Generic Programming on the Structure of Homogeneously Nested Arrays

Stephan Herhut[1] and Sven-Bodo Scholz[1]

Dept. of Computer Science, University of Hertfordshire, United Kingdom
e-mail: {S.A.Herhut,S.Scholz}@herts.ac.uk

Abstract

In this paper we propose a new means to model and operate on nested arrays that allows for a high level of abstraction without introducing a performance penalty. We achieve this by using a nesting structure on array types which allows us to shift the nesting information of arrays from the runtime representation level to the type system level. This information can then be exploited for generic function definitions on the nesting structure of arrays which, as we show, neatly integrates with subtyping based function overloading. Finally, we demonstrate for an example how nested arrays and generic function definitions can be fully stripped out using existing optimisation techniques.

1 INTRODUCTION

Poor runtime efficiency is no longer a show stopper for using functional array languages. Techniques for enabling destructive updates via monads [LP94] or uniqueness types [vG97] in a lazy setting as well as reference counting based approaches [Can89] in a strict setting provide the grounds for reasonable runtime performance. Various compiler optimization techniques have been developed [Sch03, Sch98, CK01, GST04, GHS06] for generating more efficiently executable code from high-level specifications. Most recent application studies show that real world numerical applications can benefit two-fold from applying functional programming languages such as SAC. Programs can be specified more conveniently than in traditional, FORTRAN based languages, and they can be compiled into more efficiently executable code [SSH+06].

The expressiveness of languages such as SAC stems from its capability to handle n-dimensional arrays in a uniform way, similar to the capabilities of traditional array languages such as APL or NIAL. The effective use of n-dimensional arrays (as opposed to the vectors-of-vectors of C or SISAL) forces a language to have n-dimensional arrays as abstract data which can themselves serve as the elements of other m-dimensional arrays. Such structures are referred to as *nested arrays* and several approaches have been developed in order to provide proper programming language support for them [Ben92, Ber88, JG89]. However, all these approaches have in common that the programmer needs to be aware of the nesting of a given data structure. All operations apply on the outer nesting level only. Whenever an inner level is be to be tackled, the programmer has to make that explicit, either by denesting the array or by denoting the axes to be operated on.

We propose a novel approach which uses function overloading facilities of a type system to propagate operational behaviour to the desired level of nesting. Thus, functionality can be specified separately for all intermediate levels of nesting, enabling the programmer to keep proper data hiding for the abstract data types involved. Furthermore, this approach enables the compiler to flatten out all homogeneously nested arrays into non-nested n-dimensional arrays. To achieve that goal, only some specialization machinery needs to be put in place. Highly optimizing compilers such as `sac2c` (cf. http://www.sac-home.org/) provide enough optimization infrastructure to statically eliminate all nesting as well as all function overloading involved.

The remainder of this paper is organised as follows. In the next section, we give a short overview of SAC and its array types. Based thereon, we introduce a means to model nested arrays on the type level in Section 3. Section 4 introduces inductive definitions of generic functions on nested array types. In Section 5, an extension of the subtyping based function overloading in SAC to generic functions on nested array types is presented. Section 6 gives a full example combining nested array types, generic functions and overloading. Furthermore, the compilation into efficient code is demonstrated. Finally, after giving related work in Section 7, the conclusion is drawn.

2 SINGLE ASSIGNMENT C

In this section we give a short overview of SAC and introduce its existing array types. SAC is a purely functional first-order programming language targeted at numerical applications on homogeneous arrays. Therefore, it supports arrays as first class objects of the language. In the type system, this manifests in the form of dedicated array types.

In general, a SAC array type consists of two components: an element type $\mathcal{E}$ and a shape vector *shp*. The element type $\mathcal{E}$ – as its name suggests – gives the type of the scalar elements of the array. Note here, that for the current SAC type system, only the SAC built in base types can be used as element type.

The second component, the shape vector *shp*, gives the shape, or number of elements along each axis of the array, written in row-major order.

Array shapes and dimensionality may be fixed or dynamic. Dynamic shapes and dimensionality are specified syntactically by wild-card specifiers within the shape vector. The `*` wild card within the shape vector denotes an array of unknown dimensionality and shape. An array of integer values with unknown dimensionality thus has the type `int[*]`.

