Generic Programming on the Nesting Structure of Arrays

Stephan Herhut Sven-Bodo Scholz Clemens Grelck

Dept. of Computer Science, University of Hertfordshire, United Kingdom
{S.A.Herhut,S.Scholz,C.Grelck}@herts.ac.uk

Abstract

Multi-dimensional arrays lack expressiveness with respect to logical layers of data; they provide no means to encode that, for example, a three-dimensional array of double values is used to represent a matrix of RGB-values. Existing approaches, *e.g.*, boxing and unboxing the inner dimensions of an array, allow the programmer to encode additional structure, but make programming on nested arrays a tedious and error-prone task: nesting and de-nesting operations have to be explicitly encoded in function applications. Apart from the additional work spent on program specification, this scattered encoding of structural information renders refactoring of widely used data structures difficult.

We propose a new means to model the structure of homogeneously nested arrays in the type system, rather than the actual data. We exploit this additional type knowledge for subtyping-based function overloading, liberating the programmer from explicitly encoding nesting operations in function applications. Furthermore, we propose and demonstrate a nesting-structure generic programming extension that allows us to define user-defined homogeneously nested array-types without the usual boiler-plate code.

1. Introduction

The term *generic programming* has gained popularity in the programming language community; it is widely used for language features that allow programmers to abstract over certain properties of data and thus facilitate software reuse.

In object-oriented languages like C++ [Str00], JAVA [GJSB05] or C# [Int03], the term generic programming is used in the context of *parametric polymorphism* [BML97, BOSW98, OW97], *i.e.*, the specification of classes that are parameterised with respect to the type of the contained data. As an example consider a class for lists of one specific, but arbitrary datatype. Here, the type of the list's elements is a parameter of the list class.

Functional languages such as HASKELL [Pey03] and CLEAN [PvE01] have recently been extended by generic programming facilities on their predominant data structure: algebraic datatypes. *Datatype generic programming* [JJ97, Hin00, AP02] allows the programmer to specify algorithms on the structure of a datatype rather than on the actual datatype itself. Applications of this technique include, for example, generic definitions of pretty printers and parsers for arbitrary tree structures.

A different approach to generic programming is taken in array languages such as APL [Int93], NIAL [JJ93], J [HI04] and SAC [Sch99, Sch03]. Their expressiveness stems from the use of one single data structure, *i.e.*, multi-dimensional arrays, and the introduction of a *shape-generic programming* model thereon; it allows the programmer to define algorithms on arrays of statically unknown shape (*i.e.*, the extent along each axis of the array) or even statically unknown dimensionality (*i.e.*, the number of axes).

Multi-dimensional arrays are a versatile data structure. Combined with shape-generic programming, they offer a high level of abstraction and good opportunities for software reuse. However, from a software engineering perspective, they lack a certain kind of expressiveness. Take as an example a 10×10-matrix of complex numbers, represented by a $10 \times 10 \times 2$-array of double values. Even defining simple arithmetic operations on this representation of complex matrices is awkward. Software engineering calls for a solution that properly distinguishes between the two logical layers of defining arithmetic on individual complex numbers represented by two-element vectors of double values and on matrices of some numerical values in an element-wise manner. Examples of this kind are manifold in practice, *e.g.*, pictures comprising of RGB-colour-encoded pixels or films represented as vectors of pictures.

In APL and J this separation of logical layers can be achieved by explicitly boxing inner dimensions, thereby creating a scalar. In the above example of complex numbers, the inner dimension, *i.e.*, two-element vectors of double values, can be boxed into scalar values, *i.e.*, complex numbers. These scalar values can then be used as elements of the outer array, leading to a 10×10 matrix of complex numbers. Consecutive function applications operate only on the non-boxed outer dimensions.

Although boxing inner dimensions allows the programmer to encode logical layers of data, it has its drawbacks. First, boxing the inner dimensions into scalar values inhibits shape-generic programming. As an example, consider adding two 10×10 matrices of complex numbers. This is easily done by element-wise adding the underlying $10 \times 10 \times 2$ array of double values, but to achieve this, the inner dimensions need to be unboxed first. Second, the boxing and unboxing operations need to be explicitly inserted into the code by the programmer. Apart from being tedious and error-prone, this requires the full knowledge of the underlying structure of an array. Even more, this knowledge is scattered throughout the code, making refactoring the structure of widely used arrays a difficult task.

In the following, we propose a novel representation that we have developed in the context of the functional array language SAC. It makes heavy use of subtyping, overloading and generic programming techniques. We explicitly model nested array-types in our type system: we introduce types of the form `<complex=double[2]>`. These *nesting constructors* combine a type definition and the name of the defined type as a new type, which may serve as the base type of an array. By means of these nesting constructors we can define array operations on the nested elements of an array. Nesting constructors as types enable full use

of overloading, and definition of different instances of functions on types that are structurally identical, but semantically require different handling in certain situations, *e.g.*, multiplication of complex numbers *vs.* element-wise multiplication of pairs of double values.

