Report on the Experimental Language X10

DRAFT v 0.32: Please do not cite.
Please send comments to vsaraswa@us.ibm.com
(IBM Confidential)

17 July 2004

SUMMARY

This draft report provides an initial description of the programming language X10. X10 is a single-inheritance class-based object-oriented programming language designed for high-performance, high-productivity computing on high-end computers supporting $O(10^5)$ hardware threads and $O(10^{15})$ operations per second.

X10 is based on state-of-the-art object-oriented programming languages and deviates from them only as necessary to support its design goals. The language is intended to have a simple and clear semantics and be readily accessible to mainstream object-oriented programmers. It is intended to support a wide variety of concurrent programming idioms, incuding data parallelism, task parallelism, pipelining, producer/consumer and divide and conquer.

This document provides an initial description of the language. We expect to revise this document in several months in the light of experience gained in implementing and using this language.

The X10 design team consists of David Bacon, Bob Blainey, Perry Cheng, Julian Dolby, Kemal Ebcioglu, Allan Kielstra, Robert O'callahan, Filip Pizlo, V.T. Rajan, Vijay Saraswat (contact author), Vivek Sarkar and Jan Vitek.

For extended discussions and support we would like to thank: Calin Cascaval, Elmootaz Elnozahy, Orren Krieger, John McCalpin, Paul McKenney, Ram Rajamony.

We also thank Jonathan Rhees and William Clinger with help in obtaining the LaTeX style file and macros used in producing the Scheme report, after which this document is based. We also acknowledge the influence of the Java Language Specification [6] document, as evidenced by the numerous citations in the text.

CONTENTS

INTRODUCTION

Background

Bigger computational problems need bigger computers capable of performing a larger number of operations per second. The era of increasing performance by simply increasing clocking frequency now seems to be behind us; faster chips run hotter and current cooling technology does not scale as rapidly as the clock. Instead, computer designers are starting to look at "scale out" systems in which the system's computational capacity is increased by adding additional nodes of comparable power to existing nodes, and connecting nodes with a high-speed communication network.

A central problem with scale out systems is a definition of the *memory model*, that is, a model of the interaction between shared memory and multiple simultaneous (read, write) operations on that memory by multiple processors. The traditional "one operation at a time, to completion" model that underlies Lamport's notion of *sequential consistency* (SC) proves too expensive to implement in hardware, at scale. Various models of *relaxed consistency* have proven too difficult for programmers to work with.

One response to this problem has been to move to a *fragmented memory model.* Multiple processors – each sequentially consistent internally – are made to interact via a relatively language-neutral message-passing format such as MPI [9]. This model has enjoyed some success: several high-performance applications have been written in this style. Unfortunately, this model leads to a *loss of programmer productivity*: the mesage-passing format is integrated into the host language by means of an application-programming interface (API), the programmer must explicitly represent and manage the interaction between multiple processes and choreograph their data exchange; large data-structures (such as distributed arrays, graphs, hash-tables) that are conceptually unitary must be thought of as fragmented across different nodes; all processors must generally execute the same code (in an SPMD fashion) etc.

One response to this problem has been the advent of the *partitioned global address space* (PGAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [2]. These languages permit the programmer to think of a single computation running across the multiple processors, sharing a common address space. All data resides at some processors, which is said to have *affinity* to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global *barriers* are used to ensure that processors remain roughly in lock-step.

X10 is a programming language based on Java in the PGAS family. The fundamental goal of X10 is to enable high-performance, high-productivity programming for high-end (scale-out) computers – for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems etc) as well as commercial server workloads. X10 is based on state-of-the-art object-oriented programming ideas primarily to take advantage of their proven flexibility and ease-of-use for a wide spectrum of programming problems. X10 takes advantage of several years of research (e.g. in the context of the Java Grande forum, [1]) on how to adapt such languages to the context of high-performance numerical computing. Thus X10 provides support for user-defined *value types* (such as `int`, `float`, `complex` etc), including operator overloading, supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [3]) and supports IEEE-standard floating point arithmetic.

The major novel contribution of X10 however is its flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 introduces *places* as an abstraction for a *virtual shared-memory multi-processor*; a computation runs over a large collection of places. Each place hosts some data and runs one or more *activities.* Activities are extremely lightweight threads of execution and may dynamically spawn new activities locally or at remote places. *Clocks* are used to ensure that a programmer-specified, data-dependent set of activities has quiesced before another action is initiated. Arrays may be distributed across multiple places. A static type system allows the programmer to keep track of the location of objects and ensures statically that an activity does not synchronously attempt to read/write remote data.

X10 is an experimental language. This document lays out an initial set of ideas which we expect to be the basis of an initial implementation. Several representative concurrent idioms have found pleasant expression in X10. We intend to develop several full-scale applications to get better experience with the language, and revisit the design in the light of this experience.

DESCRIPTION OF THE LANGUAGE

1. Overview of X10

1.1. Semantics

X10 may be thought of as (generic) Java less concurrency, arrays and built-in types, plus *places*, *activities*, *clocks*, (distributed, multi-dimensional) *arrays* and *value* types. All these changes are motivated by the desire to use the new language for high-end, high-performance, high-productivity computing.

1.1.1. Places and activities

The central new concept in X10 is that of a *place* (§ 5). A place may be thought of conceptually as a "virtual shared-memory multi-processor": a computational unit with a finite, though perhaps dynamically varying, number of hardware threads and a bounded amount of shared memory uniformly accessible by all threads. An X10 program is intended to run on a computer capable of supporting millions of places.

An X10 computation acts on *data objects*(§ 3.3) through the execution of lightweight threads called *activities*(§ 6). Objects are of two kinds. A *scalar* object has a small, statically fixed set of fields, each of which has a distinct name. A scalar object is located at a single place. An *aggregate* object has many fields (the number may be known only when the object is created), uniformly accessed through an index (e.g. an integer) and may be distributed across many places.

X10 has a *unified* or *global address space*. This means that an activity can reference objects at other places. However, an activity may synchronously access data items only in the current place (the place in which the activity is running). It may atomically update one or more data items, but only in the current place. Indeed, all accesses to mutable shared data must occur from within an *atomic section*. To read a remote location, an activity must spawn another activitiy *asynchronously* (§ 6.1). This operation returns immediately, leaving the spawning activity with a *future* (§ 6.2) for the result. This future will be replaced with an actual value once the spawned activity returns a value. Similarly, remote location can be written into only by asynchronously spawning an activity to run at that location.

Throughout its lifetime an activity executes at the same place. An activity may dynamically spawn activities in the current or remote places.

Atomic sections X10 allows a programmer to prefix a statement block with the keyword `atomic`. The type system ensures that such a statement will dynamically access only local data. (The statement may of course throw a `BadPlaceException` – but only because of a failed place cast.) Such a statement is executed by the activity as if in a single step during which all other activities are frozen.

Asynch activities An asynch activity is a statement of the form `async (P) S` where `S` is a statement and `P` is a place expression. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`.

An async expression of type `^T` (read: *future* `T`) has the form `future (P) E` where `E` is an expression of type `T`. It executess the expression `E` at the place `P` as an async activity immediately returning with a future. The future may later be forced causing the activity to be blocked until the return value has been computed by the async activity.

1.1.2. Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic, asynchronous network of activities in an X10 computation. Instead, it becomes necessary to allow a computation to use multiple barriers. X10 *clocks* (§ 7) are designed to offer the functionality of multiple barriers in a dynamic context while still supporting determinate, deadlock-free parallel computation.

Activities may use clocks to repeatedly detect quiescence of arbitrary programmer-specified, data-dependent set of activities. Each activity is spawned with a known set of clocks and may dynamically create new clocks. At any given time an activity is *registered* with zero or more clocks. It may register newly created activities with a clock, un-register itself with a clock, suspend on a clock or require that a statement (possibly involving execution of new async activities) be executed to completion before the clock can advance. At any given step of the execution a clock is in a given phase. It advances to the next phase only when all its registered activities have *quiesced* (by executing a `continue` operation on the clock), and all statements scheduled for execution in this phase have terminated. When a clock advances, all its activities may now resume execution.

Thus clocks act as *barriers* for a dynamically varying collection of activities. They generalize the barriers found in MPI style program in that an activity may use multiple clocks simultaneously. Yet programs using clocks are guaranteed not to suffer from deadlock. Clocks are also integrated into the X10 type system, permitting variables to be declared so that they are `final` in each phase of a clock.