To specify the type for an array of known dimensionality but unknown shape, the `.` wild card is used to mark those dimensions within the shape vector whose extent is not known. A two-dimensional matrix of integer values with unknown shape therefore has the type `int[.,.]`. Note here that mixing the `.` wild card with dimensions of known extent is not permitted. The complete syntax of SAC

$$
\begin{array}{lcl}
\mathcal{T} & \Rightarrow & \mathcal{E}\ \texttt{[}\ \mathit{shp}\ \texttt{]} \\
 & | & \mathcal{E}\ \texttt{[}\ \texttt{.}\left[\ \texttt{,}\ \texttt{.}\ \right]^{*}\ \texttt{]} \\
 & | & \mathcal{E}\ \texttt{[}\ \texttt{*}\ \texttt{]} \\
 & & \\
\mathcal{E} & \Rightarrow & \texttt{int} \\
 & | & \texttt{double} \\
 & | & \texttt{float} \\
 & | & \texttt{char} \\
 & & \\
\mathit{shp} & \Rightarrow & \mathit{Num}\left[\ \texttt{,}\ \mathit{Num}\ \right]
\end{array}
$$

FIGURE 1. Array types in SAC**.**

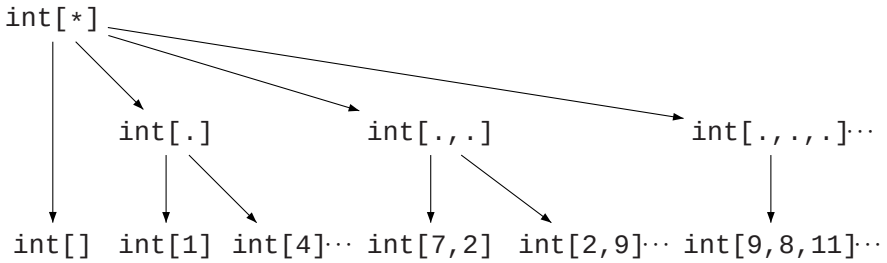

FIGURE 2. Subtyping structure on array types in SAC**.**

array types is shown in figure 1.

The three classes of array types introduced above naturally form a three-level subtyping hierarchy. A schematic outline of this hierarchy is given in figure 2.

On top of the hierarchy resides the type `int[*]`[1] which represents arrays of arbitrary dimensionality. Obviously, any array type of known dimensionality particularly is an array of arbitrary dimensionality. Thus each type from the class of array types with statically known dimensionality is a subtype of the type for arrays with statically unknown dimensionality.

Furthermore, an array of known shape is, clearly, an array of known dimensionality. So each type from the class of array types with statically known shape is a subtype of the corresponding type from the class of array types where only the dimensionality is statically known.

In SAC, this subtyping structure is exploited for the definition of shape generic functions and for overloading functions on the shape of their arguments. As an example consider the following shape generic definition of a function for element

[1] We have chosen to use the element type `int` here for demonstration purposes. The hierarchy is identical for any built in base type.

wise multiplication of two arrays.

```
1 int[] (*) ( int[] A, int[] B)
  {
3   return( _mul_(A, B));
  }
5
  int[*] (*) ( int[*] A, int[*] B)
7 {
    return( { iv -> A[iv] * B[iv] } );
9 }
```

The first instance of `*` defined in line 1 is defined on arguments of scalar type[2]. It returns the result of calling the built in primitive function `_mul_` to multiply the two scalar arguments.

As a second instance, `*` is defined in line 7 on arguments of arbitrary dimensionality. The expression `{ iv -> A[iv] * B[iv] }` used in line 9 within the definition of `*` is known as axis control notation; it iterates over the full shape of arrays `A` and `B` and applies the function `*` to each pair of scalar elements. A full explanation of the axis control notation would be beyond the scope of this paper. Details can be found in [GS03].

By using subtyping-based function overloading, both instances of `*` as defined above are combined to one overloaded function `*` which is defined for arrays of arbitrary dimensionality. Each application of `*` is then dispatched to the instance whose argument types are the least possible supertypes of the actual argument types.

