A symmetry-based formalism for array subtyping

A Shafarenko
Department of Computer Science
University of Hertfordshire, Hatfield, England
e-mail: a.shafarenko@herst.ac.uk

Abstract

This paper presents an array algebra based on the concept of symmetry. The symmetries taken into account are translational, affine and polyhedral symmetry as well as access symmetry inherent in distributed arrays. The full set of data-parallel array operations is represented in terms of 4 fundamental skeletons: Map, Juxtapose, Select and Concatenate which are strongly typed and overloaded for all combinations of operand symmetries. A hierarchy of three further skeletons are used as parameters to Map in order to express reductions: commutative-associative, associative but not commutative, and neither associative nor commutative. Any user-defined part of the computation is represented in the form of scalar functions as skeleton parameters, without any loss of generality. Successful type inference in the presence of subtyping is enabled by a homomorphism restriction imposed on all overloadings of the skeletons. As a result, we are able to infer the most symmetric type of any well-formed term, which is usually the cheapest one to compute.

1 Introduction

An important property of arrays is their uniformity. An array in the narrow sense is a set of elements *of the same type* distinguished by a multi-index, which usually ranges over the Cartesian product of integer intervals. The type uniformity of elements does not imply any uniformity of the element values, or locations or the cost of accessing them. Arrays per se are just type-uniform collections. This abstraction does not always correspond to the concrete arrays used in applications, where the properties (or values) of different elements are not entirely independent.

When an object is characterised by fewer parameters than it has degrees of freedom, it is often associated with some form of *symmetry*. Spatial symmetry, which is relevant to multidimensional arrays, manifests itself in geometry: for example, a square is a parallelogram with equal sides and is thus fully defined by one parameter, a cube is a particular type of prism, etc.Symmetry is gradeable: there can be more or less symmetry in an object, which gives rise to symmetry-based hierarchical classifications. For instance, an elipsoid (three parameters) can be axially symmetric about *x*, *y* or *z* axis (two parameters), or a sphere (one parameter), which is a subclass of them all. Spatial symmetry can also be manifest in properties other than shape, notably in array values which may depend on a combination of indices rather than each of them individually. In particular, users of APL[BPP88] are quite familiar with translational symmety of arrays, even though they may not use this paricular term. Indeed, one can easily construct array objects in APL which depend only on a subset of their indices.

This paper will report the results of our effort to take symmetry into account systematically throughout an array formalism for data-parallel (and to some extent, process-parallel) processing of arrays. It will define and explore high-order operators that use such symmetry. A type system with subtyping will be outlined, which is capable of determining the most symmetric type of an expression. This information can be used by a compiler to derive the data distribution and the execution schedule of an array-based code.

2 Restrictive subtyping

Since symmetry is usually gradable, there is always a classification of objects in the increasing order of symmetry (which can be partial to allow incommensurable symmetry types). We must reflect this classification in an *inclusion relation on types*(see [Rey85]), also known as subtyping, which has to be maintained by a type system throughout our formalism. Also, the explicit account of symmetry that has a complex Cartesian structure creates too many symmetry-specific versions of each operator. It is possible, however, to use *overloading* to reduce this number, provided that the type system is able to disambiguate any operation to its specific type using the context it occurs in. The following restriction is necessary to avoid unpredictable result types:

Definition 2.1 *(Homomorphism restriction) For an overloaded operator L, overloadings* $L^{[1]} :: a_1 \to b_1$ *and* $L^{[2]} :: a_2 \to b_2$ *are said to satisfy the* homomorphism restriction *when either*

1. $a_1 \sqcap a_2 = \bot$, *or*

2. $a_1 \sqcap a_2 = a_0 \neq \bot$, *in which case one of the pair* b_1,b_2 *must be included in the other and the following equation must hold for any value of* X *of type* a_0*:*

$$c_{b_2 \to b_1}\ (L^{[2]}\ (c_{a_0 \to a_2}\ X)) = L^{[1]}\ (c_{a_0 \to a_1}\ X),$$

if $b_2 \subseteq b_1$, *or*

$$c_{b_1 \to b_2}\ (L^{[1]}\ (c_{a_0 \to a_1}\ X)) = L^{[2]}\ (c_{a_0 \to a_2}\ X),$$

otherwise. (Here $\sqcap$ *denotes the greatest lower bound of two elements and* $\bot$ *is the only subtype of all possible types.)*

An operator is said to satisfy the homomorphism restriction if all its overloadings do pairwise. For any instance of the operator where more than one overloading is compatible with the type of the operand, the one with the least result type is chosen. If in fact a different one was intended, which has a senior result type, the context would expect that type of the result. In this case the compiler should insert the appropriate coercion automatically, which, due to the homomorphism restriction, would have the same effect as the intended overloading.

For the Cartesian product of types, we introduce a standard subsumption order whereby a junior tuple must be junior in all components. Due to the standard subsumption we will always obtain type lattices from products of linearly ordered, bounded subtypes. We shall therefore always assume that we can deal with individual components of the multitype separately: while introducing a subtyping in one component, exact matching of the other components can be assumed without loss of generality.

As far as type constructors are concerned, μ-recursion on array types is not allowed as it would introduce nesting. This means that for any multitype

```
DIMENSION Q(IL,JL,KL),V(KL)
DIMENSION W(JL,IL)
...
DO I=1,IL
  DO J=1,JL
     DO K=1,KL
       Q(I,J,K)=V(K)*W(J,I)
     END DO
  END DO
END DO
```

Figure 1: Example of a vectorisable loop nest

there is a non-array element type, or *el-type* component, which, although not an array, can be as complex and abstract as the type system can handle. In particular, it can be a disjoint union or a μ-abstraction if the subtyping logic can resolve the subtyping of such types. It can also be an arrow type introducing an array of functions. In the last case, however, another non-nesting restriction applies: each element function must return a scalar. Otherwise applying the array of functions to an argument would result in an array of generally incongruent arrays, i.e. a nested data structure. However, only the el-type has to be restricted this way: indeed, if a function is not an array element, its application can not produce a nested array (see next section).