1.1.3. Interfaces and Classes

Programmers write X10 code by writing *generic interfaces* (§ 8) and *generic classes* (§ 9). Generic interfaces and classes may be defined over a collection of *type parameters.* Instances can be created only from *concrete* classes; such a class has all its type parameters (if any) instantiated with concrete classes and concrete interfaces.

1.1.4. Scalar classes

An X10 scalar class (§ 9) has fields, methods and inner types (interfaces, classes), subclasses another class, and implements one or more interfaces. Thus X10 classes live in a single-inheritance code hierarchy. X10 allows the programmer to define literals for classes, and overload infix/prefix/postfix operators.

There are two kinds of scalar classes: *reference* classes (§ 9.1) and *value* classes (§ 9.2).

A reference class typically has updatable fields. Objects of such a class may not be freely copied from place to place. Methods may be invoked on such an object only by an activity in the same place.

A value class (§ 9.2) has no updatable fields (defined directly or through inheritance), and allows no reference subclasses. (Fields may be typed at reference classes, so may contain references to objects with mutable state.) Objects of such a class may be freely copied from place to place, and may be implemented very efficiently. Methods may be invoked on such an object from any place.

X10 has no primitive classes. However, the standard library `x10.lang` supplies (final) value classes `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `complex` and `string`. The user may defined additional arithmetic value classes using the facilities of the language.

1.1.5. Arrays, Regions and Distributions

An X10 array is a function from a *distribution* (§ 10.2) to a base type (which may itself be an array type).

A distribution is a map from a *region* (§ 10.1) to a subset of places. A region is a collection of indices.

Operations are provided to construct regions from other regions, and to iterate over regions. Standard set operations, such as union, disjunction and set difference are available for regions.

A primitive set of distributions is provided, together with operations on distributions. A *sub-distribution* of a distribution is one which is defined on a smaller region and agrees with the distribution at all points. The standard operations on regions are extended to distributions.

In future versions of the language, a programmer may specify new distributions, and new operations on distributions.

A new array can be created by restricting an existing array to a sub-distribution, by combining multiple arrays, and by performing pointwise operations on arrays with the same distribution.

X10 allows array constructors to iterate over the underlying distribution and specify a value at each item in the underlying region. Such a constructor may spawn activities at multiple places.

1.1.6. Nullable type constructor

X10 has a `nullable` type constructor which can be applied uniformly to scalar (value or reference) and array types. This type constructor returns a new type which adds a special value `null` to the set of values of its argument type, unless the argument type already has this value.

1.1.7. Statements and expressions

X10 supports the standard set of primitive operations (assignment, classcasts) and sequential control constructs (conditionals, looping, method invocation, exception raising/catching) etc.

Place casts The programmer may use the standard classcast mechanism (§ 12.4) to cast a value to a located type. A `BadPlaceException` is thrown if the value is not of the given type. This is the only language construct that throws a `BadPlaceException`.

1.1.8. Translating MPI programs to X10

While X10 permits considerably greater flexibility in writing distributed programs and data structures than MPI, it is instructive to examine how to translate MPI programs to X10.

Each separate MPI process can be translated into an X10 place. Async activities may be used to read and write variables located at different processes. A single clock may be used for barrier synchronization between multiple MPI processes. X10 collective operations may be used to implement MPI collective operations. X10 is more general than MPI in (a) not requiring synchronization between two processes in order to enable one to read and write the other's values, (b) permitting the use of high-level atomic sections within a process to obtain mutual exclusion between multiple activities running in the same node (c) permitting the use of multiple clocks to combine the expression of different physics (e.g. computations modeling blood coagulation together with computations involving the flow of blood), (d) not requiring an SPMD style of computation.

1.1.9. Summary and future work

X10 is considerably higher-level than thread-based languages in that it supports dynamically spawning very lightweight activities, the use of atomic operations for mutual exclusion, and the use of clocks for repeated quiescence detection of a data-dependent set of activities. Yet it is much more concrete than languages like HPF in that it forces the programmer to explicitly deal with distribution of data objects. In this the language reflects the designers belief that issues of locality and distribution cannot be hidden from the programmer of high-performance code in high-end computing. A performance model that distinguishes between computation and communication must be made explicit and transparent.[1] At the same time we believe that the place-based type system and support for generic programming will allow the X10 programmer to be highly productive; many of the tedious details of distribution-specific code can be handled in a generic fashion.

We expect the next version of the language to be significantly informed by experience in implementing and using the language. We expect it to have constructs to support continuous program optimization, and allow the programmer to provide guidance on clustering places to (hardware) nodes. For instance, we may introduce a notion of hierarchical clustering of places.

2. Lexical structure

In general, X10 follows Java rules [6, Chapter 3] for lexical structure.

Lexically a program consists of a stream of white space, comments, identifiers, keywords, literals, separators and operators.

Whitespace Whitespace follows Java rules [6, Chapter 3.6]. ASCII space, horizontal tab (HT), form feed (FF) and line terminators constitute white space.

Comments Comments follows Java rules [6, Chapter 3.7]. All text included within the ASCII characters "/*" and "*/" is considered a comment and ignored. All text from the ASCII character "//" to the end of line is considered a comment and ignored.

Identifiers Identifiers are defined as in Java.

Keywords X10 reserves the following keywords from Java:

```
abstract   break  case       catch
class      const  continue   default
do         else   extends    final
finally    for    goto       if
implements import instanceof interface
native     new    package    private
protected  public return     static
super      switch this       throw
throws     try    void       while
```

(Note that the primitive types are no longer considered keywords.)

X10 introduces the following keywords:

```
async     ateach       atomic   clock
clocked   distribution flow     foreach
future    here         nullable place
reference region       value
```

Literals *Note:* We have to figure out the syntax for literals, since we do not wish to build knowledge of any type into the language. For now, assume Java style literals.

Separators X10 has the following separators:

```
( ) { } [
] ; , .
```

Operators X10 has the following operators:

```
=>  <   !   ~   ?   :   ==  <=
>=  !=  &&  ||  ++  --  +   -
*   /   &   |   ^   %   <<  >>
>>> +=  -=  *=  /=  &=  |=  ^=
%=  <<= >>= >>> =
```

3. Types

X10 is a *strongly typed* object language: every variable and expression has a type that is known at compile-time. Further, X10 has a *unified* type system: all data items created at runtime are *objects* (§ 3.3. Types limit the values that variables can hold, and specify the places at which these values lie.

X10 supports two kinds of objects, *reference objects* and *value objects*. Reference objects are instances of *reference classes* (§ 9.1). They may contain mutable fields and must stay resident in the place in which they were created. Value objects are instances of *value classes* (§ 9.2). They are immutable and may be freely copied from place to place. Either reference or value objects may be *scalar* (instances of a non-array class) or *aggregate* (instances of arrays).

Correspondingly, X10 has two kinds of types. A *reference type* consists of a *data type*, which is a set of values, and a *place type* which specifies the place at which the value resides. A *value type* simply consists of a data type, since X10 automatically copies value objects from one place to another. Types are constructed through the application of *type constructors* (§ 3.1).

[1]In this X10 is similar to more modern languages such as ZPL [3].

Types are used in variable declarations, casts, object creation, array creation, class literals and `instanceof` expressions.[1]

A variable is a storage location (§ 3.2). All variables are initialized with a value and cannot be observed without a value. Variables whose value may not be changed after initialization are called *final variables* (or sometimes *constants*). The programmer indicates that a variable is final by using the annotation `final` in the variable declaration and using the `=` operator to set its value. Variables whose value may change are said to be *mutable*. The programmer indicates that a variable is mutable by prefixing it with `mutable` and using the `:=` operator to set its value.

3.1. Type constructors

X10 specifies five *type constructors*. A type constructor takes one or more types as arguments and produces a type.

Some of these constructors – the *generic* constructors – take *type parameters*. Type parameters are of two kinds, *reference type parameters* and *value type parameters*.

TypeParameter ::
`value` *DataTypeName* `[`*TypeBound*`]`
`[reference]` *DataTypeName*`‘@’`*PlaceTypeName* `[`*TypeBound*`]`

A ***PlaceTypeName*** is the name of some place variable currently in scope, or the literal `here` (standing for the “current” place) or the literal `threadlocal`. The role of place types is discussed in more detail in § 3.5.

Type bounds are defined as in Java [4] and place an upper bound on the types that may be substituted in for the type parameter. X10 uses “_” instead of “?” as an anonymous type parameter.