While this solution is adequate in principle, it lacks convenience. Nested array-types that share structural properties often do not require different handling: Unlike multiplication, summation of complex numbers and element-wise summation of pairs of double values are the same. In general, nested array-types bring the burden of redefining many basic operations. Although this is usually not difficult, it is tedious and error-prone.

We propose to acompany shape-generic programming by a further means of generic program specification: generic programming on the nesting structure of arrays. To allow us to specify abstract operations that solely reflect the nesting structure, but not the specific needs of certain nested types like complex numbers, we introduce generic nesting constructors of the form `<a=b[shp]>`, where `a`, `b` and `shp` are placeholders for concrete types and shape specifications. Using these nesting constructors, we define generic functions on arbitrary homogeneously nested arrays.

In order to support both, generic function implementations and, where necessary, specific function implementations, we define a subtyping relation on nesting constructors in which each specific nesting constructor, *e.g.*, `<complex=double[2]>`, is a subtype of the generic nesting constructor `<a=b[shp]>`. As a result, SAC supports two orthogonal hierarchies of subtyping, overloading and generic programming: on the level of element types through nesting constructors and on the level of array types through varying levels of static knowledge of shapes. These hierarchies let us provide a generic library of abstract basic nested operations. Whenever we need to introduce a new nested array-type, we simply import that library, then overload only those operations by specific implementations where the new nested array-type requires a non-standard solution.

The remainder of this paper is organised as follows. In the next section, we give a brief overview of SAC and its array types. In Section 3 we introduce nesting constructors. Section 4 illustrates 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. Related work is discussed in Section 7, and some conclusions are drawn in Section 8.

2. SAC — Single Assignment C

SAC is a purely functional first-order array programming language with a syntax that is largely adopted from C and genuine support for multi-dimensional arrays: any type in SAC describes an array. As shown in Figure 1, SAC array types consist of two components: an element type, which can be chosen from a fixed set of built-in types, and a shape specification, which defines the shape of an array to varying degrees of accuracy.

$$
\begin{array}{lcl}
\mathit{Arraytype} & \Rightarrow & \mathit{Elementtype}\ \ \mathit{Shape} \\
\mathit{Elementtype} & \Rightarrow & \texttt{int} \mid \texttt{double} \mid \texttt{float} \mid \texttt{char} \mid \texttt{bool} \\
\mathit{Shape} & \Rightarrow & \texttt{[}\ \lceil\ \mathit{Num}\ \lceil\ \texttt{,}\ \mathit{Num}\ \rfloor^{*}\ \rfloor\ \texttt{]} \\
 & \mid & \texttt{[}\ \texttt{.}\ \lceil\ \texttt{,}\ \texttt{.}\ \rfloor^{*}\ \texttt{]} \\
 & \mid & \texttt{[}\ \texttt{*}\ \texttt{]}
\end{array}
$$

Figure 1. Array types in SAC.

To this effect we support three classes of array types. The first class of array types specifies the shape of an array using a comma-separated list of non-negative numbers. The length of that list equals the number of axes or dimensions of the array, and each individual number gives the extent of the array along that axis or dimension. For example, an integer matrix of 10×2 elements is of type `int[10,2]`. This class of array types also includes scalars, which in SAC, as in APL and J, are considered zero-dimensional arrays. Hence, a scalar floating point number hast type `float[]`.

The second class of array types lifts the restriction of fixed array shapes, but still specifies the exact number of axes. Syntactically, we achieve this by using dots, instead of concrete numbers, in the shape vector. For example, an arbitrarily-sized matrix of boolean values has type `bool[.,.]`.

Finally, the third class of array types in SAC abstracts from both the extent of an array along its axes and the number of axes or dimensions itself. For any given element type α this class consists of a single element only. In the style of regular expression syntax, we denote this type α`[*]`.

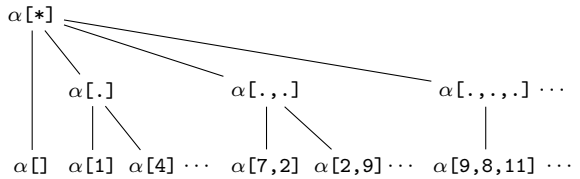

Figure 2. Subtyping structure on array types in SAC.

Our three classes of array types naturally form a subtyping hierarchy as shown in Figure 2. For any element type α, an array type with fixed shape naturally is a subtype of the corresponding array type that leaves the concrete shape unspecified, but prescribes the number of dimensions. Likewise, an array type with fixed dimensionality naturally is a subtype of the most general type that abstracts from a concrete dimensionality.