3 Translational symmetry

Fig 1 shows an example of a vectorisable loop nest as it appears in Fortran. This example uses 3 arrays of different ranks in a treble loop nest. Consequently, some of the indexed variables will not depend on some of the loop indices, for example `V(K)` is not affected by `I`- or `J`-iterations and `W(J,I)` does not change with iterations in `K`.

Definition 3.1 *An m-*orientation *of a rank-R array A is an array object that, if indexed with $[i_0, i_1, \ldots, i_r]$, where r equals the length of the mask $|m|$, selects the element $[j_0, j_1, \ldots, j_R]$ of the array A, with the indices $[j_0, j_1, \ldots, j_R]$ drawn from $[i_0, i_1, \ldots, i_r]$ according to the Boolean mask m in order. The number of ones in a Boolean mask is called the* character *of the mask and is denoted $\Box m$. For any valid orientation of A, $\Box m = R$.*

Note that orientation introduces translational symmetry in each result dimension corresponding to a zero in the mask.

Using orientations instead of the original arrays one can bring the example in fig 1 to a common dimensionality and then drop the explicit iteration space altogether:

```
Q=[001]V * [110]W'
```

where the prime denotes matrix transposition. The notation here is syntactically similar to the “numbers in brackets” of APL 2[BPP88].

In a complete data-parallel formalism, it should be possible to apply a function to an array. Although we can limit our analysis to a function of a single argument (and use currying), the function must have a certain rank: since the rank is a component of the multi-type, a function must have a static type signature in the rank component. If a function is applied to an object of a rank higher than the one the function expects for the argument, this can only be interpreted as a data-parallel application of the function in the extra dimensions.

Definition 3.2 *An m-orientation of a function of a rank-r argument is a function that accepts an array argument of a higher rank $R = |m| > r$. It uses a subset of the argument indices, according to the mask m, with the rest of the indices appended to the index list of the function result.*

Note that *array of scalar functions of arrays* is as far as the type construction may proceed without introducing nesting in the model, which raises the question of what exactly gets orientated if orientation is applied to such an array. Is it the array itself or the arguments of all the functions that it has as elements? In fact, the former takes place while the latter can only be achieved by mapping the orientation operator onto the array, see section 7.

Although any valid orientation must use an explicit constant mask with the character equal to the rank, a function symbol can be overloaded to represent a member of a family of functions with different argument ranks. For example, denote as `sum` the function that computes the sum of the elements of its argument. There is a family of such functions, parametrised with the argument rank (this is the only way to introduce `sum` with a definite type signature). For example, the type system should have no difficulty in determining that the `sum` in `([011]sum) B` is two-dimensional, so this expression evaluates to a vector $v_i = \sum_{j,k} B_{ijk}$.

The fact that an array object has an additional spatial symmetry is very important in distributed parallel computing. It enables the compiler to map the array elements in such a way that series of identical values need not be distributed but can, if necessary, be obtained by broadcasting. It is of benefit therefore to make the information about orientation a component of the multitype by replacing the rank by a *rank mask*, which indicates by 1's the object axes along which the elements change and by 0's those ones that have translational symmetry. For example, if orientation 01 is applied to a vector, that vector becomes a matrix each row of which is a replica of the original vector. Obviously, the rank proper is equal to the length of the rank mask. Wherever symmetry is not important we shall continue to use numerical rank instead of the mask, assuming that any valid rank mask is acceptable.

The type inclusion relation for types of translational symmetry follows from the fact that the lack of symmetry along an axis is a more general case than its presence, taking into account that symmetries associated with different axes are independent.

Definition 3.3 *Let two objects x and y have different rank masks $\rho(x) \neq \rho(y)$, with the actual ranks being the same: $|\rho(x)| = |\rho(y)| = r$. Then the type inclusion relation $\rho(x) \subset \rho(y)$ is defined by the partial order $(\forall i : 1..r)\rho(x)_i \leq \rho(y)_i$, according to the standard subsumption.*

The list of object dimensions, also called the array *shape*, is not (can not be made) part of the multitype if geometric operations on arrays are are allowed; it is therefore part of the object value. It is represented as the shape vector $\mathbf{s} : N_+^r$, where $N_+ = N \cup \{\infty\}$, which at all times must satisfy the condition (the double arrow denotes implication):

$$(\forall i : 1..r)(\rho(x)_i = 1) \Rightarrow (s_i < \infty),$$

that is, all nonreplicated dimensions must be finite.

The orientation symbol used above for array orientation (a mask in square brackets) has the following "rank-mask signature":

$$(\forall a : |a| = \Box m)[m_1..m_n] : \{a_i\} \to \{b_j\},$$

where

$$b_j = \begin{cases} a_{\omega(m,j)}, & \text{if}\, m_j = 1 \\ 0, & \text{otherwise} \end{cases}.$$

Here $\omega(m, j) = \sum_{k=1}^{j} m_k$.

Rank masks of different lengths (i.e. corresponding to different dimensionalities) are incommensurable. One can, however, drop some, or all, of the translationally symmetric axes using the projection operator $\widehat{m}$ whose application to any sequence $\{a_i\}$

results in the sequence of elements $\{a_{j_i}\}$ corresponding to the ones in mask m, in the same order. If the projection operator is applied to an array, it effectively prefixes the sequence of indices to that array. Such an application is legal only if $|a| = |m|$ and $(\forall i : 1..|m|)(m_i = 0) \Rightarrow (a_i = 0)$, which means that the discarded indices have no effect on the element value.