Interface declarations. The *interface declaration* (§ 8) takes as argument one or more interfaces (the *extended* interfaces), one or more type parameters and the definition of constants and method signatures and the name of the defined interface. An interface

Each such declaration introduces a data type constructor.

InterfaceType ::
`value` *InterfaceTypeName* `[`*TypeParameters*`]`
`reference` *InterfaceTypeName* `[`*TypeParameters*`]`

The constructor takes as argument the name of the interface optionally prefixed by `value` or `reference` (if neither is specified `reference` is assumed) followed by values for its type arguments (if any) in angle brackets (e.g. `value Field<double>`). It returns a data type provided that the actual type arguments satisfy the associated bounds. Semantically, the data type is the set of all objects which are instances of classes which are value or reference classes (based on the qualifier to the type) and which implement the interface obtained from the interface declaration by substituting the actual arguments for the formals.

[1]In order to allow this version of the language to focus on the core new ideas, X10 v0.5 does not have user-definable classloaders, though there is no technical reason why they could not have been added.

Reference class declarations. The *reference class declaration* (§ 9.1) takes as argument a reference class (the *extended class*), one or more interfaces (the *implemented interfaces*), one or more type parameters, the definition of fields, methods and inner types, and returns a class of the named type.

It may be used to construct a reference data type by supplying values for its type arguments in the same way as an interface (e.g. `Cell<float>`). Semantically, the data type is the set of all objects which are instances of (subclasses of) the class obtained from the class declaration by substituting the actual arguments for the formals. A reference data type can be augmented to a reference type by adding place information, e.g. `Cell<float>@P`.

Value class declarations. The *value class declaration* (§ 9.1) is similar to the reference class declaration except that it must extend either a value class or a reference class that has no mutable fields. It may be used to construct a value type in the same way as a reference class declaration can be used to construct a reference type. (Note that a value type does not require a place type.)

Array type constructor. X10 v0.5 does not have array class declarations (§ 10). This means that user cannot define new array class types. Instead arrays are created as instances of array types constructed through the application of *array type constructors* (§ 10).

The array type constructor takes as argument a type (the *base type*), a distribution (§ 10.2), and optionally either the keyword `reference` or `value` (the default is reference):

ArrayType::
BaseType `value` `‘[’`[*Region*]`‘]’`
BaseType `[reference]` `‘[’`[*Distribution*]`‘]’`

The keyword `value` indicates that the resulting type is a value array data type all of whose components are final; the keyword “reference” indicates that the resulting type is a reference array data type and the components of the array are mutable. For instance, `int [(32,64)-> P]` is the data type of mutable arrays of `32x64` variables, each containing an `int`, and each located at `P` (see § 10.2). To obtain a reference type, one must specify where the array itself is located; thus `(int [(32,64)->P])@Q` is the type of array objects located at `Q` where the array components themslves are at `P` as discussed above.

Note that a distribution can be multidimensional, arrays can be nested, value arrays of reference base types can be constructed, as can reference arrays of value base type. Indeed, value arrays of reference components (where the components themselves may be arrays) are often useful in programs that desire to share only the bottom layers of the array while allowing the top layers to be copied to the referencing places.

Nullable data type constructor. The *nullary type constructor* (§ 11) takes as argument a base data type and returns a new data type which has the same values as the original one and a value denoted by the literal `null`, in case it did not already have this value.

NullaryDataType ::
  `?`*BaseDataType*

This type constructor may be applied to value or reference, scalar or array types.

3.2. Variables

A variable of a reference data type `reference R` where `R` is the name of an interface (possibly with type arguments) always holds a reference to an instance of a class implementing the interface `R`.

A variable of a reference data type `R` where `R` is the name of a reference class (possibly with type arguments) always holds a reference to an instance of the class `R` or a class that is a subclass of `R`.

A variable of a reference array data type `R [D]` is always an array which has as many variables as the size of the region underlying the distribution `D`. These variables are distributed across places as specified by `D` and have the type `R`.

A variable of a nullary (reference or value) data type `?T` always holds either the value (named by) `null` or a value of type `T` (these cases are not mutually exclusive).

A variable of a value data type `value R` where `R` is the name of an interface (possibly with type arguments) always holds either a reference to an instance of a class implementing `R` or an instance of a class implementing `R`. No program can distinguish between the two cases.

A variable of a value data type `R` where `R` is the name of a value class (possibly with type arguments) always holds a reference to an instance of `R` (or a class that is a subclass of `R`) or an instance of `R` (or a class that is a subclass of `R`). No program can distinguish between the two cases.

A variable of a value array data type `V value [R]` is always an array which has as many variables as the size of the region `R`. Each of these variables is immutable and has the type `V`.

As in Java, X10 supports seven kinds of variables: *class variables* (static variables), *instance variables* (the instance fields of a class), *array components*, *method parameters*, *constructor parameters*, *exception-handler parameters* and *local variables.* See [6, §4.5.3].

3.2.1. Final variables

A final variable satisfies two conditions:

- it can be assigned to at most once,
- it must be assigned to before use.

X10 follows Java language rules in this respect [6, §4.5.4,8.3.1.2,16]. Briefly, the compiler must undertake a specific analysis to statically guarantee the two properties above.

3.2.2. Initial values of variables

Every variable declared at a type must always contain a value of that type.

Every class variable, instance variable or array component variable is initialized with a default value when it is created. A variable declared at a nullary type is always initialized with `null`. For a variable declared at a scalar class type it must be the case that a nullary constructor for that class is visible at the site of the declaration; the variable is initialized with the value returned by invoking this constructor. For a variable declared at an array type it must be the case that the base type is either nullable or a class type with a nullary constructor visible at the site of the declaration. The variable is then initialized with an array defined over the smallest region and default distribution consistent with its declaration and with each component of the array initialized to `null` or the result of invoking the nullary constructor.

Each method and constructor parameter is initialized to the corresponding argument value provided by the invoker of the method. An exception-handling parameter is initialized to the object thrown by the exception. A local variable must be explicitly given a value by initialization or assignment, in a way that the compiler can verify using the rules for definite assignment [6, § 16].

Each class `C` has an explicitly or implicitly defined nullary constructor. If `C` does not have an explicit nullary constructor, it is a compile-time error if the class has a field at (a) a scalar type that is a `class` whose nullary constructor is not visible in `C` or is an `interface`, or (b) at an array type whose base type is a `class` whose nullary constructor is not visible in `C` or is an `interface`.

Otherwise a `public` nullary constructor is created by default. This constructor initializes each field of the class (if any) as if it were a variable of the declared type of the field, as described above.

3.3. Objects

An object is an instance of a scalar class or an array type. It is created by using a class instance creation expression (§ 12.4) or an array creation (§ 10.3) expression, such as an array initializer. An object that is an instance of a reference (value) type is called a *reference* (*value*) *object.* In X10 v0.5 a reference object stays resident at the place at which it was created for its entire lifetime.

X10 has no operation to dispose of a reference. Instead the collection of all objects across all places is globally garbage collected.

X10 objects do not have any synchronization information (e.g. a lock) associated with them. Thus the methods on `java.lang.Object` for waiting/synchronizing/notification etc are not available in X10. Instead the programmer should use atomic sections (S 6.3) for mutual exclusion and clocks (S 7) for sequencing multiple parallel operations.

A reference object may have many references, stored in fields of objects or components of arrays. A change to an object made through one reference is visible through another reference. X10 mandates that all accesses to mutable objects shared between multiple activities must occur in an atomic section (§6.3).

Note that the creation of a remote async activity (§ 6.1) `A` at `P` may cause the automatic creation of references to remote objects at `P`. (A reference to a remote object is called a *remote object reference*, to a local object a *local object reference*.) For instance `A` may be created with a reference to an object at `P` held in a variable referenced by the statement in `A`. Similarly the return of a value by a `future` may cause the automatic creation of a remote object reference, incurring some communication cost. An X10 implementation should try to ensure that the creation of a second or subsequent reference to the same remote object at a given place does not incur any (additional) communication cost.

A reference to an object may carry with it the values of final fields of the object. The programmer is guaranteed that the implementation will not incur the cost of communicating the values of final fields of an object from the place where it is hosted to any other place more than once for each target place, even for reference objects.