As an example consider an application of `*` to two integer vectors. As `int[*]` is the least supertype of `int[.]` for which an instance is defined, the function application will be dispatched to the second instance. Within its body, the `*` operation is mapped down to scalar level using the axis control notation. Therefore, the application of `*` in line 8 is dispatched to the `int[]` instance, as `int[]` is the least supertype of `int[]` for which an instance is defined.

Finally, the `*` operation is performed on the scalar level by the built in `_mul_` function.

A complete description of SAC, its type system and the function overloading mechanism can be found in [Sch03].

3 NESTED ARRAY TYPES

In this section, building upon the array types presented above, we introduce a means to represent nested arrays. For the array types presented so far, nesting was prohibited by limiting the element type of arrays to the built in SAC types. To overcome this limitation and to allow for representing nested arrays within the

[2]Scalar values are represented as 0 dimensional arrays in SAC.

$$
\begin{array}{lcl}
\mathcal{E} & \Rightarrow & \texttt{int} \\
 & | & \texttt{double} \\
 & | & \texttt{float} \\
 & | & \texttt{char} \\
 & | & \texttt{<}\ Id\ \texttt{=}\ \mathcal{E}\ \texttt{[}\ shp\ \texttt{]}\ \texttt{>}
\end{array}
$$

FIGURE 3. Nested array types in SAC**.**

type, we introduce one further element type. An overview of the new syntax is given in figure 3.

Aside of SAC built in types, additionally a nesting constructor can now be used as element type. The nesting constructor consists of an *Id*, giving the name of the nested type, an inner element type $\mathcal{E}$ and a nesting shape *shp*, which gives the number and extent of the nested dimensions. As the inner element type $\mathcal{E}$ may even be a further nesting constructor, arbitrary finite nesting structures can be modelled this way. Note here that the nesting shape always is a fixed shape vector, wild cards as `*` and `.` are not allowed in nested types.

As an illustration of nested types as introduced above, consider the type of a *n*-element vector of complex numbers. We will use this example throughout the paper.

A complex number can be modelled as a tuple of double values, which represent the real and imaginary part. Using that model, a vector of complex numbers can be represented as a $n \times 2$ array of double values.

The corresponding nested SAC type is `< complex=double[2] >[.]`. The nesting constructor `< complex=double[2] >` specifies the element type of the array: a one dimensional, two element array of double values. As the length of the vector is unknown, the outer shape is `[.]`.

To be able to operate on the structure of nested arrays, we define two operations similar to the APL `enclose` and `disclose` functions. Other than their APL counterparts, these manipulate the type of an array instead of altering the array itself.

The `type_enclose` operation converts the type of the array passed as the second argument to a nested type by nesting the inner dimensions using the type passed as first argument. As an example, the application `type_enclose( < complex=double[2] >, A)` with `A` being an array of type `double[5,2]` converts the type of array `A` to `< complex=double[2] >[5]`. Note here that the `type_enclose` operation can only be applied if the shape of the inner dimensions of the array argument and its element type match those nested by the type argument.

Similar, the `type_disclose` operation converts the type of its argument to the type with one nesting level stripped off. The application `type_disclose( A)` with array `A` being of type `< complex=double[2] >[5]` results in the

type of array A being converted to `double[5,2]`.

Using the given nesting structure on types and the two operations introduced above, we can now define functions inductively on the nesting structure of their arguments. As an example consider the `shape` function which returns the shape vector of its first argument. For vectors of complex numbers as introduced earlier, the `shape` function can be defined as follows.

```
1 int[.] shape( < complex=double[2] >[.] A)
  {
3   A = type_disclose( A);

5   result = drop( -1, shape( A));

7   return( result);
  }
```

In the above code, first the type of array A is denested. The application of the function `shape` to array A in line 5 is thus not a recursive call of the given function itself but is instead dispatched for an argument of type `double[.,.]`[3]. Thus the shape returned by the application in line 5 is a two element vector, giving the extent of both dimensions of argument A. The consecutive application of `drop` then strips the extent of the inner dimensions, leading to the final result.

With nested array types and the `type_enclose` and `type_declose` operations at hand, the programmer can easily define nested arrays and operations thereon. Although this gives the desired level of abstraction it comes at a relatively high cost to the programmer. As, using the presented approach, defining nested arrays always comes with defining new types, the programmer cannot reuse the operations that are defined within the standard library. Instead, for every introduced nested array type, all basic operations like `+`, `*`, `shape`, etc. have to be redefined. Although a definition in most cases can be easily given, specifying all the basic operations for every type is a tedious and error-prone task.