It is convenient to define the action of orientation $[m]$ on a bit sequence, too, in addition to its effect on arrays: we shall assume that $[m]v$ for any bit sequence v denotes the result of replacing all 1's in the mask by the members of v, in order. For example, $[10110]101 = 10010$ and $[10110]010 = 00100$. This form of orientation will be required later in the analysis of affine symmetry.

For any array A and mask m,

$$\widehat{m}[m]A = A,$$

provided the orientation is valid.

Finally, we join all rank lattices together at the bottom, by making every scalar type a member of all ranks since this only introduces unambiguous upgrading coercions.

4 Individual access symmetry

Abstract parallelism of data can be described as the lack of interference between different elements of a nonscalar assignment so that the hardware *may* perform all elemental assignments at once. In practice, however, a distributed implementation would perform DP assignment in a certain order to minimise the communication and scheduling costs. In the simplest case of a rectangular processor array, data objects participating in the same DP operation will be co-mapped onto the array with a certain block size. Although scheduling of different blocks may be totally independent, within a block computing is strictly sequential.

To separate out objects with different symmetries the model requires an *a priori* access cost, which is an asymptotic ($N \gg 1$, with N being the object size) measure that guides the user in the choice of the correct access type. For the purposes of classification, the machine is modelled as a single array processor that retrieves arrays as wholes from shared memory into its internal storage and then performs operations. The cost estimates ignore local storage access overheads.

Definition 4.1 *The a priori access cost is a triplet* $(c_\tau, c_\alpha, c_\rho)$*, where* c_τ *is the cost of* **total access***, i.e. retrieval of all items of the arrangement, but not necessarily in order;* c_α *is the maximum cost of* **affine access***, i.e. an arrangement of array elements with the indices forming an arithmetic progression, and* c_ρ *is the maximum cost of* **random access***.*

Now we are in a position to introduce access subtypes, initially for a single dimension of an array, by giving upper bounds to the corresponding access costs.

Subtype	c_τ	c_α	c_ρ
locator	$O(N)$	$O(N)$	$O(N)$
collector	$O(1)$	$O(\log N)$	$O(\log N)$
sequencer	$O(1)$	$O(1)$	$O(\log N)$
director	$O(1)$	$O(1)$	$O(1)$
replicator	0	0	0

The intuition for a locator object is a localised arrangement of indexed items. Since no distribution across a parallel system is assumed, all the access costs are proportional to the number of elements. A collector is the simplest possible DP arrangement. The intuition behind it is that the object is well-distributed but not structured in any way. It should be possible to access all of it in unit time, but any given processor may receive an arbitrary subset of the indices. Such an arrangement is quite acceptable for operations such as reductions as long as those are based on a commutative, associative operation. Putting a collector into a given order will necessitate sorting, hence a higher cost of affine and random access.Sequencer is the most frequent access type. Its main purpose is to define array structures with locality, i.e. the ability to be shifted and/or decimated in place at negligible cost. Asymptotically the cost of any linearly ordered selection of elements is of order unity, which reflects the fact that only a small number of elements need to be exchanged between processors at a relative cost of $O(1/N)$ vanishing as $N \to \infty$. Arbitrary indexing, however, violates locality and hence still requires at least a parallel sort, which is reflected in the respective cost component. Director is obviously the least restricted access to an array distributed across a parallel processing system. Note that at present this is also the only intuition of cost the user of Fortran-90[For91] can be expected to have, as it is the one corresponding to the random-access model of Fortran and the natural meaning of synchronous data-parallelism.

The above definitions also define the chain of type inclusions $replicator \subset director \subset sequencer \subset collector \subset locator$, which is a linear order on types.

Definition 4.2 *The access type of a multidimensional object is the Cartesian product of per-axis types.*

The type inclusion relation between multidimensional access types is one of partial order: a subtype has to be junior in *all* dimensions of a supertype. Objects of different ranks have incommensurable access types. All access types with a common rank form a lattice.

The cost intuition based on the above hierarchy may suggest that one should use types that cost the least. This is not the case. The real objective is to maximise the access type consistent with a given usage of the object, as that reduces the importance of efficient data access. For example, if a piece of program does not require random access to a nonscalar object and only uses regular selections from it, the object should have the type sequencer. This would prompt a compiler to choose the block distribution mode. On the other hand, the type "director" may cause a pseudo-random distribution with a large communication cost.

5 Collective access symmetry

The previous section introduced access classification for a single nonscalar object. In real applications, a few nonscalar objects participate in an expression, with the data dependencies causing strong correlations between their patterns of access. For example, the most frequent occurrence of a matrix X could be as an operand to an element-wise operator that adds it to another matrix Y. Although each matrix may be implemented as an **ss**-type object which ensures a low access cost of $O(1)$, the executing agent would have to "zip" the elements of X and Y with matching indices since arithmetic operations can only be performed having both operands in the same place. In our simplified costing model, the cost expectancy of the zipping is at least $\log N$ since one of the operands has to be recast to be co-located with the other.

Considering a set of objects participating in a DP computation, one can introduce the relationship of alignment between individual axes belonging to different objects. The axes that are not aligned by the computation remain independent. This leads to the introduction of a joint alignment space (JAS), which can be treated as a pseudo-object having no content or shape, but which has a certain rank (i.e., the number of axes) and a definite access type. Then each real object is assumed to be aligned with the JAS in each of its dimensions. Those dimensions acquire the access type associated with the corresponding axes of the JAS.