X10 does not have an operation (such as Pascal's "dereference" operation) which returns an object given a reference to the object. Rather, most operations on object references are transparently performed on the bound object, as indicated below. The operations on objects and object references include:

- Field access (§ 12.4). An activity holding a reference to a reference object may perform this operation only if the object is local. An activity holding a reference to a value object may perform this operation regardless of the location of the object (since value objects can be copied freely from place to place). In this case the cost of cost of copying the field from the place where the object was created to the referencing place will be incurred at most once per referencing place, according to the rule for final fields discussed above.
- Method invocation (§ 12.4). An activity holding a reference to a reference object may perform this operation only if the object is local. An activity holding a reference to a value object may perform this operation regardless of the location of the object (since value objects can be copied freely). The X10 implementation must guarantee that the cost of copying enough relevant state of the value object to enable this method invocation to succeed is incurred at most once for each value object per place.
- Casting (§ 12.4). An activity can perform this operation on local or remote objects, and does not incur any communication cost.
- `instanceof` operator (§ 12.4). An activity can perform this operation on local or remote objects, and does not incur any communication cost.
- The stable equality operator `==` and `!=` (§ 12.4). An activity can perform these operations on local or remote objects, and does not incur any communication cost.

Built-in interfaces:
```
Field             FixedPoint           FloatingPoint
SignedFixedPoint  UnsignedFixedPoint
```

Built-in reference types:
```
Object Reference
```

Built-in value types:
```
boolean byte         char   complex<Field>
double  doubledouble float  int
long    longlong     short  string
ubyte   ushort       value

place   distribution region clock
```

Figure 3.1: The contents of the `x10.lang` package.

3.4. Built-in types

The package `x10.lang` provides a number of built-in class and interface declarations that can be used to construct types.

For instance several value types are provided that encapsulate abstractions (such as fixed point and floating point arithmetic) commonly implemented in hardware by modern computers.

Please consult [5] for more details.

3.5. Place types and Type reconstruction

X10 distinguishes two kinds of places: *shared places*, which correspond to the memory of individual processors which is shared across multiple activities, and *scoped places* such as threadlocal memory and method memory which is available only in limited scope. X10 v0.5 supports only `threadlocal` scoped memory, i.e. memory accessible only to the current activity. Future versions may support `methodlocal` and `blocklocal` memory.

3.5.1. Place expressions

The following are place expressions:

- `here`: the place of the current computation (in a method) or the place of the current object (in field declarations). At the top level of a method, `here` is the place at which the method invocation's object lives. Inside an `async`, `here` is the place of the nearest enclosing `async`.
- `threadlocal`: the scoped place for the current thread (only in scope for method activations)
- `D[i]`: where `D` is a distribution (§ 10.2) and `i` is a point in the region of the distribution.
- `e.place` where `e` is a variable (§ 3.2) at a reference type. X10 allows the use of the expression `e` in contexts expecting a `place` parameter as shorthand for `e.place`.

Note that X10 v0.5 does not permit the dynamic creation of a place. Each X10 computation is initiated with a fixed number of places, as determined by a configuration parameter. The number is available from place.MAX_PLACES. The set of places is available

Place expressions are used in the following contexts:

- As a place type in a reference type (§ 3.1).
- As a target for an async activity (§ 6.1).
- In a class cast expression (§ 12.4).
- In an instanceof expression (§ 12.4).
- In stable equality comparisons, at type place.
- As a parameter to a method invocation (§ 12.4).
- As a type parameter to a generic class or interface (§ 3.1), or an actual argument to a generic class or interface (§ 12.4).

An example:

```
<P> void foo(Foo@P foo, Bar@P bar) {
   int x = async (P) { return foo.f + bar.b; };
   ...
}
```

Places can be passed as parameters to classes. The place where the object lives is an implicit parameter, accessible within constructors and instance initializers as here:

```
class Foo <place P, place Q> {
    Bar@P bar;
    Baz@Q baz;
    Qux@here qux;
}
```

A reference type Foo (with no place annotation) is always taken as shorthand for Foo@here.

An object can be cast to a particular place. If the object is not at the right place, a BadPlaceException is thrown:

```
<P> m(Object@P obj) {
    Object@here h = (Object@here)obj;
    h.blah();
}
```

Places can be checked for equality. One can view this as the analog of the instanceof operator for places.

```
<P> m(Object@P obj) {
    if (here == P) {
       // will never throw an exception
      Object@here h = (Object@here)obj;
      h.m();
    }
}
```

A special form of object reference type, anywhere or ?, constrains the object to be at a shared place but does not constrain the place. This allows collections of objects at heterogenous places.

```
Object@?[] objects;
```

We allow async and future statements to use ? to stand for any place. For instance:

```
Object o = !future(?){new Object();}
```

spawns an activity at some arbitrary place and return an object created at that place.

Similarly the object reference type current can be used in (i) constructing a distribution for a reference array or (ii) specifying the location of the base type of a reference array. In the first case points mapped to current by the distribution will reside in the same place as the array object itself, and in the second case the value of the array at a particular point is an object in the same place as that array component. Example:

```
Object@current[] objects;
```

Type reconstruction

X10 permits the use of the generic method syntax in variable declarations. Any variable declaration may be prefixed with a type parameter list and may use these parameters in type expressions in the declaration. Such a parameterized variable declaration succeeds at compile time if it is possible for the compiler to assign unique types to the parameters in such a way that the declaration type-checks. The scope of the parameter is the scope of the variable introduced by the declaration. Throughout this scope the parameter has the value inferred by the compiler.

For instance:

```
 // This introduces P as a constant place, the
 // location of objects[0]
<P> Object@P obj = objects[0];
async (P)  obj.blah(); ;
```

Often it is the case that a type parameter is not referenced after it is introduced. In such cases X10 permits the use of "_" (the *anonymous parameter*) as a parameter. Multiple occurrences of "_" are taken to stand for "fresh names" in each occurrence. For instance:

```
<P> _@P obj = objects[0];
async (P)  obj.blah(); ;
```

should be taken as shorthand for

```
<P,Q> Q@P obj = objects[0];
async (P)  obj.blah(); ;
```

where Q does not occur anywhere else in the program. If a declaration only uses the anoymous parameter the angle brackets may be omitted. Thus for example the often used special case:

```
_ obj = new int[D];
```

is shorthand for:

```
<> _@_ obj = new int[D];
```

By default, objects are created `here`. An object can be created in thread local storage by using an `@threadlocal` suffix to the data type name:

```
new Foo@threadlocal<here, here>();
```

Note that the effect of creating an object at a place `P` and returning a reference to it may be obtained by:

```
Foo@P x = future (P)  new Foo();
```

X10 imposes the rule that the lifetime of places passed as place parameters to objects must be no shorter than the lifetime of the object itself. It also maintains the invariant that all place variables in scope are guaranteed to last longer than this method. This implies that given an invocation:

```
new Foo@P<..., Q, ...>
```

the following constraints must be true

- `P` is `threadlocal`, or
- `P` is `here` (and not in an `async`) and `Q` is a parameter to this class, or `Q` is `shared` or `P` is `Q`

In a generic object constructor invocation, a type parameter is always replaced with a type which includes the place. For instance

```
_ s = new Stack<Point@P>();
```

Constructor calls, method calls and field accesses have the following place constraints.

```
Foo@P foo = new Foo@P();
int[]@P data;
foo.m();
int i = foo.f;
foo.f = i;
int i = data[j];
data[j] = i;
```

Here `P` is scoped or `here` or `accessible`. `f` must be a final field.

X10 also provides place inference for `asyncs` and `futures` (§ 6.1):

```
async {Statement;}
async (_) {Statement;}
future {Statement;}
future (_) {Statement;}
```

In the `async` (`future`) statement, the target place of the `async` (`future`) is the unique place that will satisfy the place constraints of the body; if there is more than one such place, an error is thrown at compile-time. We permit the second form so as to allow the programmer to document that s/he intends the compiler to infer the location.

Place variables outside an async are accessible inside the async.

Examples Consider a `Stack` containing elements of some type `T` which must all be located at a given place `P`:

```
class Stack<T@P> {
    ?T@P[] elements;
    int size;
    Stack() {
        // filled with nulls.
        elements = new ?T@P[1000];
        size = 0;
    }
    void push(T@P t) { elements[size++] = t; }
    T@P pop() { return elements[--size]; }
    void fill(T@P v) {
        for (int i : elements) {
           elements[i] = v;
        }
    }
}
```

Consider the following array initializers (§ 10.3):

```
distribution D = block(1000);
// region 1..1000 treated as 1..1000 here
_ data = new int@current[1000](i) return i*i; ;
  // gives in P the place to which data[2] was
  // mapped, i.e. the first place.
<P> int@P q = data[2];
  // initialize the array with 10 times
  // the index value
float[D] d = new float[D] (i) return 10.0*i; ;
float[D] d2 = new float[D] (i) return i*i; ;
float[D] result =
  new float[D] (i) return di] + d2[i]; ;
```

The code fragments type-check because the compiler may make the inference that `here` inside the `ateach` is `D[i]`, and that the place of the elements `d[i]` and `d2[i]` is also `D[i]`. The X10 compiler uses only syntactic equality and simple intraprocedural dataflow identities to determine which places are the same.

