relation $\leq$ on $\mathcal{C}$ can be defined

$$f^{(k)} \leq f^{(l)} \quad :\Longleftrightarrow \quad \forall i \in \{1,...,m\} : \mathcal{T}_i^k \preceq \mathcal{T}_i^l \quad ,$$

and the following holds:

$$\forall k,l \in \{1,...,N\} : ((k \neq l) \Rightarrow (f^{(k)} < f^{(l)} \ \vee \ f^{(k)} > f^{(l)})) \quad .$$

Thus, it is guaranteed that the function overloading can be resolved uniquely.

3.3 High-level Code Optimizations

The SAC compiler applies several high-level code optimizations. Among these are standard optimizations [2] like function inlining, constant folding, constant propagation, dead code removal, common subexpression elimination, and loop unrolling. In general, the benefits gained by these optimizations heavily depend on the types inferred for the array objects. The more precisely the shapes have been inferred, the higher is the potential for optimizations. For example, having specialized the function `Det()` for arguments of shape `[3,3]`, the `if`-clause in its body (see Fig. 2, line 8) is redundant and can be eliminated, whereas this is not the case for the generic instance. Thus, by allowing arrays whose shapes cannot be inferred statically some overhead is introduced.

Another potential source of overhead are the wrapper functions that resolve function overloading. Since these wrappers are designed for arguments of most general types, they contain a lot of redundancy if applied to arguments of more specific types. Fortunately, this redundancy is eliminated by the optimization techniques mentioned above. Take, for instance, an application of the wrapper function `Det__i()`, as defined in Fig. 3, to an argument of type `int[3,3]`. First,

the function is inlined, i.e. the function application is replaced by the body of the wrapper function. Subsequently, constant folding detects that the conditions of all `if`-clauses can be evaluated statically, and therefore replaces the whole nesting of `if`-clauses by a call of the function `Det__i_3_3()`. Thus, as intended, the call of the wrapper function has been replaced by a call of the correct specialized function.

3.4 Code Generation

In the final compilation phase the optimized SAC code is translated into ANSI C code. The most important issue in this context is to find an appropriate C representation for the whole hierarchy of array types provided by SAC.

Array Representation. SAC arrays that are statically identified as scalars are represented in C by scalar values.

Other SAC arrays are uniquely defined by means of a data vector and a shape vector. Additionally, a reference counter [8] (short: rc) for the implicit memory management is needed. In order to get a compact representation, the reference counter and the shape vector are combined to a so-called descriptor containing reference counter, dimensionality, and all shape components. Keeping data vector and descriptor separately allows arrays to be handled uniquely irrespective of the length of their data and shape vectors, and it facilitates interfacing to external languages such as C.

The performance evaluation presented in the next section shows that this simple and uniform array representation is not sufficient for obtaining best possible runtime performance. Storing the shape in the descriptor (only) is in many situations inefficient, because the shape information is frequently used. For instance, consider a variable `A` representing an array of shape `[.,.]`. In this case it is guaranteed that all descriptors of the arrays, `A` is pointing to during program execution, contain a dimensionality of 2. So, in order to avoid costly accesses to the main memory, all references to the dimensionality of `A` should be replaced by the constant value 2. As a consequence, it is superfluous to store the dimensionality of `A` in the descriptor at all. Moreover, even the shape components of `A` are constant until a new array object is assigned to `A`. Therefore, it is recommended to buffer the shape components on the runtime stack or even in registers.

Unfortunately, it is unlikely that the C compiler will be able to apply such optimizations. In an imperative language like C any function call or any reference to a vector may cause a side-effect, hence it is almost impossible to detect that a value behind a pointer is constant or in fact superfluous. Therefore, rather than relying on the C compiler, these optimizations have to be done on the SAC level. For this purpose additional local variables are used, which always mirror the shape information of the descriptor. Whenever the shape has to be inspected, these local variables are accessed rather than the descriptor.

Figure 4 depicts the optimized C representations for the different categories of SAC arrays. For a variable `A` representing a non-scalar array, a data vector `A`

Declaration in SAC	Declaration in C
τ `[] A;`	τ `A;`

τ `[4,3] A;`	τ `*A;`
	`int *A_desc;  /* rc */`
	`const int A_dim = 2;`
	`const int A_shp0 = 4;`
	`const int A_shp1 = 3;`

τ `[.,.] A;`	τ `*A;`
	`int *A_desc;  /* rc, shp0, shp1 */`
	`const int A_dim = 2;`
	`int A_shp0;`
	`int A_shp1;`

τ `[+] A;`
and
τ `[*] A;`
	τ `*A;`
	`int *A_desc;  /* rc, dim, shp0, ... */`
	`int A_dim;`

Fig. 4. C representations for the different categories of SAC arrays.

and a descriptor `A_desc` is needed. All statically known parts of the shape are not expected in the descriptor, i.e. the corresponding descriptor entries possibly contain undefined values, but are declared as scalar constants instead. Furthermore, all variable parts of the shape are mirrored in scalar variables. Note, that mirroring the shape components is impossible for arrays of shape `[+]` or `[*]`, because the number of needed scalars is unknown during compilation.