The only language feature that is required to introduce alignment is therefore a declaration of access type synonyms via an explicit JAS. For example, the following is a definition (in some syntax) of 4 objects using the JAS `atmo` with three axis (ellipses denote irrelevant components of type):

```
syntype atmo=(s,s,d) in

        Elevation:        ac-
cess=(atmo.1, atmo.2) ...
        DurationOfWinter: ac-
cess=atmo.2 ...
        Temperature:      ac-
cess=(atmo.1,atmo.2,atmo.3) ...
        GravityForce:     ac-
cess=atmo.3 ...

        <... body of the block...>

end_syntype
```

Here objects `Elevation` and `Temperature` are aligned in the first two dimensions, the second axis of `Temperature` is aligned with the axis of `DurationOfWinter` and the second axis of `Elevation`, etc. In addition `atmo.1,2` are equivalent to sequencer and `atmo.3` is a synonym of director.

6 Data-parallel skeletons

In order for the hierarchy defined above to be useful, the operators acting on arrays should be given access types. That means that although without access types, any operator can be applied to any array (all the other type components permitting), as soon as a lattice of access types is introduced in every rank, an operator must be set at some node of this lattice that corresponds to its principal access type. If the operand(s) fall short of the principal type, appropriate coercions are inserted by the type system. Out of the set of possible overloadings, the one with the least type compatible with the types of the operands is used.

In the framework of the skeleton approach[Col89], the DP operators can be regarded as instances of a few high-order functions that depend on functional parameters. Let us denote as ${}^{r}x$ the type of an array which has rank r and el-type x. When a superscript follows a type variable, as in x^n, this denotes a product type, i.e. the type of all n-tuples of objects of type x. When we use both preceding and succeeding superscripts, this can be either an array of tuples or a tuple of arrays. To avoid any ambiguity, parentheses should be used in all such cases. Finally, wherever the access component of type must be specified, it will be denoted by a preceding subscript, so ${}^{2}_{\mathrm{sl}}t$ denotes the type of any 2d array with el-type t whose access types in the first and second dimensions are **s** and **l**, respectively. Note that **sl** in this example is, in fact, the Cartesian product of per-axis types (see def. 4.2), which makes it legal to use power notation, e.g. $\mathrm{c}^3 = \mathbf{ccc}$.

7 Map.

This is the fundamental skeleton of DP computing. It applies a pure function to an array element-wise and has the following type signature:

$$(\forall r > 0, a, b)\,({}^{0}a \to {}^{0}b) \to {}^{r}a \to {}^{r}b\,,$$

which introduces overloading in rank. For any function f, $Map\ f$ is indifferent to the access type of the argument: the access part of the signature is therefore fully decoupled from the rest and is given by

$$(\forall r > 0, x \subseteq \mathbf{l}^r)\, () \to x \to x\,,$$

where $\mathbf{l}$ is the locator access type and $()$ is the access type of a scalar.

A generic *Map* skeleton must also allow the function argument to accept arrays of any rank not exceeding the rank of the second argument of the *Map*. Therefore, a family of skeletons $\{Map_m\}$ is required, which is parametrised by an orientation mask m, with the following signature:

$$(\forall k = |m|; \forall a, b)\, ({}^{\square m}a \to {}^{0}b) \to {}^{k}a \to {}^{k-\square m}b\,.$$

Now let us define the (still disjoint) *access* type signature of Map_m:

$$(\forall y : (\widehat{m}y) \subseteq x)\, (x \to ())\to y \to (\widehat{\overline{m}}y)\,,$$

where the bar above m is the standard denotation of bit compliment, and the hat over the mask denotes the projection operator defined earlier. Note that the first argument, a function returning a scalar, is anti-monotonic in the access type of its argument.

The above access type signature assumes an important *principle of locality*: wherever a function application is element-wise along an axis of its argument, it conserves the access type of that axis. The justification of the locality principle is that it costs the same to obtain a value or any scalar function of it from memory.

Observe that according to the signature, the operand axes unaffected by the function application carry their access types through to the result, and therefore the result array becomes aligned with the operand automatically.

The functional parameter can be any function taking rank $\square m$ into rank 0 (this guarantees non-nesting). However, three important cases below structure the functional parameter further, down to the level of scalar user-defined functions, which can be regarded as operator-parameters, and hence be treated algebraically.

Computation. This is a case of applying the functional parameter to the nonscalar argument to compute a new array. If the rank of the functional parameter argument is 0 then it defines an ordinary unary operator, such as $(-)$; if the rank is 1 or higher, the meaning of the *Map* is one of a reduction. Consider the following subskeletons:

$$\begin{aligned}
\Gamma^l &: (\forall a, b)^0(a \to b \to a) \to {}^0a \to {}^1_{\mathbf{l}}b \to {}^0a\,,\\
\Gamma^s &: (\forall a)^0(a \to a \to a) \to {}^0a \to {}^1_{\mathbf{s}}a \to {}^0a\,,\\
\Gamma^c &: (\forall a, r)^0(a \to a \to a) \to {}^0a \to {}^r_{\mathbf{c}^r}a \to {}^0a\,.
\end{aligned}$$

The reader familiar with high-order functions will easily recognise the `foldr` type signature of Γ^s, which has the meaning of a reduction with any associative (but not necessarily commutative) operator typed $(a \to b \to a)$ and its identity value typed a. Due to noncommutativity, the access signature requires type sequencer for the last argument. If the reduction operator is commutative as well, Γ^c should be used instead, generally with an increase in parallelism. Γ^c is polymorphic in the rank of the last argument as its semantics is not sensitive to the array structure: indeed it uses the array argument as a bag. Now, for example, the `sum` operator can be defined thus:

$$\texttt{sum}_m A = Map_m(\Gamma^c\ (+)\ 0)A$$

where 0 is overloaded consistently with the plus. Finally Γ^l describes the most general reduction process that goes sequentially from index 0 to the maximum index, and which does not require its first argument to have any algebraic properties whatsoever.

Selection. This is a case of using the nonscalar argument of *Map* to provide some location information that the functional parameter can use to select a specific element from another array: such a function can always be represented as $\lambda x.(\Xi S\, f(x))$, with some numerical function f, some array S and the constant Ξ being the element selection function which returns the element of its first argument selected using the second argument as an index tuple.

