Adding dependent types to X10

Vijay Saraswat
IBM TJ Watson Research Center
PO Box 704, Yorktown Heights
NY 10598
(Draft Version 0.2)
(Please do not cite)
(Send comments to `vsaraswa@us.ibnm.com`.)

24 December 2004

Abstract

X10 is a new modern statically typed object-oriented (OO) language designed for high productivity in the high performance computing (HPC) domain. X10, like most OO languages is designed around the notion of objects, as instances of *classes*. However, X10 places equal emphasis on *arrays*, a central data-structure in High Performance Computing. In particular, X10 supports dense, distributed multi-dimensional arrays of value and reference types, built over index sets known as *regions*, and mappings from index sets to places, known as *distributions*. X10 supports a rich algebra of operations over regions, distributions and arrays.

In designing a static type system for X10 a central problem arises. It becomes necessary to permit types, such as `region(2)`, the type of all 2-dimensional regions, `int[5]`, the type of all arrays of `int` of length 5, and `int[region(2)]`, the type of all arrays over two dimensional regions. The underlying general idea is that of *dependent types* – types that are parametrized by *values*, just as generic types are types paramterized by other types.

In this paper we develop a general syntactic and semantic framework for user-defined dependent types in the context of modern class-based OO languages such as Java, C# and X10. The design supports the definition of dependent and generic interfaces, classes and methods. As in staged languages, it distinguishes between compile-time evaluation and run-time evaluation. Dependent types are checked at compile-time; expressions occurring in them are perforce evaluated at compile-time. We outline the design for a compiler which can use an extensible constraint-solver for type-checking. No extension of an underlying Virtual Machine is necessary.

1 Introduction

1.1 The basic idea

Our basic idea about the introduction of dependent types in class-based statically typed OO languages is as follows. Broadly, we follow the spirit of generic types, but use values instead of types.

We permit the definition of a class `C` to specify a *parameter list*, a list of typed parameters `(T1 x1, ..., Tk xk)` similar in syntactic structure to a method argument list. Each parameter in this list is treated as a `public final` instance field of the class. We also permit the specification of a *where clause* in the class definition. A where clause is a boolean expression on the parameters separated from the parameter list with a "`:`". All instances of the class created at runtime must satisfy the where clause associated with the class.

It is now straightforward to see what a parametric type should be. It should simply by the name of a class (the *base class* of the type) followed by a where clause (a boolean expression on the parameters of that class), called the *condition* of the parametric type.[1] The denotation (semantic interpretation) of such a type is the set of all instances of subtypes of the base class whose parameters satisfy the where clause. Clearly, for the denotation of a parametric type t to be non-empty it must be the case that the condition of the parametric type is consistent with the where clause (if any) of the base class of t. We shall require that the type of any variable declaration must be non-empty.[2] For simplicity we shall permit the syntax `C(t1,..., tk)` for the type `C (:x1 = t1 && ...&& xk = tk)` (assuming that the parameter list for `C` specifies the `k` parameters `x1,..., xk`).

Parametric types naturally come equipped with a *subtyping structure*: type t_1 is a subtype of t_2 if the denotation of t_1 is a subset of t_2. Clearly this definition satisfies Liskov's Substitution Principle.

We will permit constructors and methods to specify preconditions as where clauses on parameters. For an invocation of a method (or constructor) to be type-correct it must be the case that the associated where clause is statically known to be satisfied.

We will permit *dependent methods* (similar to generic methods): a method may specify a list of parameters in the same way that a class may. The parameters may be used to construct dependent types for the arguments and return value of the method.

We will permit a type to be simultaneously a dependent and a generic type. For instance, we will permit the type:

```
List(3)<region(1)>
```

[1]By insisting that the condition reference only parameters, we ensure that whether an object belongs to a parametric type or not does not depend on the state of the object. By explicitly marking final fields as parameters we ensure that only those fields can be used to construct parametric types which were intended by the programmer to be used in types. An "accidentally" final field cannot be used to construct a parametric type.