To achieve this strictly separated hierarchy of shape knowledge and maintain a tree structure as subtyping hierarchy, SAC does not support types that mix dots with numbers and, hence, abstract from concrete extents along selected axes only.

We exploit this subtyping hierarchy for function overloading and for shape-generic programming. As an example consider the shape-generic definition of a function for the element-wise multiplication of two arrays, as shown in Figure 3.

```
1 double[] (*) ( double[] A, double[] B)
2 {
3   return( _mul_SxS_(A, B));
4 }

5 double[*] (*) ( double[*] A, double[*] B)
6 {
7   return( { iv -> A[iv] * B[iv] } );
8 }
```

Figure 3. Shape generic definition of element-wise multiplication of two arrays.

The first definition of function `*` takes two scalar arguments and computes by the help of a built-in scalar-scalar primitive `_mul_SxS_`. In contrast, the second definition of function `*` takes two arguments of any dimensionality and shape. The expression `{ iv -> A[iv] * B[iv] }` essentially is a SAC array comprehension that maps the `*` operator to pairs of corresponding elements of arrays `A` and `B`. Since a deeper understanding of the SAC expression language is not required in the following, we refrain from an appropriate explanation and refer the interested reader to [Sch03] and [GS03] for reference.

Function overloading and subtyping effectively combine the two definitions of function * in Figure 3, referred to as *instances* in the following, into a single *overloaded* function *. The resulting overloaded function is defined on the superset of its instances. Thus, in our example, the resulting overloaded function * is defined on arrays of any shape and dimensionality.

Each application of * is then dispatched to that available instance whose formal arguments have the least supertype of the corresponding actual arguments' types. As an example, consider an application of function * to two double vectors. As `double[*]` is the least supertype of `double[.]` 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. Therefore, the application of function * in line 7 is dispatched to the `double[]` instance, as `double[]` is the least supertype of `double[]` for which an instance is defined. Finally, the * operation is performed on the scalar level by the built-in `_mul_SxS_` function.

```
1 double[*] (+) ( double[*] A, double[] B)
2 {
3   return( { iv -> _add_SxS_( A[iv], B) } );
4 }

5 double[*] (+) ( double[] A, double[*] B)
6 {
7   return( { iv -> _add_SxS_( A, B[iv]) } );
8 }
```

Figure 4. Ambiguously overloaded function with two non-covariant instances.

In general, such an instance may not be uniquely defined. We give an example in Figure 4, using two instances for addition on arrays and scalar values. The first instance (cf. line 1ff.) is defined for an array as first argument and a scalar value as second. The second instance (cf. line 5ff.) is defined for the opposite argument combination. Here, the first argument is a scalar value and the second an array. We use the same SAC array comprehension as in the previous example, given in Figure 3, to map the operation down to scalar level. The actual addition is then performed by the built-in scalar-scalar `_add_SxS_` operation in lines 3 and 7, respectively.

When applying the resulting overloaded function + to two scalar values, both of the instances match equally well. The first instance matches on the first argument, as the formal and actual arguments both have type `double[]`. The same holds true for the second instance and second argument. For the remaining arguments, both instances are defined on type `double[*]`, which is a supertype of the actual argument's type `double[]`. Overall, both instances match, not exactly, but equally well on the given arguments.

To ensure a unique dispatch, we rule out instance definitions as those shown in the previous example. We achieve this by enforcing a strict order on function instances with respect to function dispatch, *i.e.*, we ensure that for given actual arguments and every two matching instances, one is a closer match than the other. Formally, this idea is captured by the concept of covariance [Pie02]: For each pair of function instances a and b with argument types $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$, we require that for all $i \in 1, \ldots, n-1$ the statement $\alpha_i \prec \beta_i \Rightarrow \alpha_{i+1} \prec \beta_{i+1}$ holds. The symbol $\prec$ thereby denotes the subtype relation, *i.e.*, $\alpha \prec \beta$ denotes α being subtype of β.

For the example given in Figure 3, this requirement is met as for both arguments the type `double[]` of the first instance is a subtype of the type `double[*]` of the second instance. However, the second example in Figure 4 does not meet this condition. Although the type of the first argument of the first instance, `double[*]`, is a supertype of `double[]`, the type of the corresponding argument of the second instance, this does not hold for the types of the second arguments.