So, the hierarchy of array types in SAC is represented by a hierarchy of C representations. As a result, the compilation scheme for array operations must be parameterized with respect to array categories, and in certain situations arrays must be converted from one representation into another.

Transformation Rules. With an appropriate array representation at hand, the code generation can be specified by means of a transformation scheme $\mathcal{C}$ which transforms SAC code into semantically equivalent C code. The basic set of transformation rules for array operations is depicted in Fig. 5.

Transformation rule (7) applies to variable declarations. The pseudo statement `DECL_ARRAY()` represents a C declaration as shown in Fig. 4.

A simple example for creating a new array is given in (8). `ALLOC_ARRAY()` allocates memory for the data vector as well as the descriptor of the array, and initializes the descriptor entries and the mirror variables. `ASSIGN_CONST()` stores the elements of the constant array `[1,2,...]` in the data vector.

Assignments of a variable `A` representing an array of type τ to a variable `B` representing an array of type σ are compiled into the statement `ASSIGN_ARRAY()` that also converts the array representation if needed. Take as an example `B:int[+]` and `A:int[.,.]`. Then, the following code is created:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\mathcal{T}\ \texttt{A;}\\ \mathit{Rest}\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{DECL\_ARRAY(}\ \texttt{A:}\mathcal{T}\texttt{)}\\ \mathcal{C}\,[\![\ \mathit{Rest}\ ]\!]\end{array}\right. \tag{7}$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{A:}\mathcal{T}\ \texttt{= [1,2,...];}\\ \mathit{Rest}\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{ALLOC\_ARRAY(}\ \texttt{A:}\mathcal{T}\texttt{)}\\ \texttt{ASSIGN\_CONST(}\ \texttt{A:}\mathcal{T}\texttt{, [1,2,...])}\\ \mathcal{C}\,[\![\ \mathit{Rest}\ ]\!]\end{array}\right. \tag{8}$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{B:}\mathcal{O}\ \texttt{= A:}\mathcal{T}\texttt{;}\\ \mathit{Rest}\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{ASSIGN\_ARRAY(}\ \texttt{B:}\mathcal{O}\texttt{, A:}\mathcal{T}\texttt{)}\\ \mathcal{C}\,[\![\ \mathit{Rest}\ ]\!]\end{array}\right. \tag{9}$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{B:}\mathcal{O}\ \texttt{= fun(}\ \texttt{A:}\mathcal{T}\texttt{);}\\ \mathit{Rest}\end{array}\right]\!\!\right] \quad \text{where} \quad \mathcal{O}'\ \texttt{fun(}\mathcal{T}'\ \texttt{A')\ \{ ... \}}$$

$$\longmapsto \left\{\begin{array}{l}\texttt{FUN\_AP(}\ \texttt{B:}\mathcal{O}\texttt{, fun, A:}\mathcal{T}\texttt{)}\\ \texttt{REFRESH\_MIRROR(}\ \texttt{B:}\mathcal{O}\texttt{)} \qquad ;\quad \text{iff } (\mathcal{T}{=}\mathcal{T}' \wedge \mathcal{O}{=}\mathcal{O}')\\ \mathcal{C}\,[\![\ \mathit{Rest}\ ]\!]\end{array}\right. \over \left\{\begin{array}{l}\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{A':}\mathcal{T}'\ \texttt{= A:}\mathcal{T}\texttt{;}\\ \texttt{B':}\mathcal{O}'\ \texttt{= fun(}\ \texttt{A':}\mathcal{T}'\texttt{);}\\ \texttt{B:}\mathcal{O}\ \texttt{= B':}\mathcal{O}'\texttt{;}\\ \mathit{Rest}\end{array}\right]\!\!\right] \quad ;\quad \text{otherwise}\end{array}\right. \tag{10}$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\mathcal{O}\ \texttt{fun(}\ \mathcal{T}\ \texttt{A)}\\ \texttt{\{}\\ \quad \mathit{Body}\\ \quad \texttt{return(}\ \texttt{B:}\mathcal{O}\texttt{);}\\ \texttt{\}}\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{FUN\_DEF(}\ \texttt{B:}\mathcal{O}\texttt{, fun, A:}\mathcal{T}\texttt{)}\\ \texttt{\{}\\ \quad \texttt{DECL\_ARRAY\_ARG(}\ \texttt{A:}\mathcal{T}\texttt{)}\\ \quad \mathcal{C}\left[\!\!\left[\begin{array}{l}\mathit{Body}\\ \texttt{return(}\ \texttt{B:}\mathcal{O}\texttt{);}\end{array}\right]\!\!\right]\\ \texttt{\}}\end{array}\right. \tag{11}$$