[2]Types that are not parametric are always non-empty, assuming that there is at least one invocation of a constructor for the base class which does not throw an exception.

This is the type of all lists with `3` elements, each of which is a 1-dimensional region.

We will permit *static parameters*. These are static final fields of classes marked as parameters. Some static parameters are *configuration* parameters: they may be directly assigned a value from the invocation environment for a program (e.g. via a command line or a config file).

In order to facilitate compile-time type-checking we shall require that all expressions occurring in parametric types must reference only constants or (instance or static) parameters visible at that type, and involve methods that have been specially marked as parametric. This is necessary because it must be possible for the compiler to reason about the execution of these methods (e.g. by executing them symbolically). Further details are discussed below.

Rest of this paper In the next section I flesh out the syntactic details of the current proposal and consider a small example, `List(n)`. After that I present the code for `region`, `point`, `distribution` and `arrays` in X10. Most of the methods are `extern` methods (defined natively); however their signature shows how considerable information about the structure of the datatypes and the operations on them can be expressed through the type system.

2 Dependent Types

A dependent type is a type that takes values as arguments; such a value is called a *parameter*. Thus a dependent type is just like a generic type except that the arguments to the type are values, not types. Syntactically, these values are enclosed in parentheses, rather than in angle brackets. For instance:

```
List(10)<Cell> // The type of all lists of Cells of length 10.
```

The primary mechanism for the introduction of dependent types in X10 is the *dependent class declaration*.

2.1 Dependent class definitions

Dependent classes are specified by providing a parameter list (similar in structure to a method definition) right after the name of the class. This list is said to define the *explicit parameters* of the class. Each parameter introduces a `final` instance field of the same name and type in the class; hence parameter names must be distinct from field names. The list of parameters may be followed by a "`:`" (read as "where") and a boolean expression involving the explicit parameters (called the *where clause*). An instance of a class can only be created for actual values for the explicit parameter which can statically be known to satisfy the where clause.

Example 2.1 (List) *Consider the class* `List`*:*

```
public value class List( int n : n >= 0) <Node> {
  nullable Node node = null;
  nullable List(n-1)<Node> rest = null;

  /** Returns the empty list. Defined only when the parameter n
      has the value 0. Invocation: new List(0)<Node>().
   */
  public List (: n=0) () {}

  /** Returns a list of length 1 containing the given node.
      Invocation: new List(1)<Node>( node ).
   */
  public List (: n=1) (Node node) {
    this.node = node;
    this.rest = new List(0)<Node>();
  }
  public List (: n >= 2) {Node node, List(n-1)<Node> rest) {
      this.node = node;
      this.rest = rest;
  }
  public (int m : m >= 0) List(n+m)<Node> append(List<m>Node l) {
    return
        (n == 0) ? l : new List(n+m)<Node>(node, rest.append(m)(l));
  }
}
```

Intuitively, this definition states that a `List` *object can only be created when supplied with an* `int` *parameter* `n` *(intended to represent the length of the list).* `n` *is required to be non-negative. The class has two fields intended to hold the data item in the head of the list and the rest of the list. The nullary constructor is defined only if the value of the parameter is* `0`*. The unary constructor is defined only if the value of the parameter is* `1`*. Similarly for the binary constructor.*

The example also illustrates dependent methods*.* `append` *is a dependent method that has an* `int` *parameter* `m`*; it is required that* `m` *be non-negative.* `append` *returns a value of type* `List(n+m)<Node>` *and takes a* `List<m>Node` *argument. The parameter* `m` *is passed to the recursive call to* `append`*.*

A dependent class may also specify additional parameters in the body of the class. Such parameters are called *implicit* parameters. Syntactically, an implicit parameter is specified using instance field declaration syntax and prefixing the field declaration with the keyword `parameter`. The declaration of parameters must include an initializer which may depend only on parameters defined in the parameter list for the class or on previously defined parameters in the class body.