Function Ξ must be polymorphic in its array argument (source) since it does not use the contents of individual elements. There is no need to distinguish different rank versions either, since the rank of the first argument determines the rest of the signature unambiguously. The resulting signature is as follows

$$\Xi :\ (\forall r > 0, A)\ {}^{r}A \to int^{r} \to {}^{0}A$$

It is tempting to express the whole variety of selections via Ξ overloadings. This however is not possible due to their insensitivity to the access type of the nonscalar index since only a scalar element of it is used at any given time. The only access type Ξ can assume is this:

$$\Xi :\ \mathbf{d}^{r} \to () \to ()$$

which demands direct access to the array argument along all axes. Since there are many element selections that are more complex than that, and at the same time are satisfied with more basic access to the array, such a primitive is unsatisfactory. We shall discuss selections as a separate skeleton in section 9.

Concluding this section, it should be noted that user-defined functions for *Map* need not be nonscalar, since a nonscalar functional parameter can always be expressed in terms of one of the constant skeletons Γ.

8 Juxtapose.

The purpose of this skeleton is to penetrate the array structure of each of its two arguments down to the individual elements and then collect the pairs of elements with the same multi-index into a new array of the conforming shape.

The type signature is as follows:

$$(\forall a, b)\ {}^{r}a \to {}^{r}b \to {}^{r}(a, b)\,,$$

and the result rank mask m is the bitwise "or" of the rank masks of the arguments: $m = m_a \vee m_b$.

Here we need to decide how to deal with the shape attributes of the arguments and result. On the one hand, the intuition of shape has always been close to that of a type attribute, i.e. one would rather require the exact congruence of the operand index spaces. On the other hand, any attempt to *make* the shape a type attribute (see, for example, [HM93]) fails as the use of any reasonably comprehensive array formalism renders shape conformity undecidable. Consequently, the shape of an array has to be part of its run time value. Instead of exact conformity, we are following the example of f-code [MSS93]: every extent of the result is the smaller of the respective extents of the operands. Similar to exact conformity of the operands, this "intersection" rule can be applied to a group of objects in any order.

The access type signature for juxtaposition is quite intricate. There are four principal overloadings for every pair of corresponding axes:

1. Aligned types. Any access type is acceptable, since using the access mechanism of either of the operands the system will locate both elements of the result pair at once. The result is aligned with the operand axes.

2. Unaligned types; the senior type is **l**. If the other argument is a locator as well, the result can be

aligned with either of the arguments at the same cost. Moreover, aligning the result with a different index space is not *a priori* cheaper. Therefore there is no natural choice of alignment for the case and the juxtaposition of two unaligned locator axes is undefined. Consequently the type system will assume two locator arrays to be aligned if it encounters their juxtaposition. If the other argument is not a locator, the system will perform a poll to gather the elements of the other axis and align them with the locator dimension. The result type is locator.

3. Unaligned types; the senior type is **c**. Since a collector has no intrinsic order, it can not be juxtaposed with another collector axis other than by sorting both in the same order, which is essentially the coercion of both operands to an aligned sequencer type. Such an action is similar to a downgrading coercion and so ought to be explicit. The same applies to juxtaposing a collector with a sequencer except that the sequencer argument need not be sorted. However, if the other argument is a director or replicator, the juxtaposition is valid. The result type is collector in either case, and the result alignment is with the collector operand.

4. Unaligned types; the senior type is junior to **c**. The result access type is the senior of the argument types and the result axis is aligned with that of the senior type argument if the types are different. If the types arre the same, the result has the same access type and is aligned with neither of the arguments, since different portions of the object could be aligned either way to reduce the cost.

Note that juxtaposition can be generalised to any *n* objects by applying the rules repeatedly and then flattening the result el-type tuple:

$$((\dots(a_1,a_2),a_3),\dots,a_n) \longmapsto (a_1,a_2,\dots,a_n)\,.$$

We shall denote such a juxtaposition by double square brackets:

$$[[A_1,A_2,\dots,A_n]]$$

9 Select.

There are two reasons for treating selections separately from the *Map* skeleton. Firstly, as was mentioned in section 7, they are sensitive to the access type of the array source. Secondly, a more complex subtyping structure is required for the nonscalar index argument, which combines the already encountered translational with yet another, affine, symmetry, which occurs in integer objects.

The type signature of the Select skeleton is as follows:

$$Sel :: (\forall r,d,x)^r x \to ({}^d I)^r \to {}^d x,$$

where ${}^d I$ is some rank-d index type defined below, which we shall assume to be a subtype of ${}^d int$. (Remember that the notation t^n is used for the nth power of type t in the Cartesian product sense, i.e. the type of n-tuples of type-x components.)

9.1 Affi ne integer type.

In this section we shall use the translational symmetry notation introduced in the end of section 3.

Definition 9.1 *The* purely affine *type ${}^d A$ is the type of all d-dimensional, integer arrays v whose elements satisfy the following formula*

$$v_{i_1 i_2 \dots i_d} = \sum_{k=1}^{d} a^{[k]} i_k + b$$

with some integer $a^{[k]}$ and b. (The superscript is in brackets to avoid any confusion with Cartesian powers of types)

For example, the vector $(3,5,7)$ is of type ${}^1A$, with $a^{[1]} = 2$ and $b = 3$, while the matrix $\begin{pmatrix} 2 & 20 \\ 5 & 23 \end{pmatrix}$ is of type ${}^2A$ with $a^{[1]} = 18$, $a^{[2]} = 3$ and $b = 2$. If an array is purely affine, the implementation should only store the coefficients of the affine form since the element values can always be re-evaluated from them. Since the number of coefficients required in all cases is very small, the affine dimensions assume the access type replicator.