The place associated with a particular distribution element may be accessed using array syntax.

```
place P = D[i];
Object@P obj = async  new Object@P(); ;
int x = async (D[77])  return data[77]; ;
```

Distributions can be passed as parameters much as places can.

```
<P> void m(Object@P f) {
   _ stack = new Stack@local <Object@P>();
   fill(stack);
}
```

In the following, the type specifier for the argument expands to `Stack<Object@P>@here`:

```
<P> void fill(Stack<Object@P> stack) {
   for (int i : 1000 ) {
       stack.push( new Object@P(););
```

```
      }
   }

   <P> void unordered_fill(Stack<Object@P> stack) {
      ^Object@P[] futures = new ^Object@P[1000]
            {return new Object@P();};
      for (^Object@P f : futures) {
          stack.push(f);
      }
   }
```

Examples with type inference

```
   <P> void m() {
      _ stack = new Stack@threadlocal<Object, P>();
      fill(stack);
  }

   <P> void fill(Stack@threadlocal<Object@P> stack) {
      for (int i = 0; i < 1000; i++) {
          stack.push(async {return new Object@P();});
      }
  }

  <P> void unordered_fill(Stack@local<Object@P> stack)
      _ values = new _[1000] {return new Object@P();};
      for (_ f : values) {
          stack.push(f);
      }
  }

   // Use of anywhere types and newplace
   // to create heterogenous collections
   void anywhere_test() {
      // create 1000 objects at
      // 1000 different shared places
     _ objs = new Object@?[1000]@local
            (i){ async(?) { return new Object();}};
      for (_ o : objs) {
          <P> Object@P v = o;
          async {v.blah(); };
      }
   }
```

3.6. Conversions and Promotions

X10 v0.5 supports Java's conversions and promotions (identity, widening, narrowing, value set, assignment, method invocation, string, casting conversions and numeric promotions) appropriately modified to support X10's built-in numeric classes rather than Java's primitive numeric types.

This decision may be revisited in future version of the language in favor of a streamlined proposal for allowing user-defined specification of conversions and promotions for value types, as part of the syntax for user-defined operators.

4. Names and packages

X10 supports Java's mechanisms for names and packages [6, §6,§7], including `public`, `protected`, `private` and package-specific access control. X10 supports Java's naming conventions.

X10 also supports Java 1.5 static imports [7].

5. Places

An X10 place is a repository for data and activities. Each place is to be thought of as a locality boundary: the activities running in a place may access data items located at that place with the efficiency of on-chip access. Accesses to remote places may take orders of magnitude longer.

In X10 v0.5, the set of places available to a computation is determined at the time that the program is run and remains fixed through the run of the program. The number of places available to a computation may be determined by querying a run-time int constant (`place.MAX_PLACES`).

All scalar objects created during program execution are located in one place, though they may be accessed from other places. Aggregate objects (arrays) may be distributed across multiple places using distributions.

The set of all places in a running instance of an X10 program {s denoted by `place.places`. (This set may be used to define distributions, for instance, § 10.2.)

The set of all places is totally ordered. Places may be used as keys in hash-tables. Like a value object, a place is "unlocated".

X10 permits user-definable place constants (= final variables = variables that must be assigned before use, and can be assigned only once). Place constants may be used in type expressions after the `@` sign. For instance, consider the following class definition:

```
public class Cell<BaseType@P> {
 BaseType@P value;

 public Cell(BaseType@P value) {
    this.value := value;
 }

 public BaseType@P getValue() {
    return this.value;
 }

 public void setValue(BaseType@P value) {
    this.value := value;
 }
}
```

This class may be used thus:

```
Cell<Point@Q> cell =
   new Cell<Point@Q>(new Point@Q());
```

`Cell` is a generic class whose single located type parameter specifies type and location information. At runtime, `BaseType` will be replaced by an unlocated type (e.g. a class or an interface) and `P` will be replaced by a place constant (e.g. `here`). `P` may be used in the body of `Cell` anywhere a place expression may be used. See § 3.

6. Activities

An X10 computation may have many concurrent *activities* "in flight" at any give time. We use the term activity to denote a piece of code (wtih reference to data) that is intended to execute in parallel with other pieces of code. Activities are much lighter-weight than threads. In particular, there is no object associated with an activity. Activities may not be interrupted, suspended or resumed from outside. There is no notion of "activity groups".

An activity is spawned in a given place and stays in that place for its lifetime. An activity may be *running*, *blocked* on some condition or *terminated*.

An activity may be long-running and may invoke recursive methods (thus may have a stack associated with it). (On the other hand, many activities may contain a single atomic section with a dozen or so operations.)

An activity may have local variables (just like a stack frame) that are not accessible from outside the activity.

An activity may asynchronously and in parallel launch activities at other places.

6.1. Spawning an activity

An activity is created by executing the statement:

```
AsyncStatement::
    async PlaceDesignator BlockStatement
BlockStatement::
   '{'Statement'}'
PlaceDesignator::
    empty
    '('_')'
    '('PlaceExpr')'
    '('NonFinalVariable')'
```

In many cases the compiler may infer the unique place at which the statement is to be executed by an analysis of the types of the variables occuring in the statement. In such cases the programmer may omit the place designator; the compiler will throw an error if it cannot uniquely determine the designated place. The programmer may use the anonymous place syntax (_) to explicitly indicate that the compiler should infer the designated place. In all other circumstances the designated place is determined by examining the specified expression.

The place designator may be any place expression (§ 5) designating a place (e.g. `here`, or a place constant). Or it may be any non-final variable; the designated place is then the location of this variable.

The statement is subject to the restriction that it must be acceptable as the body of a `void` method for an anoymous inner class declared at that point in the code. As such, it may reference variables in lexically enclosing scopes (including `clock` variables, § 7) provided that the compiler is able to infer that these variables are not assigned to after the invocation of the `async` statement. (Java places a similar restriction requiring that such code access only `final` lexically enclosing variables.)

Such a statement is executed by launching a new activity at the the designated place, to execute the specified statement. Any exception thrown by the statement is trapped and an error message printed out in the standard error file.

6.2. Asynchronous Expression and Futures

An asynchronous expression is of the form:

```
AsyncExpr::
  future PlaceDesignator '{'Expr'}'
```

If `T@P` is the type of `Expr` then the type of this expression is `^T@P`, read as *future* `T` *at* `P`. The expression is evaluated at the designated place asynchronously. The calling activity continues immediately with a future for the real value (of the given type).

A future may be stored in variables, communicated in method calls and returned from method invocations. The value computed by the future may be retrieved by invoking the method `force`:

```
^T@P promise = future (P) { a[3];};
T@P value = promise.force();
```

The invocation of this method suspends until the value has been computed by the asynchronous activity. This value is returned by `force`, and memoized so that subsequent invocations return the same value. We also provide the alternate notation `! p` as shorthand for `p.promise()`.

6.2.1. Implementation notes

Futures are provided in X10 for convenience; they may be programmed using conditional atomic sections as follows. Nevertheless introducing futures directly is useful because they use conditional atomic sections in a circumscribed way that is guaranteed not to create deadlocks.

Consider the class:

```
public class Future<T@P> {
    ?Box<T> b := null;

    public Future(Runnable<Runnable<T@P>> r) {
        r.run(new Runnable<T@P>() {
            public void run(T@P t) {
                // respond with the value.
                asynch (Future.this) atomic {
                    Future.this.b := new Box<T@P>(t);
                }
            }
        });
    }
    public T force() {
        when (b != null) {
            return b.value;
        }
    }
}
```

Assume the interface `Runnable` is defined by:

```
value interface Runnable<T@P> {
  void run(T@P value);
}
```

An object of the class `Future` is created with a runnable object `r` which represents the computation to be run to determine the underlying value. `r` is given a runnable object which represents the capability to write the boxed value enapsulated in the future. An attempt to force a future is blocked until such time as the value is known; this value is then returned.