In what follows by the *parameters* of a (dependent) class we will mean either the explicit parameters enumerated in the parameter list, or the implicit parameters defined using a `parameter` declaration in the body of the class. Parameters may be used in an expression specifying the value of some other parameter (in a constructor or method call).

Example 2.2 *The class* `List` *may be extended to define an implicit* `boolean` *parameter* `isEven` *which records whether the length of the list is even.*

```
public value class List(int n : n >= 0) <Node> {
  parameter boolean isEven = (n % 2 = 0);
  ...
}
```

The parameters of a class may be used in any type expression in the body of the class at which they are visible. Parameters have the status of final instance fields, hence they are not visible in static fields, initializers or methods, or when they are overridden.

Creating instances of a dependent class. An instance of a dependent class is created using a `new` operation and passing an actual list of parameters (of the right type) after the class name and before the generic type parameters (if any). For instance:

```
new List(0)<Node>();
new String(10) (``abcdefghij'');
```

Dependent constructors. Sometimes a constructor for class `C` can only be invoked if the parameters of the class satisfy a certain condition. For instance, a nullary constructor for `List` is only defined when the parameter `n` has the value `0`.

Such conditions may be specified in a constructor declaration between the name of the class and the argument list of the constructor using a where clause. The where clause can reference only the parameters of the class.

Consider the definition of the nullary constructor for instance:

```
  /** Returns the empty list. Defined only when the parameter n
      has the value 0. Invocation: new List(0)<Node>().
   */
  public List (: n=0) () {}
```

Any attempt to invoke this constructor with a parameter that cannot be determined at compile-time to equal `0` will cause a compiler error.

Dependent methods. Instance method declarations may specify parameters after the qualifiers and before the return type of the method, through a parameter list (called the parameter list of the method). The parameter list may be followed by a where expression on the parameters of the method and/or the parameters of the class. (Recall that a where expression is a boolean expression following a "`:`".)

The parameter list of the method is visible in the declaration of the method and the body of the method. Thus the types of arguments to the method, and the return type of the method, may be built from type expressions referencing these parameters.

For example, consider the definition of the `append` method on lists:

```
  public (int m : m >= 0) List(n+m)<Node> append(List<m>Node l) {
    return
        (n == 0) ? l : new List(n+m)<Node>(node, rest.append(m)(l));
  }
```

The method specifies an `int` parameter, `m`. This parameter is referenced in the return type of the method `List(n+m)<Node>` and the argument type of the method `List<m>Node`.

A method invocation must specify the parameters in an actual parameter list after the name of the method and before the arguments to the method, e.g:

```
rest.append(m)(l)
```

For each such method invocation, the compiler must determine the corresponding method definition and statically check that its where clause (if any) is satisfied by the invocation.

MetaNote Need to determine rules for overriding for generic methods and compare with the rule above.

Extending dependent classes. *MetaNote: This should be standard. A class definition may extend a dependent super class, e.g. class Foo(int i) extends Fum(i*i) { ...}. The expressions in the actual parameter list for the super class may involve only the parameters of the class being defined. The intuition is that these parameters are analogous to explicit arguments that must be passed in every super-constructor invocation.*

2.2 Types defined by a Dependent Class definition

A dependent class definition introduces many types.

Let `C` be a class declared with explicit parameters `x1,..., xk` of type `T1,..., Tk`. A parametric type is an expression of the form `C (: e)` where `e` is a boolean expression in the parameters of the class. We permit the syntactic abbreviation `C(t1,..., tk)`, for values `t1,..., tk` of type `T1,..., Tk` respectively for the type `C(: x1=t1 && x2=t2...&& xk=tk)`. If `C` is a generic class with type parameters `<T1,..., Tm>`, then any type expression based on `C` must also specify the generic type parameters.

An object is an instance of a parametric type if its an instance of the base type, and the values of its parameters satisfy the boolean condition.

Example 2.3 *The expression* `List(3)<Node>` *is a type – the type of all lists of* `Node` *of length* `3`*.*

The expression `List (: isEven) <Node>` *is a type – the type of all lists of* `Node` *of even length.*

2.3 Dependent interfaces

3 Type-checking dependent classes

We first consider an example.