3. Nested Array Types

The existing type system of SAC only covers multi-dimensional arrays of primitive data. In order to extend our type system to homogeneously nested arrays, we introduce a special element type, which we refer to as *nesting constructor*; Figure 5 shows its syntax. The nesting constructor comprises an identifier, associating a name with the nested array-type, an inner element-type and a nesting shape, which specifies the number of nested dimensions and their extents, respectively. As the inner element-type may again be a nesting constructor, we can model arbitrary finite nesting structures in this way. We require a concrete nesting shape to ensure homogeneity of nested arrays.

$$
\begin{array}{rcl}
\mathit{Elementtype} & \Rightarrow & \texttt{int} \mid \texttt{double} \mid \texttt{float} \mid \texttt{char} \mid \texttt{bool} \\
& \mid & \texttt{<}\ \mathit{Id}\ \texttt{=}\ \mathit{Elementtype}\ \mathit{FixedShape}\ \texttt{>} \\
\\
\mathit{FixedShape} & \Rightarrow & \texttt{[}\ \lceil\ \mathit{Num}\ \lfloor\ \texttt{,}\ \mathit{Num}\ \rfloor^{*}\ \rceil\ \texttt{]}
\end{array}
$$

Figure 5. Nested array-types in SAC.

As an illustration of nested types, consider a vector of complex numbers. An individual complex number can be represented as a two-element vector of double values. Hence, an n-element vector of complex numbers can be represented as an $n \times 2$ array of double values. The corresponding SAC array type is `<complex=double[2]>[.]`. Likewise, we may define a nested array-type for vectors of tuples of two double values as `<tuple=double[2]>[.]`.

Although both types essentially express the same structural information, they differ in the name introduced by the nesting constructor. The ability to distinguish between types of identical structure permits different treatment of nested types. As an example consider element-wise multiplication. While the element-wise multiplication of two vectors of tuples boils down to the element-wise multiplication of the tuples themselves, we certainly prefer to have proper complex arithmetic in the case of the vector of complex numbers.

To be able to operate on the structure of nested arrays, we define two operations `type_enclose` and `type_disclose`. They are effectively type coercions and do not actually modify the representation of an array. 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. For 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 type `<complex=double[2]>[5]`. Of course, `type_enclose` requires the type of the second argument to match the nesting constructor given as first 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]`.

The `type_enclose` and `type_disclose` operations serve a similar purpose as the APL `enclose` and `disclose` functions; both are used to specify nested arrays. In APL, the `enclose` function boxes an array, which can then be used as an element of another array. Its dual, the function `disclose` unboxes such a boxed array, leading to the original array. Using these two operations, arbitrary nesting structures of arrays can be created and resolved at runtime.

In contrast to the APL functions, `type_enclose` and its dual `type_disclose` solely convert types and do not modify arrays

as such. This way, the concept of nested arrays is lifted from the level of runtime representations of arrays to the level of the type system. As a consequence, the use of `type_enclose` and `type_disclose` has no impact on the runtime representation of an array. In particular, no conversion of array representations at runtime is needed.

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

4   result = drop( [-1], shape( A'));

5   return( result);
6 }

7 int[.] shape( double[*] A)
8 {
9   return( _shape_( A));
10 }

11 int[.] shape( <tuple=double[2]>[.] A)
12 {
13   A' = type_disclose( A);

14   result = drop( [-1], shape( A'));

15   return( result);
16 }
```

Figure 6. Function `shape` for vectors of complex numbers and tuples.

Using the nested array-types and the `type_enclose` and `type_disclose` operations introduced above, we can now define functions inductively on the nesting structure of their arguments. As an example, consider the `shape` function, which yields the shape vector of its argument. Figure 6 provides implementations of the `shape` function for both vectors of complex numbers and vectors of tuples. First, the type of argument array `A` is de-nested. Thus, the application of the function `shape` to array `A'` in line 4 is not a recursive application, but instead is dispatched for an argument of type `double[.,.]`[1]. For primitive element types, the function `shape` is defined using the built-in primitive similar to the definition of the `*` operation in Figure 3. Using this instance of `shape`, the value returned by the application in line 4 is a two-element vector giving the extent of both dimensions of argument `A'`. The subsequent application of `drop` with first argument `[-1]` then strips the extent of the inner dimension, leading to the final result.[2] The definition of `shape` on vectors of tuples follows the same idea.

So far, the nesting shape and inner element-type of a nested array-type are re-defined each time a type is used. To ensure the consistent use of type names and their corresponding nesting shape and inner element-type, we introduce type definitions. As an example, consider the two type definitions given in Figure 7. The first type definition (cf. line 1) defines complex numbers as introduced in Figure 6. A definition for tuples of double values is given in line 2. The syntax for type definitions closely resembles C: the keyword `typedef` is followed by the nested array-type and the name of the defined type.

[1] As mixing dots with numbers is not supported by SAC array types, the type is promoted to `double[.,.]`.

[2] Note here that a negative vector as first argument to `drop` encodes stripping elements from the end of the shape vector.