Now an expression `future (P) E`, where the type of `E` is `T@Q` may be translated as follows (assuming that `reply` is not a free variable in `E`). Return a future created with a runnable object (say `q`) which expects a reply object `reply` and which on being run will evaluate the expression `E` at the location `P` and use the `reply` object to communicate the result back to the future.

```
new Future<T@Q>(new Runable<Runnable<T@Q>>() {
    public void run(Runnable<T@Q> reply) {
        async (P) {
            T@Q t = E;
            reply.run(t);
        }
    }
})
```

6.3. Atomic sections

Languages such as Java use low-level synchronization locks to allow multiple interacting threads to coordinate the mutation of shared data. X10 eschews locks in favor of a very simple high-level construct, the *atomic section*.

A programmer may use atomic sections to guarantee that invariants of shared data-structures are maintained even as they are being accessed simultaneously by multiple activities running in the same place.

6.3.1. Unconditional atomic sections

The simplest form of an atomic section is the *unconditional atomic section*:

UnconditionalAtomicSection::
  `atomic` *BlockStatement*

`BlockStatement` may include method calls, conditionals, asynchronous method invocations etc. However, it should not include invocations of conditional atomic sections (see below) which may suspend. (Recall that the invocation of `force` on a future may invoke a conditional atomic section.) Also for the sake of efficient implementation X10 v0.5 requires that the atomic section be *analyzable*, that is, the set of locations that are read and written by the `BlockStatement` are bounded and determined statically.

Such a statement is executed by an activity as if in a single step during which all other concurrent activities in the same place are suspended. If execution of the statement may throw an exception, it is the programmer's responsibility to wrap the atomic section within a `try/finally` clause and include undo code in the finally clause.

The X10 compiler guarantees that if a program compiles correctly then either all memory locations accessed within an atomic section are local or the runtime will throw a `BadPlaceException` because of a failed classcast.

We allow methods of an object to be annotated with `atomic`. Such a method is taken to stand for a method whose body is wrapped within a `atomic` statement.

Note an important property of an (unconditonal) atomic section:

$$\texttt{atomic\{atomic\{S\}\}} \quad = \quad \texttt{atomic\{S\}} \qquad (6.1)$$

Further, an atomic section will eventually terminate successfully or thrown an exception; it may not introduce a deadlock.

Example

The following class implements an atomic set and returns the old value:

```
public class TestAndSet {
  private int v;
  public int run( final int newVal) {
      int old = 0;
      atomic {
          old = this.v;
          this.v = newVal;
      }
      return old;
  }
}
```

6.3.2. Conditional atomic sections

Conditional atomic sections are of the form:

ConditionalAtomicSection::
  `'when'` `'('`*Expression*`')'`*BlockStatement*
  [`'or'` `'('`*Expression*`')'` *BlockStatement*]`*`

In such a statement the one or more expressions are called *guards* and must be `boolean` expressions. The statements are the corresponding *guarded statements*. The first pair of expression and statement is called the *main clause* and the additional pairs are called *auxiliary clauses*. A statement must have a main clause and may have no auxiliary clauses.

An activity executing such a statement suspends until such time as any one of the guards is true in the current state. In that state, the statement corresponding to the first guard that is true is executed. The checking of the guards and the execution of the corresponding guarded statement is done atomically.

We note two common abbreviations. The statement `when (true) S` is behaviorally identical to `atomic S`: it never suspends. Second, `when (c) {;}` may be abbreviated to `await(c);` – it simply indicates that the thread must await the occurrence of a certain condition before proceeding.

Conditions on `when` clauses. For the sake of efficient implementation certain restrictions are placed on the guards and statements in a conditional atomic section. First, guards are required not to have side-effects, not to spawn asynchronous activities and to have a statically determinable upper bound k on their execution. These conditions are expected to be checked statically by the compiler which may impose additionl restrictions (e.g. all method invocations are recursion-free).

Second, as for unconditional atomic sections, guarded statements are required to be bounded and statically anlayzable.

Third, guarded statements are required to be *flat*, that is, they may not contain conditional atomic sections. (The implementation of nested conditional atomic sections may require sophisticated operational techniques such as rollbacks.)

Third, X10 guarantees only *weak fairness* when executing conditional atomic sections. Let c be the guard of some conditional atomic section A. A is required to make forward progress only if c is *eventually stable*. That is, any execution $s_1, s_2, \ldots$ of the system is considered illegal only if there is a j such that c holds in all states s_k for $k > j$ and in which A does not execute. Specifically, if the system executes in such a way that c holds only intermmitently (that is, for some state in which c holds there is always a later state in which c does not hold), A is not required to be executed.

Rationale: The guarantee provided by `wait/notify` in Java is no stronger. Indeed conditional atomic sections may be thought of as a replacement for Java's wait/notify functionality.

Sample usage. There are many ways to ensure that a guard is eventually stable. Typically the set of activities are divided into those that may enable a condition and those that are blocked on the condition. Then it is sufficient to require that the threads that may enable a condition do not disable it once it is enabled. Instead the condition may be disabled in a guarded statement guarded by the condition. This will ensure forward progress, given the weak-fairness guarantee.

Example. The following class shows how to implement a bounded buffer of size 1 in X10 for repeated communication between a sender and a receiver.

```
class OneBuffer<value T> {
  ?Box<T> datum = null;
      public
        void send(T v) {
        when (this.datum == null) {
          this.datum := new Box<T>(datum);
        }
      }
      public
        T receive() {
          when (this.datum != null) {
            T v  = datum.datum;
            value := null;
            return v;
          }
      }
  }
```

Similar techniques may be used to implement semaphores and other inter-process communication mechanisms.

7. Clocks

The standard library for X10, `x10.lang` defines a `final value class`, `clock` intended for repeated quiescence detection of arbitrary, data-dependent collection of activities.

This chapter describes the syntax and semantics of clocks and statements in the language that have parameters of type `clock`.

At any stage of the computation, a clock has zero or more *registered* activities. An activity may use only those clocks it is registered with. An activity may be subscribed to zero or more clocks. An activity is registered with a clock when it is created. During its lifetime the only additional clocks it is registered with are exactly those that it creates. In particular it is not possible for an activity to obtain access to a clock by reading a data-structure. This is accomplished by requiring that clocks cannot be stored in objects, only in “flow” variables that live on the stack.

An activity may perform the following operations on a clock `c`. It may *unregister* with `c`; after this, it may perform no further actions on `c`. It may *check* to see if it is unregistered on a clock. It may *register* a newly forked activity with `c`. It may *mark* a statement `S` for completion in the current phase by executing the statement `now(c) S`. It may *continue* the clock by executing `c.continue();`. This indicate to `c` that it has finished marking all statements it wishes to perform in the current phase. Finally, it may *block* (through the statement `next`) on all the clocks that it is registered with. (This operation implicitly `continue`'s all clocks for the activity.) It will resume from this statement only when all these clocks are ready to advance to the next phase.

A clock becomes ready to advance to the next phase when every activity registered with the clock has executed at least one `continue` operation on that clock and all statements marked for completion in the current phase have been completed.

Though clocks introduce a blocking statement (`next`) an important property of X10 is that clocks cannot introduce deadlocks. That is, the system cannot reach a quiescent state (in which no activity is progressing) from which it is unable to progress. For, before blocking each activity continues all clocks it is registered with. Thus if a configuration were to be stuck (that is, no activity can progress) all clocks will have been continued. But this implies that all activities blocked on `next` may continue and the configuration is not stuck.

7.1. Clock operations

7.1.1. Creating new clocks

Clocks are created using the nullary constructor for `clock`

```
clock timeSynchronizer = new clock();
```

The current activity is automatically registered with the clock.

7.1.2. Transmission of clocks

Clocks may only be stored in *flow* variables, i.e. local variables and parameters to methods (that are marked with the `flow` modifier). Values received by a method in a `flow` argument or stored in a `flow` local variable may not be assigned to fields of objects.

Each activity is initiated with zero or more clocks, namely those available in its lexically enclosing environment that are referenced by the activity. Because of the condition above, an activity cannot acquire any new clocks during its lifetime (except by creating them). Therefore for each activity we can statically determine (a superset of) the clocks used by that activity through a simple flow analysis.

7.1.3. Marking clocks referenced in statements

The programmer may explicitly indicate the set of currently visible clocks that are to be used inside a statement using the clocked statement:

Statement::
  `clock (c1,...,ck)` *Statement*

The X10 compiler throws an error if the statement statically references any clock other than the ones enumerated. The X10 compiler also performs "type reconstruction" on such clock assertions so that often the user may omit the `clock (c1,...,ck)` prefix. Nevertheless the programmer may wish to use this form of the statement to explicitly document the programmer's intention.