Example 3.1 (List, revisisted) *Consider the code in Example 2.1. In type-checking this code, the compiler must perform the following inferences. It must verify that* `null` *is a permissible value for the type* `nullable Node`*. It must also verify that* `null` *is a permissible value for the type* `nullable List(n-1)<Node>` *under the assumption that* `n >= 0`*. Both these inferences follow from the inference rule that* `null` *is a permissible value for the type* `nullable T` *for any* `T`.

To type-check the body of the second constructor, the compiler must establish that the value `0` *for* `n` *satisfies the condition* `n=0` *(the condition guarding the nullary constructor).*

To type-check the third constructor, the compiler must establish that the type `List(n-1)<Node>` *is non-empty, given that* `n >= 2`*.*[3] *To establish this, it must show that* `n >=2` *implies* `n-1 >= 0`*. The check for the body of the constructor is straightforward.*

To type-check the method, the compiler must infer that if if `m` *and* `n` *are non-negative, then so is* `m+n`*. This establishes that the return type of the method is not empty. It must also infer that under the condition* `n==0`*,* `m+n` *is identical to* `m`*, and hence returning the value* `l` *is type-correct. It must also determine that the type of* `rest.append(m)(l)` *is* `List(m+n-1)<Node>`*, and hence the invocation of the binary constructor is correct.*

The general idea is this. In general, the compiler must use constraints (perform algebraic reasoning). Thus corresponding to each program is an underlying logical theory in which reasoning will be performed.

Each programming language – such as `X10` – will specify the base underlying classes (and the operations on them) which can occur as types in parameter lists. For instance, in the code for `List` above, the only type that appears in parameter lists is `int`, and the only operations on `int` are addition, subtraction, `>=`, `==`, and the only constants are `0` and `1`. (This language falls within Presburger arithmetic, a decidable fragment of arithmetic.) The compiler must come equipped with a constraint solver (decision procedure) that can answer questions of the form: does one constraint entail another? Constraints are atomic formulas built up from these operations, using variables. For instance, the compiler must answer each one of:

```
n >= 2 |- n-1 >= 0
n >= 0, m >= 0 |- m+n >= 0
```

Ultimately, the only variables that will occur in constraints are those that correspond to `config` parameters and those that are defined by implicit parameter definitions. We need to establish that the verification of any class will generate only a finite number of constraints, hence only a finite constraint problem for the constraint solver.

Second, it should be possible for instances of user-defined classes (and operations on them) to occur as type parameters. For the compiler to check conditions

[3]The `X10` compiler must ensure that all types used as types of arguments, fields, or local, loop or try/catch variable are statically known to be non-empty. Without dependent types, all types other than `void` are non-empty. Dependent types introduce the possibility that some types may denote empty sets, e.g. `List(-1)`.

involving such values, it is necessary that the underlying constraint solver be extended.

There are two general ways in which the constraint solver may be extended. Both require that the programmer single out some classes and methods on those classes as *pure*. (We shall think of constants as corresponding to zeroary methods.) Only instances of pure classes and expressions involving pure methods on these instances are allowed in parameter expressions.

How shall constraints be generated for such pure methods? First, the programmer may explicitly supply with each pure method `T m(T1 x1, ..., Tn xn)` a constraint on `n+2` variables in the constraint system of the underlying solver that is entailed by `y = o.m(x1,..., xn)`. Whenever the compiler has to perform reasoning on an expression involving this method invocation, it uses the constraint supplied by the programmer. A second more ambitious possibility is that a symbolic evaluator of the language may be run on the body of the method to automatically generate the corresponding constraint.

Finally an additional possibility is that the constraint solver itself be made extensible. In this case, when a user writes a class which is intended to be used in specifying parameters, he also supplies an additional program which is used to extend the underlying constraint solver used by the compiler. This program adds more primitive constraints and knows how to perform reasoning using these constraints. This is how I expect we will initially implement the X10 language. As language designers and implementers we will provide constraint solvers for finite functions and `Herbrand` terms on top of arithmetic.