```
1 typedef double[2] complex;

2 typedef double[2] tuple;
```

Figure 7. Type definitions for complex numbers and tuples of double values.

Given these type definitions, the specification of the nesting shape and inner element-type in function signatures becomes redundant; it suffices to give the name of a nested array-type. Apart from added brevity, this can be used to define abstract datatypes by hiding the actual definition of a type from the programmer. However, to keep our examples self contained, we will use explicit nested array-types throughout the paper.

4. Generic Functions on Nested Types

The example of Figure 6 illustrates how functions like `shape` can straightforwardly be implemented on arbitrary homogeneously nested arrays. As the two types — vectors of complex numbers and vectors of tuples of double values — are structurally identical, so are the corresponding definitions of the `shape` functions. This illustrates a downside of having named nested array types: Each newly introduced nested array type requires redefinition of basic operations such as `+`, `*` and `shape`. While definitions of these functions in most cases are not complicated to derive, specifying all the basic operations for every nested array type is both tedious and error-prone. This calls for a more generic solution.

A closer examination of the function `shape` reveals that its definition does not depend on the semantics of type `complex` or `tuple`, but merely on the nesting structure of the type: Its result is computed from the inner nesting-shape and the de-nested type of the argument. This pattern is common to many functions that can be defined inductively on the nesting structure of arrays. Instead of defining functions on a specific type, we introduce a generic definition based on the nesting constructor. All that is needed to give such a generic specification is the nested and de-nested type of the arguments and the nesting shape. Figure 8 shows a generic definition of the `shape` function.

```
1 int[.] shape( <a=b[shp]>[*] A)
2 {
3   A' = type_disclose( A);

4   return( drop( [-len( shp)], shape( A')));
5 }
```

Figure 8. Nesting generic definition of function `shape`.

Instead of using the concrete types `complex` and `double` in the nesting constructor of the type declaration, we use a type variable `a` to represent the name of the nested type and two placeholders `b` and `shp` for its element type and the inner nesting-shape, respectively. The type variable `a` is implicitly universally-quantified, i.e., it ranges over all nested array-types in the current module. The actual computation of the shape is quite similar to the non-generic versions of `shape` in Figure 6. The number of elements to be dropped from the shape of the de-nested array is deduced from the length of the nesting shape, represented by `shp`. Syntactically, generic instances can be distinguished from type-specific instances by the use of a variable as nesting shape rather than a concrete list of numbers.

The type variable `a` in the example given in Figure 8 serves two purposes. First, it is used as a type variable within the function signature of the function `shape`. Thus, the above function definition

```
1 <a=b[shp]>[] (*)( <a=b[shp]>[] A,
2                   <a=b[shp]>[] B)
3 {
4   A' = type_disclose( A);
5   B' = type_disclose( B);

6   result = A' * B';

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

8   return( result');
9 }
```

Figure 9. Nesting generic definition of multiplication on scalars.

defines the function `shape` for all nested array types `a`. Second, the type variable is used to bind an identifier `a` within the function body to the type of the argument of the function. This is merely a syntactical convenience to simplify the specification of `type_enclose` operations. The two other variables, the nesting type `b` and the nesting shape `shp` serve the second purpose; they bind the nesting type and nesting shape to the identifiers `b` and `shp`, respectively. For the function signature they are redundant: the nesting type and the nesting shape are part of the nested array-type and are encoded within the type variable `a`.

Figure 9 shows another example of a function that inductively extends over array nestings: a generic definition of the arithmetic function `*`. In general, the `*` operation expects two arguments of the same type. We encode this in our syntax by using the same type variables for the generic types of both arguments. This ensures that the given instance can only be applied to arguments of the same type. Furthermore, we use a generic return-type in the definition of function `*`. Other than the function `shape`, whose result is, regardless of its argument's type, of type `int[.]`, the type of the result of the function `*` equals its arguments' type. Again, we express this type equality by using the same type variables in the return type as in the argument types. However, for return types we only permit the use of type variables that occur at least in one type of the arguments. This is to ensure that the result type of a function is uniquely determined by the type of its arguments.

As the result of function `*` is a homogeneously nested array, we use the dual to the de-nesting of arguments in line 7. The `type_enclose` operation nests the type of the result array with the nesting constructor. This illustrates how type variables bound by generic function signatures may be used in function bodies.

Since the application of function `*` in line 6 within the function body is dispatched for the de-nested type, it is not a recursive call of the given instance of `*`. Instead, the `*` operator is inductively extended across array nestings. The above definition of `*` is fully generic, in the sense that it defines multiplication for all nested array-types, regardless of their inner nesting-structure. Furthermore, it defines multiplication on multiply-nested arrays, *e.g.* arrays of tuples of complex values. As the nesting structure is finite, the recursion contained in the definition of `*` terminates with a call to an instance of the `*` function on a built-in type, as defined in Section 2.

5. Overloading on Nested Array Types

The generic definition of `*` yields the correct result, as long as the semantics of the multiplication operation is the same for both, the nested and de-nested 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. However, this would force the programmer 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 an error-prone task and implicitly distributes structural information throughout the program.

Instead of defining multiple functions for different nested array-types, we adopt a similar approach as we use 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.

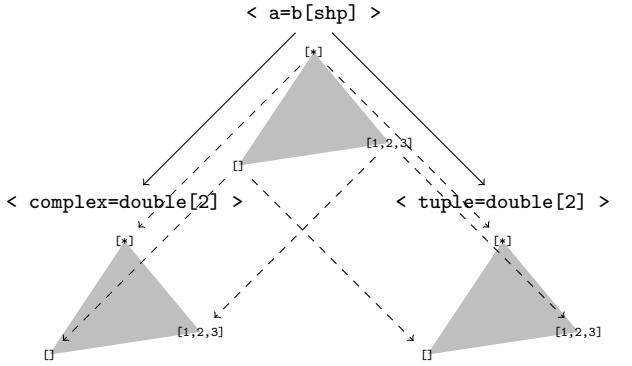

Figure 10. Extended SAC subtyping hierarchy.

Quite naturally, every concrete nesting type is a subtype of the generic nesting type 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 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 10 gives a schematic overview of this extended subtype hierarchy. The grey triangles indicate the shape subtyping hierarchy, whereas the structural subtyping hierarchy is illustrated for two example 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.