To overcome this limitation, the next section introduces a further extension to SAC that allows definition of generic functions on the structure of array types.

4 GENERIC FUNCTIONS ON NESTED TYPES

A closer examination of the instance of function `shape` given in the last section reveals that its definition does not depend on the semantics of type `complex` but merely on the nesting structure of the type. The result of the function for the nested type is computed on the basis of the inner nesting shape and the denested type of the argument. For the function `shape` in particular, this is done by dropping the elements corresponding to the nested dimension from the shape vector of the

[3] As mixing wild cards with fixed shapes is not allowed in SAC types, a type like `double[.,2]` is no valid SAC type. Therefore the type is promoted to `double[.,.]` instead.

denested type. In general, this pattern is common to many functions that can be defined inductively on the nesting structure of arrays.

As the nesting structure is modelled in the type, instead of defining functions on a specific type, a generic definition can be given on the type constructor for nested arrays. All that is needed to give such a generic specification is the nested and denested type of the arguments and the nesting shape. Supplying this information to the algorithm allows to fully abstract from any given nesting structure.

As an example, reconsider the `shape` function we introduced in the last section. Using the above approach, a more generic definition can be given as follows.

```
generic int[.] shape( < a=b[shp] >[*] A)
{
  A = type_disclose( A);

  return( drop( - len( shp), shape( A)));
}
```

Here, instead of using the concrete types `complex` and `double` within the nesting structure of the type declaration of the first argument, we have used two type variables `a` and `b`; for the inner nesting shape, we have introduced the variable `shp`. `< a=n[shp] >[*]` thus matches any nested array type. To distinguish generic function definitions on the structure of an argument from function definitions on specific nested types, the keyword `generic` is used.

To calculate the generic shape for an array with the given type structure, similar to the concrete example for `complex`, the elements of the shape vector corresponding to nested dimensions need to be dropped. The number of elements to be dropped can hereby be deducted from the length of the nesting shape `shp`.

As a further example of a function that inductively extends over array nestings we give a generic definition of the arithmetic function `*`.

```
generic < a=b[shp] >[] (*)( < a=b[shp] >[] A,
                            < a=b[shp] >[] B)
{
  A = type_disclose( A);
  B = type_disclose( B);

  result = A * B;

  result = type_enclose( < a=b[shp] >, result);

  return( result);
}
```

In the example above, the dual to the denesting of arguments is used for the return value. The `type_enclose` operation in line 9 nests the type of the array given as its second argument with the nesting constructor given as its first argument.

Note here that the application of function `*` in line 7 within the function body is dispatched for the denested type and thus is not a recursive call of the given instance of `*`. Instead, the `*` operator is inductively extended across array nestings. As the nesting structure is finite, the recursion terminates with a call to an instance of the `*` function on a built in type as defined in section 2.

Although a generic definition can be given in that way for all primitive operations, an operation may not be homomorphic for all nested array types. To be able to mix generic and specific definitions of operations, we introduce the notion of overloading of generic functions in the next section.

5 OVERLOADING ON NESTED ARRAY TYPES

In the previous section, we introduced an inductive definition for the `*` operation on the nesting structure of arrays. This definition suffices, as long as the semantics of the multiplication operation is the same for both, the nested and denested type. This is not always the case. For instance, the semantics of `*` on complex numbers differs from the semantics of `*` on tuples of double values. As both instances are not homomorphic, no common generic instance can be defined.

An ad hoc approach to solve this problem would be to define a second function, e.g. `complexmul`, implementing the semantics of multiplication for complex numbers. Using this approach, the programmer has to be aware of which function to use, depending on the underlying semantics of a nested array. Again, choosing the correct instance by hand is a error prone task and may lead to hardly to track bugs.

Encoding the nesting structure within the type of an array allows to go for a similar approach as used for shape generic functions. As introduced in section 2, SAC exploits the subtyping structure of array shapes for function overloading. The same can be done for the nesting structure of array types by introducing a subtyping relation on nested element types.