4 A more extended example

Below I discuss the problem of specifying the operations on regions, distributions and arrays in X10. I specify the signatures of these classes, making extensive use of dependent types.

4.1 k-dimensional regions

```
package x10.lang;

/** A region represents a k-dimensional space of points. A region is a
 * dependent class, with the value parameter specifying the dimension
 * of the region.
 * @author vj
 * @date 12/24/2004
 */

public final value class region( int dimension : dimension >= 0 )  {

    /** Construct a 1-dimensional region, if low =< high. Otherwise
     * through a MalformedRegionException.
     */
    extern public region (: dimension==1) (int low, int high)
        throws MalformedRegionException;
```

```
/** Construct a region, using the list of region(1)'s passed as
 * arguments to the constructor.
 */
extern public region( List(dimension)<region(1)> regions );

/** Throws IndexOutOfBoundException if i > dimension. Returns the
    region(1) associated with the i'th dimension of this otherwise.
 */
extern public region(1) dimension( int i )
    throws IndexOutOfBoundException;


/** Returns true iff the region contains every point between two
 * points in the region.
 */
extern public boolean isConvex();

/** Return the low bound for a 1-dimensional region.
 */
extern public (:dimension=1) int low();

/** Return the high bound for a 1-dimensional region.
 */
extern public (:dimension=1) int high();

/** Return the next element for a 1-dimensional region, if any.
 */
extern public (:dimension=1) int next( int current )
    throws IndexOutOfBoundException;

extern public region(dimension) union( region(dimension) r);
extern public region(dimension) intersection( region(dimension) r);
extern public region(dimension) difference( region(dimension) r);
extern public region(dimension) convexHull();

/**
   Returns true iff this is a superset of r.
 */
extern public boolean contains( region(dimension) r);
/**
   Returns true iff this is disjoint from r.
 */
extern public boolean disjoint( region(dimension) r);

/** Returns true iff the set of points in r and this are equal.
 */
public boolean equal( region(dimension) r) {
    return this.contains(r) && r.contains(this);
}

// Static methods follow.

public static region(2) upperTriangular(int size) {
    return upperTriangular(2)( size );
}
public static region(2) lowerTriangular(int size) {
    return lowerTriangular(2)( size );
    }
    public static region(2) banded(int size, int width) {
        return banded(2)( size );
    }

    /** Return an \code{upperTriangular} region for a dim-dimensional
     * space of size \code{size} in each dimension.
     */
    extern public static (int dim) region(dim) upperTriangular(int size);

    /** Return a lowerTriangular region for a dim-dimensional space of
     * size \code{size} in each dimension.
     */
    extern public static (int dim) region(dim) lowerTriangular(int size);

    /** Return a banded region of width {\code width} for a
     * dim-dimensional space of size {\code size} in each dimension.
     */
    extern public static (int dim) region(dim) banded(int size, int width);


}
```

4.2 Point

```
package x10.lang;

public final class point( region region ) {
    parameter int dimension = region.dimension;
    // an array of the given size.
    int[dimension] val;

    /** Create a point with the given values in each dimension.
     */
    public point( int[dimension] val ) {
        this.val = val;
    }

    /** Return the value of this point on the i'th dimension.
     */
    public int valAt( int i) throws IndexOutOfBoundException {
        if (i < 1 || i > dimension) throw new IndexOutOfBoundException();
        return val[i];
    }

    /** Return the next point in the given region on this given
     * dimension, if any.
     */
    public void inc( int i )
        throws IndexOutOfBoundException, MalformedRegionException {
        int val = valAt(i);
        val[i] = region.dimension(i).next( val );
    }

    /** Return true iff the point is on the upper boundary of the i'th
     * dimension.
     */
    public boolean onUpperBoundary(int i)
        throws IndexOutOfBoundException {
        int val = valAt(i);
        return val == region.dimension(i).high();
    }

    /** Return true iff the point is on the lower boundary of the i'th
     * dimension.
     */
    public boolean onLowerBoundary(int i)
        throws IndexOutOfBoundException {
        int val = valAt(i);
        return val == region.dimension(i).low();
    }
}
```