An array may not have a purely affine type, with some of the dimensions still being purely affine. The importance of partially affine access to arrays has been acknowledged in a recent publication [LCB00], where several supercomputer applications with partial affinity were analysed. In our theory the most general case is described by the following expression:

$$v_{\mathbf{j}} = \sum_{k=1}^{n} a_{\mathbf{p}}^{[k]} i_k + b_{\mathbf{p}},$$

where $\mathbf{p} = \widehat{m}\mathbf{j}$, $i_k = \{\widehat{\overline{m}}\mathbf{j}\}_k$, for some mask m, and all the coefficients are of the same rank $\square m = |m| - n$.

Definition 9.2 *The* index *type ${}^dI$ is a type of a d-dimensional, integer array, all elements of which satisfy the above formula with some mask m, $|m| = d$ and rank-l coefficients (where $l = \square m$) $a^{[k]}$ and b. The general index type is fully defined by two Boolean masks:*

$$\tau = [m]\,\overline{\left(\bigvee_{k=1}^{\square m} \rho(a^{[k]})\right) \vee \rho(b)},$$

which indicates by 1's which dimensions have translational symmetry, and

$$\alpha = \overline{m},$$

showing which dimensions have affine symmetry.

$$\begin{tikzcd}
 & & c_{i,j} & & \\
 & b_i j + c_i \arrow[ur] & & a_j i + c_j \arrow[ul] & \\
c_i \arrow[ur] & & ai + bj + c \arrow[ul] \arrow[ur] & & c_j \arrow[ul] \\
 & ai + c \arrow[ul] \arrow[ur] & & bj + c \arrow[ul] \arrow[ur] & \\
 & & c \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Figure 2: 2d affine type classification.

The data constructor Υ for the general index type is parametrised with the mask α and accepts as the argument an $(l+1)$-tuple (where $l = \square\alpha$) of affine-form coefficients of equal rank:

$$\Upsilon\alpha[a^{[1]} : e_1, a^{[2]} : e_2, \ldots, a^{[l]} : e_l, b],$$

where the integer scalars $e_1..e_l$ define the dimensions of the result along the affine axes (The rest of the shape is determined by the minimum of the respective dimensions of the affine form coefficients, since, in effect, this is a case of juxtaposition).

Now we are well-equipped to define *affine subtyping* on type ${}^n\mathit{int}$. For a single dimension the type inclusion relation is as follows:

$$\text{ts} \subset \text{as} \subset \text{ns},$$

where "ts" stands for translational symmetry, "as" for affine symmetry and "ns" for no symmetry. As before, a multidimensional subtype must be junior or equal to a supertype in all dimensions. We exemplify the type lattice in fig 2, where the case $d = 2$ is displayed. Every formula placed at a node of the lattice defines the structure of all 2d integer array objects having the corresponding subtype, with the indices i and j ranging over their first and second dimensions and the coefficients a, b and c determining the value of respective (i, j)-elements of it. For example, if object A has the affine type defined by the structural formula $a_j i + b_j$, this means that the values of its elements at any given time satisfy the condition $(\exists a_j, b_j \forall i, j) A_{i,j} = a_j i + b_j$, with some integer coefficients a and b. This is a case of affine symmetry in dimension i and no symmetry in dimension j. For all the subtypes, the number of indices to a, b and c and their relations to the first and the second index of array A are static[1] and form the type of the affine object. It should be clear now in what sense the affine type defined, for example, by the structure $ai + c$ is a subtype of the one defined by $ai + bj + c$: the former is an instance of the latter with b statically known to be equal to zero. Similarly, type $ai + bj + c$ is an instance of $b_i j + c_i$ for which b_i is statically known not to depend on i and for which c_i is known to be a linear function of i, namely $ai + c$ with some (new) a and c. One can also say that the former type is a subtype of the latter since it has affine symmetry in dimension i, whereas the latter type has no symmetry in that dimension, with the type of other dimension being the same. Incidentally, the fact that the template $bj + c_i$ (as distinct from $b_i j + c_i$) does not exist is explained by the disjunctive structure of τ: since the presence of c_i makes the $i - axis$ nonsymmetric already, it would not matter if b_i did not depend on i.

The top element of the lattice in each rank represents an ordinary array in that rank, whereas the bottom element of the lattice is always a replicated integer scalar. The lattice contains orientated purely-affine types of all dimensions less than or equal to that of the top element, as well as general index types with partial affine symmetry.

The access type of an axis of affine symmetry is replicator, as, obviously, any virtual processor is in a position to compute any element of an arithmetic progression immediately given its step and starting values.

What is the intention of affine integer types? They compactify the variety of selection primitives which would otherwise have to be different functions. For example, slicing a diagonal off a matrix A requires a separate primitive in the standard paradigm, whereas it is in fact general indexing in the form $A(X_i, Y_i)$, where X and Y are *statically known* to be arithmetic progressions with some starting and step values. What if Y is an arbitrary integer vector? Can we still usefully exploit the fact that X remains a linear sequence? The answer is negative for the standard paradigm, as it would introduce yet another primitive (curved diagonal?). However, all that is really required is proper subtyping of index objects so that the variety of selections available to the programmer may include all possible affine symmetries rather than some particular cases that appear “practically important” at present. In the next section we shall propose a *single* primitive that in our type system can be overloaded for all ranks and affine types of index so that the compiler may statically know which affine version of selection is used. The type lattice of affine types will enable it to *approximate* the versions that it is not profitable to implement separately if the exploitation of the respective symmetry in the target architecture is impossible.

9.2 *Sel* skeleton.