Fig. 5. Rules for transforming SAC code into semantically equivalent C code.

```
B = A;
B_desc = A_desc;
B_dim = A_dim;
```

However, if the inferred types are `B:int[.,.]` and `A:int[+]`, descriptor accesses are needed:

```
B = A;
B_desc = A_desc;
B_shp0 = A_desc[2];
B_shp1 = A_desc[3];
```

Another important situation arises if `A` is a scalar and `B` not, e.g. `A:int[]` and `B:int[*]`. In this case a new descriptor for `B` has to be generated.

Assignments with a function application[3] on the right hand side are transformed as shown in (10). If the types of formal and actual parameters / return

[3] Function definitions and applications are, for reasons of clarity, restricted to a single argument and a single return value here.

values are identical, the assignment is directly compiled into the statements `FUN_AP()` and `REFRESH_MIRROR()`. Otherwise additional assignments before or after the function application are inserted to convert the array representations accordingly. Consider that `A` and `B` are both non-scalar arrays. Then, `FUN_AP()` represents an application of the function `fun` to the arguments `A`, `A_desc`, `&B`, `&B_desc`, i.e. return values are implemented as reference parameters, because C functions allow a single return value only. Note, that the mirror variables of the array representation (`B_dim`, ...) are *not* passed to the function, because the function signature must be suitable for all argument types. Instead, the subsequent statement `REFRESH_MIRROR()` assures that the mirror variables are initialized with the corresponding values of the descriptor.

The rule for transforming function definitions is depicted in (11). The statement `FUN_DEF()` defines the function header by analogy with `FUN_AP()`. Besides, for each function argument a statement `DECL_ARRAY_ARG()` is inserted into the body which declares and initializes the scalar variables of the array representation. Take as an example an argument `A:int[.,.]`. In this case `A` and `A_desc` are already declared in the argument list of the function header, but the scalar variables for the shape are still missing:

```
const int A_dim = 2;
int A_shp0 = A_desc[2];
int A_shp1 = A_desc[3];
```

4 Preliminary Performance Evaluation

This section evaluates the runtime behaviour of the code generated by the compilation scheme presented in the previous section.

The hardware platform used for the measurements is a Sun Ultra-10 with 256 MB of main memory running under Solaris 7. The GNU C compiler (gcc, version 2.95.3) is used to compile the C code generated by the Sac compiler into native machine code.

The evaluation is based on the function `Det()` used in the previous sections which is applied to an array of shape `[10,10]`. This example is compiled with two different compiler versions: The first one uses the simple array representation, i.e. all non-scalar arrays are represented by data vector and descriptor only, and the second one uses the optimized array representation.

Additionally, during compilation four different strategies for function specialization are used. The first strategy builds no specializations at all, i.e. only the original two instances of `Det()` are available. Whenever `Det()` is applied to an argument whose shape is not `[2,2]`, the generic instance is used. The second strategy builds a single specialization for argument shape `[3,3]`. The third one builds specializations for argument shapes `[3,3]` and `[4,4]`. The fourth one builds instances for all needed argument shapes `[10,10]`, ..., `[3,3]`. As a consequence, all array objects have statically known shapes and function overloading can be resolved statically.