4.3 Distribution

```
package x10.lang;

/** A distribution is a mapping from a given region to a set of
 * places. It takes as parameter the region over which the mapping is
 * defined. The dimensionality of the distribution is the same as the
 * dimensionality of the underlying region.

   @author vj
   @date 12/24/2004
 */

public final value class distribution( region region ) {
    /** The parameter dimension may be used in constructing types derived
     * from the class distribution. For instance,
     * distribution(dimension=k) is the type of all k-dimensional
     * distributions.
     */
    parameter int dimension = region.dimension;

    /** places is the range of the distribution. Guranteed that if a
     * place P is in this set then for some point p in region,
     * this.valueAt(p)==P.
     */
    public final Set<place> places; // consider making this a parameter?

    /** Returns the place to which the point p in region is mapped.
     */
    extern public place valueAt(point(region) p);

    /** Returns the region mapped by this distribution to the place P.
        The value returned is a subset of this.region.
     */
    extern public region(dimension) restriction( place P );

    /** Returns the distribution obtained by range-restricting this to Ps.
        The region of the distribution returned is contained in this.region.
     */
    extern public distribution(:this.region.contains(region))
        restriction( Set<place> Ps );

    /** Returns a new distribution obtained by restricting this to the
     * domain region.intersection(R), where parameter R is a region
     * with the same dimension.
     */
    extern public (region(dimension) R) distribution(region.intersection(R))
        restriction();

    /** Returns the restriction of this to the domain region.difference(R),
        where parameter R is a region with the same dimension.
     */
    extern public (region(dimension) R) distribution(region.difference(R))
        difference();

    /** Takes as parameter a distribution D defined over a region
        disjoint from this. Returns a distribution defined over a
        region which is the union of this.region and D.region.
        This distribution must assume the value of D over D.region
        and this over this.region.

        @seealso distribution.asymmetricUnion.
     */
    extern public (distribution(:region.disjoint(this.region) &&
                                dimension=this.dimension) D)
        distribution(region.union(D.region)) union();

    /** Returns a distribution defined on region.union(R): it takes on
        this.valueAt(p) for all points p in region, and D.valueAt(p) for all
        points in R.difference(region).
     */
    extern public (region(dimension) R) distribution(region.union(R))
        asymmetricUnion( distribution(R) D);

    /** Return a distribution on region.setMinus(R) which takes on the
     * same value at each point in its domain as this. R is passed as
     * a parameter; this allows the type of the return value to be
     * parametric in R.
     */
    extern public (region(dimension) R) distribution(region.setMinus(R))
        setMinus();

    /** Return true iff the given distribution D, which must be over a
     * region of the same dimension as this, is defined over a subset
     * of this.region and agrees with it at each point.
     */
    extern public (region(dimension) r)
        boolean subDistribution( distribution(r) D);

    /** Returns true iff this and d map each point in their common
     * domain to the same place.
     */
    public boolean equal( distribution( region ) d ) {
        return this.subDistribution(region)(d)
            && d.subDistribution(region)(this);
    }
```

```
    /** Returns the unique 1-dimensional distribution U over the region 1..k,
     * (where k is the cardinality of Q) which maps the point [i] to the
     * i'th element in Q in canonical place-order.
     */
    extern public static distribution(:dimension=1) unique( Set<place> Q );

    /** Returns the constant distribution which maps every point in its
        region to the given place P.
    */
    extern public static (region R) distribution(R) constant( place P );

    /** Returns the block distribution over the given region, and over
     * place.MAX_PLACES places.
     */
    public static (region R) distribution(R) block() {
        return this.block(R)(place.places);
    }

    /** Returns the block distribution over the given region and the
     * given set of places. Chunks of the region are distributed over
     * s, in canonical order.
     */
    extern public static (region R) distribution(R) block( Set<place> s);


    /** Returns the cyclic distribution over the given region, and over
     * all places.
     */
    public static (region R) distribution(R) cyclic() {
        return this.cyclic(R)(place.places);
    }

    extern public static (region R) distribution(R) cyclic( Set<place> s);

    /** Returns the block-cyclic distribution over the given region, and over
     * place.MAX_PLACES places. Exception thrown if blockSize < 1.
     */
    extern public static (region R)
        distribution(R) blockCyclic( int blockSize)
        throws MalformedRegionException;

    /** Returns a distribution which assigns a random place in the
     * given set of places to each point in the region.
     */
    extern public static (region R) distribution(R) random();

    /** Returns a distribution which assigns some arbitrary place in
     * the given set of places to each point in the region. There are
     * no guarantees on this assignment, e.g. all points may be
     * assigned to the same place.
     */
    extern public static (region R) distribution(R) arbitrary();

}
```