This function takes as many other arguments as the rank of the first one, the source. The reason they are not juxtaposed (as one would expect the indices to the same element selection operation to be) is because we do not wish to coerce the nonscalar index tuple to a single affine type. Such a coercion would cause a loss of type information and, as a result, may lead to excessive generalisation. Nevertheless, as far as the result contents are concerned, these can be de-

[1]this information follows from the rank masks of the respective arrays

fined element-wise as follows:

$$(Sel\,Z\,X_0\,X_1\ldots X_n)_{\mathbf{k}} = (Map\,(\Xi Z)\,[[X_0, X_1, \ldots, X_n]])_{\mathbf{k}}$$

for any valid multi-index **k**.[2]

However, function *Sel*, unlike *Map*, can use the information about affine symmetries of the indices as well as the source argument access type to choose the most efficient *particular* selection. This is achieved by overloading *Sel* for any combination of τ and α of each index argument. If we restrict ourselves to a maximum of 3 dimensions, in the worst case *Sel* should receive three index objects, each being 3-dimensional at most; we have therefore $3 \times 3 = 9$ per-axis affine types, each being one of $\{\mathrm{ns}, \mathrm{as}, \mathrm{ts}\}$ in every dimension of the juxtaposition space, and a total of 3^9 affine overloadings of *Sel*. This number, though large, is finite, and can easily be reduced in any specific implementation since firstly, the individual indices are all acting in the same way so that an implementation could have a regular method of generating selection overloadings, and secondly, the affine lattice provides type approximations for any lesser system of overloadings should the full set turn out to be impractical.

The access type requirements for the source of the *Sel* function are very easy to establish. Indeed, if the index corresponding to an axis of the source has an affine dimension, the axis type can be as high as sequencer. This is because the implementation can always decompose the selection across the affine axis so that it becomes a set of regular SLICE-like operations which do not require more than the sequencer access type. Otherwise the source axis is required to be a director. *Sel* is obviously polymorphic in the access type of all indices.

How is the access type of the result defined? Generally speaking, the selection operation creates an object entirely different from the source one: it may have a different rank and may contain an arbitrary subset of the source elements in each dimension, possibly with repetitions. One can not apply the principle of locality to the result of a selection since the operation itself is essentially nonlocal and so every dimension of the result is associated with n collinear axes of the indices (n being the rank of the source), each having its own access type. A solution exists, however, which is as follows.

Denote as $\{w_k\}$ the access type tuple of $[[X_0, X_1, \ldots, X_n]]$. For any k, consider the following cases:

1. w_k is senior to type replicator. The respective result axis has the same type and alignment.

2. w_k is of type replicator. If the kth axis of each of the $X_0, X_1, \ldots, X_n$ is translationally symmetric, so is the result axis, and it has the same type and alignment. Else if all but one axis are such, with the remaining axis of an X_m being affine, then the result axis is aligned with the mth axis of the source. Otherwise, same as case 1.

In the general case the result of selection is aligned with the selection index to provide control over the result access type. However, when a purely affine selection is performed in any of the dimensions, it is possible to leave the resulting layers of the source in place, at a much smaller cost. The second case of the rule takes care of that additional symmetry.

How does *Sel* act on an affine integer object as the source? If this were not defined specifically, then the type system would have to upgrade the source to the general integer type, which would cause unnecessary expansion of the affine form into an array of values. However, even if a specific overloading for affine types is defined, such an expansion may still be necessary if the selection index does not agree in symmetry with the source. A result axis will be

[2]This is *not* how *Sel* should be implemented, see section 7; we only use Ξ to defi ne the value of the result

affine only if all the selection indices are affine or translationally symmetric in that dimension and provided that those axes that are affine belong to the indices that correspond to affine/translationally symmetric axes of the source. A result axis will be translationally symmetric if for every index axis collinear with it, that axis itself, or the object axis corresponding to the index that the axis belongs to, or both, are translationally symmetric. Otherwise the result axis is not symmetric. Formally, if τ_i^s and α_i^s are the masks of the affine symmetry of the source, τ_i^r and α_i^r the respective masks of the result, and $\tau_i^{[k]}$ and $\alpha_i^{[k]}$ of the kth index object,

$$\begin{aligned}\tau_i^r &= \bigwedge_{k=1}^{d} \tau_k^s \vee \tau_i^{[k]} \\ \alpha_i^r &= \bigwedge_{k=1}^{d} \tau_i^{[k]} \vee \left(\alpha_i^{[k]} \wedge (\tau_k^s \vee \alpha_k^s)\right) .\end{aligned}$$

It should be noted that the power of *Sel* surpasses all known non-nested DP selections so that they can be expressed via it straight away. Here are some examples:

1. A SLICE of a vector V is given by

   $$Sel\, V\, (\Upsilon 1[k : l, m]) ,$$

   where m is the start, k is the increment, and l is the new horizontal dimension.

2. The transposition of a matrix R:

   $$Sel\, R\, (\Upsilon 10[[1]1 : \dim_2(R), [1]0])$$

   $$(\Upsilon 01[[1]1 : \dim_1(R), [1]0]) ,$$

   which clearly shows the 1d-affine, 1d-translational symmetry of the operation. Here and below we use the notation $\dim_k$ for the kth dimension of an array.

10 Affine Boolean and Polyhedron Types

The power of affine type can be applied to non-scalar Boolean objects that are often used to mark up computational domains. Most DP programming languages have a *WHERE* construct specifying a Boolean guard for a DP loop. When the element of the guard corresponding to the current multi-index is true, the computation proceeds, otherwise no action is taken for this iteration. It is often the case that the true elements of the guard are enclosed in some linear set, such as a half-space bordering on a hyperplane, a convex polyhedron or a union of polihedra. If the guard is defined by a formula, a compiler can attempt to analyse it. Projections of the guard can then be made to guide local computations in each of the virtual processors. However, if the guard is a stored object, its origin (whether linear or not) is generally lost and so conservative assumptions have to be about it (i.e. no symmetry). In this section we shall attempt to offer an alternative by introducing the following types.