Quite naturally, every concrete nesting type is a subtype of the generic nesting type as used for generic function definitions. Furthermore, this subtype relation neatly integrates with the existing subtyping on array shapes. An array of known nesting structure and shape is, clearly, a subtype of an array of unknown (generic) nesting structure and unknown shape. More precisely, type α is a subtype of another type β, if both its element type and shape component are in subtype relation with the element type and shape component of β. Figure 4 gives a schematic overview of this extended subtype hierarchy. The grey triangles indicate the shape subtyping hierarchy, whereas the nesting structure based subtyping is shown for two exemplary types. Subtype relations are denoted by arrows, pointing from the supertype to the subtype. The dashed arrows illustrate an exemplary subset of the combined subtyping relation.

Given this two-dimensional subtype relation, we need to uniquely define the least supertype. As shape-generic instances in general are homomorphic in SAC,

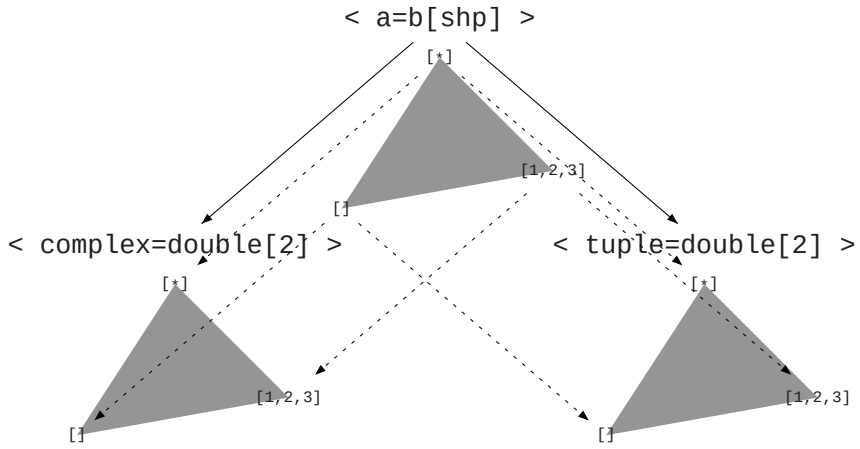

FIGURE 4. Extended SAC **subtyping hierarchy.**

whereas they might not be homomorphic with nesting-generic instances, we give precedence to the shape-generic subtyping. Thus, whenever possible, a function application is dispatched for the specific nesting type. Only if no non-nesting-generic instance matches, the nesting-generic instances are considered for dispatch.

Given this extended subtype relation, we can now overload functions on the shape and structure of their arguments. As an example for a non-generic instance specification, consider the `*` operation on values of type `complex` as given below.

```
< complex=double[2] >[] (*)( < complex=double[2] >[] A,
                             < complex=double[2] >[] B)
{
  A = type_disclose( A);
  B = type_disclose( B);

  result = [ A[0]*B[0] - A[1]*B[1],
             A[0]*B[1] + A[1]*B[0]];

  result = type_enclose( < complex=double[2] >,
                         result);

  return( result);
}
```

The function above implements the multiplication on complex numbers. The first element of the underlying double tuple is thereby treated as the real part and the second element as the imaginary part of the complex number.

Furthermore, we define a shape and structure generic version of `*` as follows.

```
generic < a=b[shp] >[*] (*)( < a=b[shp] >[*] A,
```

```
2                                   < a=b[shp] >[*] B)
{
4   return( { iv -> A[iv] * B[iv] } );
}
```

Similar to the instance of `*` presented in section 2, the above instance maps applications of `*` on arrays down to scalar level. Other than the instance defined in section 2, this instance is defined for arrays of arbitrary nested types.

Using these instances, an overloaded function `*` consisting of the above two instance and the generic instance for scalar values defined in the previous section can be defined.

As an example, consider an application of this function to two arrays of complex values. As the type of the argument is a subtype of `< a=b[shp] >[*]`, and as this is the only matching instance, the application would be dispatched to the generic instance on arrays. As that instance maps the multiplication down to scalar level, the application of `*` contained in the body has now to be dispatched for type `< complex=double[*] >[]`. Here, all instances are defined on supertypes of the argument types. As described in section 2, the least possible supertype is chosen and the application is therefore dispatched to the instance for arguments of type `< complex=double[*] >[]`. This leads to the correct result.