4.4 Arrays

Finally we can now define arrays. An array is built over a distribution and a base type.

```
package x10.lang;

/** The class of all  multidimensional, distributed arrays in X10.

    <p> I dont yet know how to handle B@current base type for the
    array.

 * @author vj 12/24/2004
 */

public final value class array ( distribution dist )<B@P> {
    parameter int dimension = dist.dimension;
    parameter region(dimension) region = dist.region;

    /** Return an array initialized with the given function which
        maps each point in region to a value in B.
     */
    extern public array( Fun<point(region),B@P> init);

    /** Return the value of the array at the given point in the
     * region.
     */
    extern public B@P valueAt(point(region) p);

    /** Return the value obtained by reducing the given array with the
        function fun, which is assumed to be associative and
        commutative. unit should satisfy fun(unit,x)=x=fun(x,unit).
     */
    extern public B reduce(Fun<B@?,Fun<B@?,B@?>> fun, B@? unit);


    /** Return an array of B with the same distribution as this, by
        scanning this with the function fun, and unit unit.
     */
    extern public array(dist)<B> scan(Fun<B@?,Fun<B@?,B@?>> fun, B@? unit);

    /** Return an array of B@P defined on the intersection of the
        region underlying the array and the parameter region R.
     */
    extern public (region(dimension) R)
        array(dist.restriction(R)())<B@P>  restriction();

    /** Return an array of B@P defined on the intersection of
        the region underlying this and the parametric distribution.
     */
    public  (distribution(:dimension=this.dimension) D)
        array(dist.restriction(D.region)())<B@P> restriction();

    /** Take as parameter a distribution D of the same dimension as *
     * this, and defined over a disjoint region. Take as argument an *
     * array other over D. Return an array whose distribution is the
     * union of this and D and which takes on the value
     * this.atValue(p) for p in this.region and other.atValue(p) for p
     * in other.region.
     */
    extern public (distribution(:region.disjoint(this.region) &&
                                dimension=this.dimension) D)
        array(dist.union(D))<B@P> compose( array(D)<B@P> other);

    /** Return the array obtained by overlaying this array on top of
        other. The method takes as parameter a distribution D over the
        same dimension. It returns an array over the distribution
        dist.asymmetricUnion(D).
     */
    extern public (distribution(:dimension=this.dimension) D)
        array(dist.asymmetricUnion(D))<B@P> overlay( array(D)<B@P> other);

    extern public array<B> overlay(array<B> other);

    /** Assume given an array a over distribution dist, but with
     * basetype C@P. Assume given a function f: B@P -> C@P -> D@P.
     * Return an array with distribution dist over the type D@P
     * containing fun(this.atValue(p),a.atValue(p)) for each p in
     * dist.region.
     */
    extern public <C@P, D>
        array(dist)<D@P> lift(Fun<B@P, Fun<C@P, D@P>> fun, array(dist)<C@P> a);

    /**  Return an array of B with distribution d initialized
         with the value b at every point in d.
     */
    extern public static (distribution D) <B@P> array(D)<B@P> constant(B@? b);

}
```