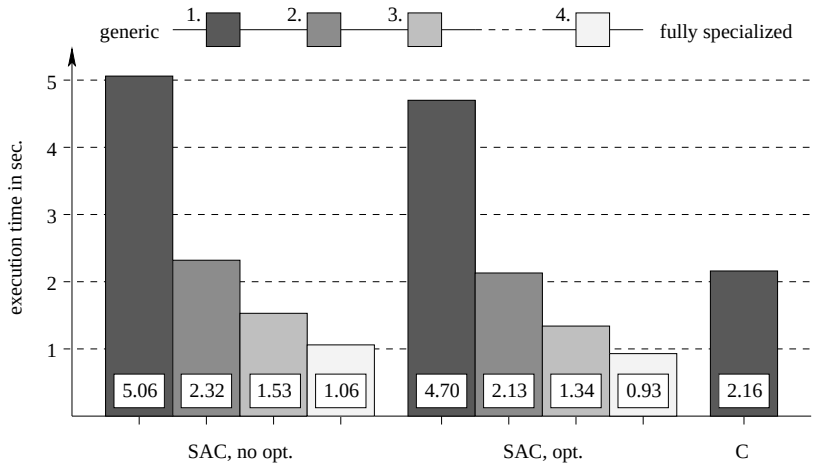

Fig. 6. Time demand for computing the determinant of a 10×10 array.

The results of the runtime measurements are shown in Fig. 6. The four bars on the left and in the middle relate to the two SAC compilers, whereas the color of each bar indicates the specialization strategy used. The single bar on the right depicts the time demand of an equivalent C implementation of the `Det()` function.

The runtime figure shows that the descriptor optimizations have a significant impact on the execution times of the generated code. Enabling the optimizations decreases the execution times by ≈ 8–14 %.

Furthermore, the measurements demonstrate that specializing functions is indeed crucial for getting best possible runtime performance. The more specializations are built by the compiler, the lower is the time demand of the generated code. The generic version without any specializations is about a factor of 5 slower than the fully specialized version. Building one or two specialized instances of the `Det()` function reduces the slowdown to a factor of ≈ 2.3 or 1.4 respectively.

However, it is also indicated that by means of the new compilation scheme even generic functions can be compiled into code with an acceptable runtime performance. Note, that it is sufficient to build a single specialization of the `Det()` function to get approximately the same execution time as the C implementation. If the compiler adds additional specializations, the SAC implementation is significantly faster than the C implementation.

5 Related Work

The concept of shape-invariant programming has been invented mainly by the designers of the language APL. Although APL allows for a very concise and elegant way of program specification, it causes difficulties when trying to execute

these generic programs efficiently. It usually requires dynamic typing and execution in an interpretating environment. Much effort has been devoted to improve runtime efficiency of such programs by application of sophisticated optimization techniques [4] and by attempts to compile them [21,9,5,3]. But code efficiency in many cases turns out to be less than satisfactory.

In order to overcome this shortcoming, languages like FISH or ZPL are designed in a way that eases static shape inference. As a result, FISH and ZPL programs, although written shape-invariantly, are compiled into very efficiently executable code. However, compilers for these languages lack the ability to generate truly shape-invariant code.

Common techniques to implement overloading, for instance in HASKELL compilers, involve the use of dictionary values [1,14]. A dictionary is a kind of a virtual function table that is passed as additional parameter to overloaded functions to resolve overloading at runtime. This dictionary-passing style must be generated by the backend of the compiler and can incur substantial overhead. In contrast, the compilation scheme presented in this paper uses static branch code written in SAC itself to implement the dynamic dispatches, therefore, no modifications in the backend are needed. A similar approach is used in the SmallEIFFEL compiler, which is described in more detail in [22]. Here, it is also shown that dynamic dispatches that have been implemented via static branch code, rather than function tables, perform better on modern hardware.

6 Conclusion and Future Work

This paper presents a compilation scheme for transforming shape-invariant array operations into efficiently executable code. This scheme has been developed in the context of the language SAC and combines two approaches: On the one hand static shape inference is performed to generate shape-specific code whenever possible, on the other hand more generic code is produced if the shape inference fails. The basic idea is to make use of a hierarchy of array types with different levels of shape information and to translate the typed programs into a corresponding hierarchy of array representations.

In order to support such a hierarchy of array types, basically two problems have to be solved. Firstly, a mechanism must be found that allows the compiler to resolve function overloading dynamically and to generate additional dynamic type checks if needed. The approach suggested in this paper achieves this by means of an elegant high-level code transformation. For each overloaded function a generic wrapper, which is also written in SAC, is generated that dynamically chooses the matching instance and performs all needed type checks. Subsequently, the code of this wrapper function is individually adapted to each function application by means of the code optimizations already integrated into the compiler.

The second problem is to find a hierarchy of array representations that is suitable for the hierarchy of array types. This paper presents two different versions of such representations. The first one is easy to implement but results in a

suboptimal runtime performance of the generated code, whereas the second one is much more complex but reduces execution times by approximately 10 %.