The statement may create and use clocks internally.

7.1.4. Clocking statements

An activity may execute the statement

```
now (c) S;
```

(for `c` a clock and `S` is a statement). Execution of this statement causes `S` to be executed. The clock `c` cannot advance as long as `S` (and any asynchronous activity it has spawned) is executing. The statement `S` is said to be *clocked* on `c`.

`S` may not suspend on `c` or any other clock visible to `now (c) S`. (It may create new clocks and suspend on them.) If it is desired to spawn a new activity and allow it to suspend on `c` (or any other clock visible to `now (c) S` then that activity should not be spawned within the scope of `now (c)`. Instead it should internally use `now(c)` as appropriate.

7.1.5. Continuing clocks

An activity may execute the statement

```
c.continue();
```

on a clock it is registered with. Execution of this statement indicates that the activity will clock no further statements on `c` until the clock has advanced.

The compiler should issue an error if any activity has a potentially live execution path from a `continue` statement to a `now` statement on the same clock that does not go through a `next` statement.

7.1.6. Advancing clocks

An activity may execute the statement

```
next c1,...,ck;
```

It is a compile time error for the body of an activity to contain such a statement if the clocks registered for that activity are not the clocks `c1,...,ck` (in some order).

Execution of this statement blocks until all the clocks that the activity is registered with (if any) have advanced. The activity implicitly issues a `continue` on all clocks it is registered with, before suspending.

An X10 computation is said to be *quiescent* on a clock `c` if each activity registered with `c` has continued `c`. Note that once the system is quiescent on `c`, it will remain quiescent on *c* forever (unless the system takes some action), since no other activity can become registered with `c`. That is, quiescence on a clock is a *stable property*.

Once the implementation has detected quiecence on `c`, the system marks all activities registered with `c` as being able to progress on `c`. An activity blocked on `next` resumes execution once it is marked for progress by all the clocks it is registered with.

7.1.7. Dropping clocks

An activity may drop a clock by executing:

```
c.drop();
```

This statement does nothing if the activity has already dropped `c`. The compiler must ensure conservatively that after dropping `c` no activity can mark a statement current on `c` or continue `c`.

7.1.8. Checking for dropped clocks

An activity may check that a clock is dropped by executing:

```
c.dropped()
```

This call returns a `boolean` value: `true` iff the activity has already executed `c.drop()`.

7.1.9. Program equivalences

From the discussion above it should be clear that the following equivalences hold:

$$\texttt{now(c)}\,\texttt{now(d)}\,\texttt{S} = \texttt{now(d)}\,\texttt{now(c)}\,\texttt{S} \quad (7.1)$$

$$\texttt{now(c)}\,\texttt{now(c)}\,\texttt{S} = \texttt{now(c)}\,\texttt{S} \quad (7.2)$$

$$\texttt{c.continue();next;} = \texttt{next;} \quad (7.3)$$

$$\begin{array}{c}\texttt{c.continue();}\\ \texttt{d.continue();}\end{array} = \begin{array}{c}\texttt{d.continue();}\\ \texttt{c.continue;}\end{array} \quad (7.4)$$

Note that `next; next;` is not the same as `next;`. The first will wait for clocks to advance twice, and the second once.

7.1.10. Implementation Notes

Clocks may be implemented efficiently with message passing, e.g. by using short-circuit ideas in [8]. Recall that every activity is spawned with references to a fixed number of clocks. Each reference should be thought of as a global pointer to a location in some place representing the clock. (We shall discuss a further optimization below.) Each clock keeps two counters: the total number of outstanding references to the clock, and the number of activities that are currently suspended on the clock.

When an activity A spawns another activity B that will reference a clock c referenced by A, A *splits* the reference by sending a message to the clock. Whenever an activity drops a reference to a clock, or suspends on it, it sends a message to the clock.

The optimization is that the clock can be represented in a distributed fashion. Each place keeps a local counter for each clock that is referenced by an activity in that place. The global location for the clock simply keeps track of the places that have references and that are quiescent. This can reduce the inter-place message traffic significantly.

7.2. Clocked types

The declaration

```
clocked(c) final int l = r;
```

asserts additionally that in each clock instant `l` is final, i.e. the value of `l` may be reset at the beginning of each phase of `c` but stays constant during the phase.

This statement terminates when the computation of `r` has terminated and the assignment has been performed.

7.2.1. Clocked assignment

We expect that most arrays containing application data will be declared to be `clocked final`. We support this very powerful type declaration by providing a new statement:

```
next(c) l = r;
```

for a variable l declared to be clocked on c. The statement assigns r to the *next* value of l. There may be multiple such assignments before the clock advances. The last such assignment specifies the value of the variable that will be visible after the clock has advanced. If the variable is `clocked final` it is guaranteed that *all* readers of the variable throughout this phase will see the value r.

The expression `r` is implicitly treated as `now(c) r`. That is, the clock `c` will not advance until the computation of `r` has terminated.

7.3. Examples

Consider the core of the ASCI Benchmark Sweep3D program for computing solutions to mass transport problems.

In a nutshell the core computation is a triply nested sequential loop in which the value of a variable in the current iteration is dependent on the values of neighboring variables in a past iteration. Such a problem can be parallelized through pipelining. One visualizes a diagonal wavefront sweeping through the array. An MPI version of the program may be described as follows. There is a two dimensional grid of processors which performs the following computation repeatedly. Each processor synchronously receives a value from the processor to its west, then to its north, then computes some function of these values and computes a new value to be sent to the processor to its east and then to its south. Ignoring the behavior of the boundary processors for the moment such a computation may be described by the following X10 program:

```
region R = [1..n0,1..m0];
clock[R] W,N;
clock(W) final double [cyclic(R)] A;
for (int t : 1..TMax) {
  ateach( i,j:A)
    clock (W[i-1,j],N[i,j-1],W[i,j],N[i,j]) {
      double west = now (W[i-1,j]) future{A[i-1,j]};
      W[i-1,j].continue();
      double north = now (N[i,j-1]) future{A[i,j-1]};
      N[i,j-1].continue();
      next(W[i,j]) A[i,j] = compute(west, north);
      next W[i-1,j],N[i,j-1],W[i,j],N[i,j];
  }
}
```

8. Interfaces

X10 v0.5 interfaces are essentially the same Java interfaces [6, §9]. An interface primarily specifies signatures for public methods. It may extend multiple interfaces.

An interface method taking a single argument of type `T` and returning a value of `T` may be marked with the qualifier `reduction`. This indicates that the method is associative and commutative, and may hence be used in reduction/scanning operations on arrays of `T` (§ 10.4.5).

Future version of X10 will introduce additional structure in interface definitions that will allow the programmer to state additional properties of classes that implement that interface. For instance a method may be declared `pure` to indicate that its evaluation cannot have any side-effects. A method may be declared `local` to indicate that its execution is confined purely to the current place (no communication with other places). Similarly, behavioral properties of the method as they relate to the usage of clocks of the current activity may be specified.

9. Classes

X10 classes are essentially the same as Java classes [6, §8]. Classes are structured in a single-inheritance code hierarchy, may implement multiple interfaces, may have static and instance fields, may have static and instance methods, may have constructors, may have static and instance initializers, may have static and instance inner classes and interfaces. Method signatures may specify checked exceptions. Method definitions may be overridden by subclasses; the overriding definition may have a declared return type that is a subclass of the return type of the definition being overridden. Multiple methods with the same name but different signatures may be provided on a class (ad hoc polymorphism). The public/private/protected/package-protected access modification framework may be used.

Because of its different concurrency model, X10 does not support `transient` and `volatile` field modifiers.

9.1. Reference classes

A reference class is declared with the optional keyword `reference` preceding `class` in a class declaration. Reference class declarations may be used to construct reference types (§ 3.1). Reference classes may have mutable fields. Instances of a reference class are always created in a fixed place and in X10 v0.5 stay there for the lifetime of the object. (Future versions of X10 may support object migration.) Variables declared at a reference type always store a reference to the object, regardless of whether the object is local or remote.

9.2. Value classes

X10 singles out a certain set of classes for additional support. A class is said to be *stateless* if all of its fields are declared to be `final` (§ 3.2.1), otherwise it is *stateful*. (X10 has syntax for specifying an array class with final fields, unlike Java.) A *stateless (stateful) object* is an instance of a stateless (stateful) class.