```
1 <a=b[shp]>[.] (+)( <a=b[shp]>[.] A, <a=b[shp]>[.] B)
2 {
3   return( { iv -> A[iv] + B[iv] } );
4 }

5 <complex=double[2]>[*] (+)( <complex=double[2]>[*] A,
6                             <complex=double[2]>[*] B)
7 {
8   return( { iv -> A[iv] + B[iv] } );
9 }
```

Figure 11. Example for ambiguity of dispatch on the combined subtype relation.

Given this two-dimensional subtype relation, the notion of least supertype needed for function dispatch is not uniquely defined anymore. As an example consider the two instances of + given in Figure 11. Both define addition on nested arrays. The first instance (cf. lines 1ff.) is defined on arguments of type `<a=b[shp]>[.]`, *i.e.*, vectors of nested arrays. The definition of the second instance (cf . lines 5ff.) uses `<complex=double[2]>[*]` as argument type,

i.e., it defines a + operation for arrays of complex values. Although both instances comply with the covariance restriction introduced in Section 2, function dispatch may be ambiguous. When applied to arguments of type `<complex=double[2]>[.]`, both instances of the overloaded function + match equally well. For the first instance, the shape component of the actual argument type matches exactly, whereas the nesting component `<a=b[shp]>` is a supertype of `<complex=double[2]>`. The second instance matches the nesting component of the actual argument type, but the shape component `[*]` is a supertype of `[.]`.

To disambiguate the dispatch decision in these cases, we give precedence to the shape subtyping over the structural 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. Using this precedence rule, the application of function +, as defined in Figure 11, to arguments of type `<complex=double[2]>[.]` is dispatched to the second instance.

Our decision to give precedence to the shape subtyping is guided by the intuition that nesting-type specific instances are usually defined only if the semantics of the function differs for the specific type from the one implemented by the generic instance. This, for example, is the case for the type `complex` introduced earlier. For shape-specific instances, the intuition is the other way round. Here, one need not specify instances on a more specific shape as they have the same semantics as the more shape-generic version.

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

5  <complex=double[2]>[] (*)( <complex=double[2]>[] A,
6                             <complex=double[2]>[] B)
7  {
8    A' = type_disclose( A);
9    B' = type_disclose( B);

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

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

14   return( result');
15 }
```

Figure 12. Definition of complex multiplication.

Given this extended subtype relation, we can now overload functions on the shape and structure of their arguments. Figure 12 demonstrates the expressive power that comes with this dual overloading by means of the $*$ example extended to arrays of both complex numbers and tuples. In conjunction with the code in Figures 3 and 9 we have four overloaded instances of the function $*$:

1. a generic instance on nested arrays of arbitrary shape,
2. a generic instance on a nested array of scalar outer shape,
3. a non-generic instance on complex numbers and
4. a non-generic instance on double values.

As an example, consider the dispatch of application of * to two vectors of complex numbers. The initial dispatch chooses the first, most general instance, which maps multiplication to the level of individual complex numbers. Here, the dispatch takes the specialised instance for complex numbers, that in turn maps multiplication to the level of double values, which is taken care of by the fourth instance.