However, the code generated for generic SAC functions may show an extremely poor runtime performance in certain situations. That is due to the fact that some high-level code optimizations of the SAC compiler are not implemented for arrays with unknown shape yet, for instance `with`-loop folding and index vector elimination. Therefore, future work will focus on remedying this shortcoming.

References

1. L. Augustsson: *Implementing Haskell Overloading.* In: Conference on Functional Programming Languages and Computer Architecture (FPCA '93), Copenhagen, Denmark. ACM Press, 1993.
2. D. F. Bacon, S. L. Graham, O. J. Sharp: *Compiler Transformations for High-Performance Computing.* ACM Computing Surveys, 26(4), pp. 345–420, 1994.
3. R. Bernecky: *APEX: The APL Parallel Executor.* Master's Thesis, University of Toronto, Canada, 1997.
4. J. Brown: *Inside the APL2 Workspace.* ACM Quote Quad, 15, pp. 277–282, 1985.
5. T. Budd: *An APL Compiler.* Springer, 1988. ISBN 0-387-96643-9.
6. C. Burke: *J and APL.* Iverson Software Inc., Toronto, Canada, 1996.
7. B. L. Chamberlain, S. J. Deitz, L. Snyder: *A Comparative Study of the NAS MG Benchmark Across Parallel Languages and Architectures.* In: Proceedings of the ACM Conference on Supercomputing. ACM Press, 2000.
8. J. Cohen: *Garbage Collection of Linked Data Structures.* ACM Computing Surveys, 13(3), pp. 341–367, 1981.
9. G. C. Driscoll, D. L. Orth: *Compiling APL: The Yorktown APL Translator.* IBM Journal of Research and Development, 30(6), pp. 583–593, 1986.
10. C. Grelck, S.-B. Scholz: *HPF vs. SAC — A Case Study.* In A. Bode, T. Ludwig, W. Karl, R. Wismüller (Eds.): Euro-Par 2000, Parallel Processing, Proceedings of the 6th International Euro-Par Conference, Munich, Germany. Vol. 1900 of: LNCS. Springer, 2000, pp. 620–624. ISBN 3-540-67956-1.
11. K. E. Iverson: *A Programming Language.* John Wiley & Sons, 1962.
12. C. B. Jay: *Programming in FISh.* International Journal on Software Tools for Technology Transfer, 2(3), pp. 307–315, 1999.
13. M. A. Jenkins, W. H. Jenkins: *The Q'Nial Language and Reference Manuals.* Nial Systems Ltd., Ottawa, Canada, 1993.
14. M. P. Jones: *Dictionary-Free Overloading by Partial Evaluation.* In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. ACM Press, 1994.
15. C. Lin: *ZPL Language Reference Manual.* UW-CSE-TR 94-10-06, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, 1996.
16. C. Lin, L. Snyder, R. Anderson, B. L. Chamberlain, S.-E. Choi, G. Foreman, E. C. Lewis, W. D. Weathersby: *ZPL vs. HPF: A Comparison of Performance and Programming Style.* TR 95-11-05, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, 1995.

17. S.-B. Scholz: *Single Assignment 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. ISBN 3-8265-3138-8.
18. S.-B. Scholz: *A Case Study: Effects of WITH-Loop-Folding on the NAS Benchmark MG in SAC*. In C. Clack, T. Davie, K. Hammond (Eds.): Implementation of Functional Languages, 10th International Workshop (IFL '98), London, England, UK, Selected Papers. Vol. 1595 of: LNCS. Springer, 1998, pp. 216–228. ISBN 3-540-66229-4.
19. S.-B. Scholz: *On Defining Application-Specific High-Level Operations by Means of Shape-Invariant Programming Facilities*. In S. Picchi, M. Micocci (Eds.): Proceedings of the Array Processing Language Conference (APL '98), Rome, Italy. ACM Press, 1998, pp. 40–45.
20. S.-B. Scholz: *A Type System for Inferring Array Shapes*. In T. Arts, M. Mohnen (Eds.): Proceedings of the 13th International Workshop on the Implementation of Functional Languages (IFL 2001). Ericsson, Älvsjö, Sweden, 2001, pp. 53–63.
21. J. Weigang: *An Introduction to STSC's APL Compiler*. In: Proceedings of the Array Processing Language Conference (APL '89). Vol. 15 of: ACM Quote Quad. ACM Press, 1989, pp. 231–238.
22. O. Zendra, D. Colnet, S. Collin: *Efficient Dynamic Dispatch without Virtual Function Tables: The SmallEiffel Compiler*. In: Proceeding of the 12th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '97). Atlanta, Georgia, USA, 1997, pp. 125–141.