As nested array types are named in SAC, even different instances for the same structure but different semantics can be defined. As an example, consider an application of the above introduced overloaded function `*` to a tuple of double values of type `< tuple=double[2] >`. Although the structure of the tuple of double values is equivalent to the structure of `< complex=double[2] >`, an application of `*` would be dispatched to the generic instance for scalar values and thereby transparently mapped onto the inner dimension.

Using the subtyping based overloading approach presented above eases the use of nested arrays even further, as the programmer is fully liberated from the task to choose the correct instance depending on the semantical context.

6 PUTTING IT ALL TOGETHER

To demonstrate the interplay of nested array types, generic function definitions on the nesting structure and function overloading, we give an example of the code that is necessary to introduce a new nested array type. In a second step, we show how the generically specified code is transformed into basic SAC code without nested arrays and generic functions by applying well known techniques like function specialisation, function inlining and withloop-scalarisation[GST04].

To shorten the amount of code to be given, we introduce two shortcut notations. As applying `type_disclose` to the arguments is a quite common pattern, we use `< complex->double[2] >` in argument type position to implicitly convert the type of an argument to its denested counterpart within the function body. Dually, `< complex<-double[2] >` as return type is used as a shortcut

notation for applying the `type_enclose` function to the return values.

First, we give parts of the generic definitions for nested array types that come with the standard library and that do not need to be specified by the application programmer.

```
1  module ArrayGenerics;
2
3  use Array : all;
4
5  export all;
6
7  generic int[.] shape( < a->b[shp] >[*] A)
   {
9    return( drop( - len( shp), shape( A)));
   }
11
   generic < a<-b[shp] >[] (*)( < a->b[shp] >[] A,
13                              < a->b[shp] >[] B)
   {
15   return( A * B);
   }
17
   generic < a=b[shp] >[*] (*)( < a=b[shp] >[*] A,
19                              < a=b[shp] >[*] B)
   {
21   return( { iv -> A[iv] * B[iv] } );
   }
23
   generic < a=b[shp] >[] scalarprod( < a=b[shp] >[.] A,
25                                     < a=b[shp] >[.] B)
   {
27   return( sum( A * B));
   }
```

Most functions shown above have already been used in this paper. The additional `scalarprod` function in line 24 defines the scalar product of two arbitrary vectors. The application of sum in line 27 computes the sum of all elements of a given vector. Using the above generic function definitions, the `complex` type introduced earlier can now be defined along with multiplication operations on it by the following code.

```
module Complex;
2
import ArrayGenerics : all;
4
< complex<-double[2] >[] (*)(
```

```
6                               < complex->double[2] >[] A,
                                < complex->double[2] >[] B)
8 {
    return( [ A[0]*B[0] - A[1]*B[1],
10              A[0]*B[1] + A[1]*B[0]]);
  }
```

The `import` statement in line 3 imports all functions and generic definitions from module `ArrayGenerics` into the namespace of module `Complex`. Thus, the generic definitions for `*`, `shape` and `scalarprod` become immediately available for type `< complex=double[2] >`. To represent the difference in semantics of the `*` operation on complex numbers, we further overload the `*` operation for the type `< complex=double[2] >[]`.

As the above example shows, using the techniques presented in this paper, introducing new nested array types comes at a significant lower cost to the programmer as he can reuse most existing generic definitions.

As an example, consider the following application of the function `scalarprod` as generically defined in `ArrayGenerics` to two vectors `A` and `B` of type `< complex=double[2] >[.]`.

```
1 S = scalarprod( A, B);
```

The application is dispatched to the single generic instance of `scalarprod` defined earlier. As the arguments of the application are vectors, the application of `*` within `scalarprod` is dispatched to the generic array instance, which maps the application to scalar level. As we defined an explicit instance for scalar arguments of type `< complex=double[2] >[]`, within the generic array instance of function `*` the instance for scalar complex values is used for dispatch. This instance finally computes the product.