As an alternative example, consider multiplication of two matrices of tuples. Once again, we start with the first instance, which maps the computation to the level of individual tuples. In contrast to the example of complex vectors, there is no specialised multiplication on tuples and, hence, we dispatch to the generic instance (instance 2), which by mapping multiplication again to the element level of the tuple, perfectly suits our needs. Eventually, we again employ the fourth instance of function * to do the numerical computation.

As all nested array types are named in SAC, the specific instance for complex values does not affect the generic definition for arrays of tuples of double values. Although the structure of type `< tuple=double[2] >` is equivalent to the structure of type `< complex=double[2] >`, once an application of * to two vectors of tuples of double values has been mapped down to scalar level, it is dispatched to the generic instance for scalar values and thereby transparently mapped onto the inner dimension. The programmer is thus fully liberated from the task to choose the correct instance of a function depending on the underlying semantics of a given nested array.

6. Putting it all together

We now give a generic definition of a dot product function on nested arrays to demonstrate the interplay of nested array-types, generic function definitions on the nesting structure and function overloading.

```
1  module ArrayGenerics;

2  import Array : all;

3  export all;

4  int[.] shape( <a->b[shp]>[*] A)
5  {
6    return( drop( - [len( shp)], shape( A)));
7  }

8  <a<-b[shp]>[] (*)( <a->b[shp]>[] A, <a->b[shp]>[] B)
9  {
10   return( A * B);
11 }

12 <a=b[shp]>[*] (*)( <a=b[shp]>[*] A, <a=b[shp]>[*] B)
13 {
14   return( { iv -> A[iv] * B[iv] } );
15 }

16 <a=b[shp]>[] dotproduct( <a=b[shp]>[.] A,
17                          <a=b[shp]>[.] B)
18 {
19   return( sum( A * B));
20 }
```

Figure 13. Generic definition of basic array operations on homogeneously nested arrays.

Figure 13 shows the definition of a SAC module `ArrayGenerics` that provides the basic generic definitions for nested array-types. The scalar operations on built-in types are imported from module `Array` using an `import` statement in line 2. Most functions shown in Figure 13 have already been used in this paper. However, we use the opportunity to introduce a more concise notation. As applying `type_disclose` to the arguments is a common pattern, we use `< complex->double[2] >`

in argument-type position to implicitly convert the type of an argument to its de-nested counterpart within the function body. Dually, `< complex<-double[2] >` as return type is used as equivalent to applying the `type_enclose` function to the return values. The additional `dotproduct` function in line 16 defines the dot product of two arbitrary vectors of homogeneously nested arrays. The application of `sum` in line 19 computes the sum of all elements of a given vector.

```
1  module Complex;

2  import ArrayGenerics : all;

3  typedef double[2] complex;

4  export all;

5  <complex<-double[2]>[] (*)( <complex->double[2]>[] A,
6                              <complex->double[2]>[] B)
7  {
8    return( [ A[0]*B[0] - A[1]*B[1],
9              A[0]*B[1] + A[1]*B[0]]);
10 }
```

Figure 14. Definition of basic array operations on complex numbers.

The type `complex` and multiplication thereon can now be defined using the module `ArrayGenerics` from Figure 13, as shown in Figure 14. The `import` statement in line 2 makes all functions and generic definitions from module `ArrayGenerics` available in the current module. Thus, the generic definitions for `*`, `shape` and `dotproduct` become immediately available for type `< complex=double[2] >` defined in line 3. We further overload the `*` operation for the type `< complex=double[2] >[]` to represent the difference in semantics of the `*` operation on complex numbers.

As can be seen, the techniques presented in this paper simplify the introduction of new nested array-types. In most cases it suffices to use the generic instances. Specialised instances need to be defined only for those cases in which a specific behaviour is demanded. As a consequence, programmers may focus on the interesting parts of a program, leaving all boiler-plate code to be generated through generic instance specifications.

Given the definition of module `Complex` in Figure 14, we can apply basic arithmetic functions to the nested array-type `<complex=double[2]>` just as well as to basic datatypes. As an example, consider the application of the `dotproduct` function to two vectors of complex numbers. As the arguments of the application are vectors, the application of `*` within `dotproduct` is dispatched to the generic array instance, which maps the application to scalar level. Since we defined an explicit instance of `*` 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 computes the product of the two elements. Finally, the application of `sum` within the function `dotproduct` (cf. line 19 in Figure 13) computes the result.

7. Related Work

In SHARP APL [BB93], nested arrays are handled using explicit applications of `enclose` and `disclose` functions. The function `enclose` boxes an array, whereas its dual operation `disclose` unboxes it. J [HI04] makes use of two functions `box` and `open` with equivalent semantics. Similar functions are present in APL2 [Ben91] in the form of `mix` and `split`. These functions permit an array to be split in between two dimensions, *e.g.*, a matrix can be split up into a vector of vectors.