Definition 10.1 *An array is said to be of* affine Boolean *type if it can be represented at any time as the result of the element-wise comparison $I \geq 0$, with some purely affine integer I.*

An affine Boolean object represents a part of a multidimensional parallelepiped (or the whole space if the object is infinite) bounded by an arbitrary hyperplane. The other part of the object is represented as $-I-1 \geq 0$, hence the greater-or-equal comparison is sufficient for both parts.

The storage mechanism for this object is very compact as only the coefficients of the affine form (i.e., the value of I) are stored. Affine Boolean is a subtype of the Boolean array type. If the element values are required by a context expecting a general

Boolean object, they can be generated "on the fly" without referencing the shared memory. On the other hand, when an affine Boolean is used as a guard, a compiler may incorporate the affine form computation with the body of the loop directly.

The access type of an affine Boolean object is replicator for obvious reasons.

Definition 10.2 *An object is said to be of* polyhedron *type if it can be represented at any time as an elementwise disjunctive form (i.e. the disjunction of conjunctive groups) of affine Boolean objects.*

The geometric meaning of the conjunction of two Boolean objects is the intersection of the true-value sets. When the operands are affine Boolean, the intersection of the half-spaces they represent is taken. For finite arrays, multiple conjunctions will result in a convex polyhedron, hence the name of the type. Similarly, a disjunction would deliver the union of the operands. Therefore the most general geometric interpretation of this type is the union of polyhedra.

If a d-dimensional object is of polyhedron type, it is represented as a $d+2$-dimensional array. Dimension 1 is associated with the conjunctive terms and dimension 2 with group disjunctions. Both of them are infinite. The rest of the dimensions may or may not be infinite. They represent the actual indices of the array. The access type of all dimensions is replicator (similarly to the affine form coefficients, there should not be too many members of the disjunctive form).

When a polyhedral object is initialised, assignments in the form $P(k,m) := I_{km} \geq 0$ are made with purely affine integerI_{km} representing the hyperplane k in the convex component m. A hyperplane can then be modified, (or removed, by assigning $P(k,m) = true$). A convex component is activated by placing a hyperplane in it; more hyperplanes can be added, repmoved or modified at any time. The whole convex component m can be wiped out by assigning $P(k,m) = false$ with any value of k. A moment's thought is enough to see that these rules are not artificial but ones following directly from the definition of the polyhedral type and from the inclusion of scalars in all rank latices. Thus an elegant mechanism of managing polyhedral domains is offered directly by the type system, without extending the language with appropriate commands.

Naturally, in implementation no assignment to elements of infinite axes ever takes place. Instead an appropriate reference structure is maintained whereby hyperplanes and convex components are linked and unlinked under the control of the right-hand-side expressions. However, to the user it all looks like ordinary assignments of array elements.

The type inclusion rules for Booleans is as follows. Two objects of polyhedron type are commensurable if and only if they are polyhedral in the same axes, in which case type inclusion is defined by the types associated with the rest of the axes. Any polyhedron is a subtype of the same rank Boolean.

Given these typing rules, the operations $\vee$ (union/disjunction), $\wedge$ (intersection/conjunction) and **not** (invertion/negation) are closed in polyhedra.

11 Concatenate.

The DP version of concatenation differs from the ordinary variety in two ways. Firstly, objects to be concatenated may have axes orthogonal to the axis of concatenation, which have to be brought to a common type, and secondly, there are access type constraints on the argument axes that participate in concatenation.

The respective dimensions of the arguments that are orthogonal to the concatenation axis must be juxtaposed and so the attributes of the result in this subspace are determined by the juxtaposition. This requirement follows from the fact that any layer of the result object that is orthogonal to the juxtaposition axis must be of the same type, and that juxtaposition is indeed the process of bringing collinear axes to a common access type. Without loss of generality, we can now limit our analysis to the case of 1d arguments to concatenation.

First consider the el-type. One can concatenate objects coercible to a single el-type, with the operation of concatenation not changing the rank:

$$Cat :: (\forall a, b) a \to b \to (a \sqcup b)\,,$$

where the symbol $\sqcup$ denotes the least upper bound of two lattice elements. Types a and b must belong to the same type lattice.

As far as the access type is concerned, concatenation applies to type sequencer and above as it itself requires a relatively weak access mechanism and destroys the regularity of direct access distribution. The natural choice of alignment is with the first argument, since this allows one to build chains of concatenated objects assuming that the operation of concatenation is left-associative. The reason why the replicator overloading is not supported is because it is mainly used for translationally or affinely symmetric axes, which would lose this type of symmetry after concatenation. The access type signature is therefore as follows:

$$Cat :: (\forall x \supseteq \mathbf{s}, y \subseteq x) x \to y \to x$$

12 Conclusions

A type system based on analysis of symmetries inherent in distributed DP computing has been introduced and the fundamental DP skeletons have been typed accordingly. It has been shown how the variety of all non-nested DP primitives can be reduced to instances of *Map*, [[]], *Sel* and *Cat* with only scalar functional parameters. While these skeletal constructs are not to be offered at the user level, they could provide a formal basis for optimisations in an array language compiler.

References

[BPP88] J A Brown, S Pakin, and R P Polivka. *APL2 at a glance*. Prentice Hall, Englewood Cliffs, N.J. 07632, 1988.

[Col89] M I Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[For91] *Fortran 90 Standard*, 1991.

[HM93] G Hains and L M R Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238–245, 1993.

[LCB00] A L Lain, D R Chakrabarti, and P Banerjee. Compiler and run-time support for exploiting regularity withing irregular applications. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):119–135, February 2000.

[MSS93] V B Muchnick, A V Shafarenko, and C D Sutton. F-code and its implementation: a portable software platform for data parallelism. *The Computer Journal*, 36(8):712–721, 1993.

[Rey85] John C Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development. LNCS vol 185.*, pages 97–138. Springer-Verlag, 1985.