Although an application of `*` conceptually results in rather many dispatch decisions and function applications, this does not necessarily imply a bad runtime behaviour. By specialising `scalarprod` and `*` for type `< complex=double[2] >` and consecutively applying function inlining and loop scalarisation, all dispatch decisions can be made statically and function aplications can be fully removed. One retains the following code:

```
1 A = type_disclose( A);
  B = type_disclose( B);
3
  S = sum( { [i] -> [ A[i,0]*B[i,0] - A[i,1]*B[i,1],
5                     A[i,0]*B[i,1] + A[i,1]*B[i,0]]});
```

Note here that the entire operation is performed on the denested type. The multiplication within the axis control notation in line 4 has been scalarised and is now performed directly on the elements of the underlying `double` array. By doing so, array nesting has been completely removed from the algorithm.

As all function applications can be statically dispatched for the corresponding base types of their arguments, the `type_disclose` operations can be removed as well. The resulting code thus fully resembles a definition without nested array types and generics.

7 RELATED WORK

In the field of array programming languages, NIAL[JJ93] is a prominent example of a language with built in support for nested arrays. Other than the approach presented in this paper, in NIAL the nesting structure of an array has to be explicitly defined by the programmer using a so called nesting vector. Furthermore, it is the programmers responsibility to explicitly state the mapping of each function application to the nesting level it is intended to operate on.

In APL[BB93], nested arrays are handled using explicit applications of `enclose` and `disclose` functions. `enclose` boxes an array into a scalar, whereas the dual operation `disclose` unboxes it. As with nesting vectors in NIAL, the programmer has to be aware of the nesting structure and has to manually insert the appropriate nesting and denesting operations to extend function applications across array nestings.

For other functional languages like HASKELL or CLEAN, generic programming extensions for algebraic datatypes have been proposed[JJ97, Hin00, AP02]. Similar to our approach, they allow to define generic instances on the type constructor level instead of on types. To our best knowledge, these approaches have not been applied to other datatypes or even arrays.

8 CONCLUSION

We have introduced a new means to model and operate on nested arrays that eases program specification by facilitating a high level of abstraction without introducing a performance penalty.

As a key concept, we presented a nesting structure on array types. Using types to represent nestings allows us to shift the nesting information of arrays from the runtime representation level into the type system.

This approach further enables us to exploit the structural information for function definitions. We did so by introducing generic function definitions on the structure of nested arrays types. Moreover, we gave an example of how this neatly integrates with the function overloading capabilities of SAC.

Finally we gave a demonstration on how nested arrays and generic function definitions can be fully stripped out using existing optimisation techniques. The impact of array nestings and generic program specification on the runtime is thus expected to be minimal.

The extension of SAC as proposed in this paper introduces an additional subtype hierarchy. To develop a means to add further subtyping hierarchies to the ex-

isting SAC subtyping relation and making them available for function overloading, remains as future work. One interesting aspect here could be to add homomorphic subtyping on built in types as proposed in [SS05].

9 ACKNOWLEDGEMENTS

This work was funded by the European Union ÆTHER project.

REFERENCES

[AP02] Artem Alimarine and Marinus J. Plasmeijer. A generic programming extension for clean. In *IFL '02: Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, pages 168–185, London, UK, 2002. Springer-Verlag.

[BB93] Robert Bernecky and Paul Berry. *SHARP APL Reference Manual*. Iverson Software Inc., 33 Major St., Toronto, Canada, 2nd edition, 1993.

[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.

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

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

[GHS06] C. Grelck, K. Hinkfuß, and S.-B. Scholz. With-Loop Fusion for Data Locality and Parallelism. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05, Selected Papers*, volume ??? of *LNCS*. Springer, 2006. to appear.

[GS03] C. Grelck and S.-B. Scholz. Axis Control in SAC. In R. Peña and T. Arts, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, 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.

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, New York, NY, USA, 2000. ACM Press.

[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.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997*, pages 470–482. ACM Press, New York, 1997.

[LP94] J. Launchbury and S. Peyton Jones. Lazy Functional State Threads. In *Programming Languages Design and Implementation*. ACM Press, 1994.

[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.

[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.

[SS05] A. Shafarenko and S.-B. Scholz. General Homomorphic Overloading. In C. Grelck and F. Huch, editors, *Proc. of the 16th International Workshop on Implementation of Functional Languages (IFL'04), Lübeck, Germany, Selected Papers*, volume 3474 of *LNCS*, pages 195–210. Springer, 2005.

[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*, LNCS. Springer, 2006. to appear.

[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.