The array-language NIAL [JJ93] was explicitly designed with nested arrays in mind. In contrast to the approach presented in this paper, the nesting structure of an array in NIAL has to be explicitly defined by the programmer using a nesting vector. Furthermore, it is the programmer's responsibility to explicitly state the mapping of each function application to the nesting level it is intended to operate on.

A similar approach is taken by FISH [Jay98]: the nesting is part of the shape vector on which nesting and de-nesting operations, similar to what we presented in this paper, can be defined. As shapes are fully static in FISH and cannot be altered for a given array other than by explicitly copying the elements into an array of the new shape, the uses of these nesting and de-nesting operations are limited. As a further difference to the approach presented here, the nesting levels are not named and thus cannot be exploited for function overloading. The programmer has the responsibility to choose the appropriate operations depending on the semantics of the data handled.

For all APL-like languages, as with nesting vectors in NIAL and FISH, the programmer has to be aware of the nesting structure and has to manually insert the appropriate nesting and de-nesting operations to extend function applications across array nestings. As the nesting operations are operations on arrays, they are scattered throughout the code. Besides the additional work and complexity during the initial development of a program, the distributed nature of data-structure definitions complicates the maintenance and refactoring of existing code: Every change in the nesting structure forces the programmer to adapt the function definitions to that particular nesting structure.

For functional languages like HASKELL [Pey03] and CLEAN [PvE01], generic programming extensions for algebraic datatypes have been proposed [JJ97, Hin00, AP02]. These extensions provide support for defining generic instances inductively on the three basic data constructors of algebraic datatypes: unit, product and sum. Combining generic and specific instances to one overloading function, *e.g.*, by using subtyping as in our approach, is not supported. To our best knowledge, these approaches have not been applied to forms of datatypes other than algebraic datatypes. Specifically, we are not aware of any datatype-generic programming approach that uses explicit array type-constructors.

8. Conclusion

We have presented a structure-generic programming extension for SAC that eases the use of nested arrays by reducing the amount of boiler-plate code that is needed to introduce nested array types. This is achieved by lifting the nesting information from the array level to the type level. By exploiting this type structure for inductive function definitions, we enable the programmer to specify generic functions for arbitrarily nested arrays. Moreover, we gave an example of how this neatly integrates with the function overloading capabilities of SAC and its existing shape-generic programming model. By combining these two approaches, homogeneously nested arrays can be used to model data structures like arrays of complex numbers without loosing the ability to write generic algorithms on n-dimensional arrays. Furthermore, as the nesting structure is encoded in the type, the programmer is liberated from explicitly encoding the nesting structure in the program code.

Our approach combines shape-generic and structure-generic programming on arrays. However, the way we introduce generic programming, namely the restriction to homogeneously nested arrays, allows us to use our existing optimisation techniques, in particular with-loop scalarisation [GST04], to compile highly generic specifications on nested arrays into efficiently executable non-

generic code that operates on flat arrays at runtime. Our approach of using inductive function definitions on the nesting structure of arrays is not limited to homogeneously nested arrays. Extending the optimisations and the type system of SAC to non-homogeneously nested arrays remains as future work.

Acknowledgments

This work was funded by the European Union ÆTHER project (cf. http://www.aether-ist.org).

References

[AP02] A. Alimarine and R. 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] R. Bernecky and P. Berry. *SHARP APL Reference Manual*. Iverson Software Inc., 2nd edition, 1993.

[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.

[BML97] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for java. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–145, New York, NY, USA, 1997. ACM Press.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification*. Java series. Prentice Hall PTR, third edition, 2005.

[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.

[HI04] R.K.W. Hui and K.E. Iverson. *J Introduction and Dictionary*. Jsoftware Inc., 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.

[Int93] International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.

[Int03] International Standards Organization. *ISO/IEC 23270:2003: Information technology — C# Language Specification*. International Standards Organization, 2003.

[Jay98] C. B. Jay. The FISh language definition. Technical report, University of Technology, Sydney, 10 1998.

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

[JJ97] P. Jansson and J. 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.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159, New York, NY, USA, 1997. ACM Press.

[Pey03] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PvE01] R. Plasmeijer and M. van Eekelen. *Concurrent Clean 1.3.1 Language Report*. High Level Software Tools B.V. and University of Nijmegen, 2001.

[Sch99] S.-B. Scholz. On Defining Application-Specific High-Level Operations by Means of Shape-Invariant Programming Facilities. *SIGAPL Quote Quad*, 29(3):32–39, 1999.

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

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.