X10 allows the programmer to signify that a class (and all its descendents) are stateless. Such a class is called a *value class*. The programmer specifies a value class by prefixing the modifier `value` before the keyword `class` in a class declaration. (A class not declared to be a value class will be called a *reference class*.) Each instance field of a value class is treated as `final`. It is legal (but neither required nor recommended) for fields in a value class to be declared final. It is a compile-time error for a value class to inherit from a stateful class or for a reference class to inherit from a value class. For brevity, the X10 compiler allows the programmer to omit the keyword class after value in a value class declaration.

The `nullable` type-constructor (§ 11) can be used to declare variables whose value may be `null` or a value type.

X10 provides a built in definition for `.equals()` for a value type, namely stable equality (`==`, § 12.4). The programmer is free to override `.equals` with his/her own definition. (The behavior of `==` cannot be overridden however.)

Value types also support user-defined operators and constructors.

9.2.1. Representation

Since value objects do not contain any updatable locations, they can be freely copied from place to place. An implementation may use copying techniques even within a place to implement value types, rather than references. This is transparent to the programmer.

More explicitly, X10 guarantees that an implementation must always behave as if a variable of a reference type takes up as much space as needed to store a reference that is either null or is bound to an object allocated on the (appropriate) heap. However, X10 makes no such guarantees about the representation of a variable of value type. The implementation is free to behave as if the value is stored "inline", allocated on the heap (and a reference stored in the variable) or use any other scheme (such as structure-sharing) it may deem appropriate. Indeed, an implementation may even dynamically change the representation of an object of a value type, or dynamically use different representations for different instances (that is, implement automatic box/unboxing of values).

Implementations are strongly encouraged to implement value types as space-efficiently as possible (e.g. inlining them or passing them in registers, as appropriate). Implementations are expected to cache values of remote final value variables by default. If a value is large, the programmer may wish to consider spawning a remote activity (at the place the value was created) rather than referencing the containing variable (thus forcing it to be cached).

9.2.2. Example

```
value LinkedList <Node> {
 Node first;
 ?LinkedList<Node> rest;
 public
    LinkedList(Node first) {
    this(first, null);
 }
 public
   LinkedList(Node first, ?LinkedList<Node> rest) {
        this.first = first;
        this.rest = rest;
      }
      public
        Node first() {
        return first;
      }
      public
        ?LinkedList<Node> rest() {
        return rest;
      }
      public
        void append(LinkedList<Node> l) {
        return (this.rest == null)
            ? new LinkedList<Node>(this.first, l)
            : this.rest.append(l);
      }
      public
        LinkedList<Node> reverse(LinkedList<Node> l) {
        return
          rest == null
          ? new LinkedList<Node>(first, l)
          : rest.reverse(new LinkedList<Node>(first, l));
       }
      public
        LinkedList<Node> reverse() {
        return (rest == null)
            ? this
            : rest.reverse(new LinkedList<Node>(first));
      }
    }
```

9.3. Ranges

X10 v0.5 supports extremely simple forms of `enums`, namely *ranges*. A range is a set of *points*.

For two constants or final variables `a` and `b` of a fixed point type (e.g. `long`), and a third constant or final variable `k` of the same type, the literal `a..b:k` (read: "from `a` to `b` with stride `k`") represents the full range of values from `a` to `b` (inclusive), starting with `a` and taking strides of length `k`. That is, the range contains all the numbers `a`, `a+k`, `a+2*k`, ... `a+n*k` which are less than or equal to `b`. The canonical order for elements of a range is the canonical order for the underlying fixed point type.

The notation `a..b` is read as shorthand for `a..b:1`.

X10 allows ranges to be named:

```
range   E = 0..100;
```

Such ranges may be used in a `for/foreach` loop (§ 12.3) or in a region declaration.

10. Arrays

An array is a mapping from a distribution to a range data type. Multiple arrays may be declared with the same underlying distribution.

Each array has a field `a.distribution` which may be used to obtain the underlying distribution.

The distribution underlying an array `a` may be obtained

10.1. Regions

A region is a set of indices (called *points*). X10 provides a built-in value class, `x10.lang.region`, to allow the creation of new regions and to perform operations on regions. This class is `final` in X10 v0.5; future versions of the language may permit user-definable regions. Since regions play a dual role (values as well as types), variables of type `region` must be initialized and are implicitly `final`.

Each region `R` has a constant rank, `R.rank`, which is a non-negative integer. The literal `[]` represents the *null region* and has rank `0`.

For instance:

```
range  E = 1..100;
region R = [0..99:2, -1..MAX_HEIGHT];
region R = region.upperTriangular(N);
region R = region.banded(N, K);
  // A square region.
region R = [E, E];
  // Same region as above.
region R = [100, 100];
  // A representation for 52*7 days.
region W = [Week, Weekday];
  // Represents  the empty region
region Null = [];
  // Represents the intersection of two regions
region AandB = A && B;
  // represents the union of two regions
region AOrB = A || B;
```

A region may be constructed using a comma-separated list of `Ranges` (§ 9.3) within square brackets, as above and represents the Cartesian product of each of the arguments. For convenience we allow an integer `n` to stand for the enumeration `1..n`. The bound of a dimension may be any final variable of a fixed-point numeric type. X10 v0.5 does not support hierarchical regions.

Various built-in regions are provided through `static` factory methods on `region`. For instance:

- `region.upperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular `N x N` matrix.
- `region.lowerTriangular(N)` returns a region corresponding to the non-zero indices in a lower-triangular `N x N` matrix.
- `region.banded(N, K)` returns a region corresponding to the non-zero indices in a banded `N x N` matrix where the width of the band is `K`

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of a region `R=[1..2,1..2]` are ordered as

```
(1,1), (1,2), (2,1), (2,2)
```

Sequential iteration statements such as `for` (§ 12.3) iterate over the points in a region in the canonical order.

A region is said to be *convex* if it is of the form `[T1,..., Tk]` for some set of enumerations `Ti`. Such a region satisfies the property that if two points p_1 and p_3 are in the region, then so is every point p_2 between them. (Note that `||` may produce non-convex regions from convex regions, e.g. `[1,1] || [3,3]` is a non-convex region.)

For each region `R`, the *convex closure* of `R` is the smallest convex region enclosing `R`. For each integer `i` less than `R.rank`, the term `R.i` represents the enumeration in the `i`th dimension of the convex closure of `R`. It may be used in a type expression wherever an enumeration may be used.

Region variables can be declared and used within user programs. They are implicitly `final` since they can be used within type expressions (and hence must not take on different values at runtime). That is, X10 does not permit the declaration of mutable `region` variables.

10.1.1. Operations on Regions

Various non side-effecting operators (i.e. pure functions) are provided on regions. These allow the programmer to express sparse as well as dense regions.

Let `R` be a region. A subset of `R` is also called a *sub-region.*

Let `R1` and `R2` be two regions.

`R1 && R2` is the intersection of `R1` and `R2`.

`R1 || R2` is the union of the `R1` and `R2`.

`R1 - R2` is the set difference of `R1` and `R2`.

Two regions are `==` if they represent the same set of points.

10.2. Distributions

A *distribution* is a mapping from a region to a set of places. X10 provides a built-in value class, `x10.lang.distribution`, to allow the creation of new distributions and to perform operations on distributions. This class is `final` in X10 v0.5; future versions of the language may permit user-definable distributions. Since distributions play a dual role (values as well as types), variables of type `distribution` must be initialized and are implicitly `final`.

The *rank* of a distribution is the rank of the underlying region.

```
region R = [100]
distribution D = block(R);
distribution D = cycle(R);
distribution D = R -> here;
distribution D = random(R);
```

Let `D` be a distribution. `D.region` denotes the underlying region. `D.places` is the set of places constituting the range of `D` (viewed as a function). Given a point `p`, the expression `D[p]` represents the application of `D` to `p`, that is, the place that `p` is mapped to by `D`.

When operated on as a distribution, a region `R` implicitly behaves as the distribution mapping each item in `R` to `here` (i.e. `R->here`, see below). Conversely, when used in a context expecting a region, a distribution `D` should be thought of as standing for `D.region`.

10.2.1. Operations returning distributions

Let `R` be a region, `Q` a set of places `{p1,...,pk}` (enumerated in canonical order), and `P` a place. All the operations described below are static methods on the class `distribution`.

Unique distribution The distribution `unique(Q)` is the unique distribution from the region `1..k` to `Q` mapping each point `i` to `pi`.

Constant distributions. The distribution `R->P` maps every point in `R` to `P`.

Block distributions. The distribution `block(R, Q)` distributes the elements of `R` (in order) over the set of places `Q` in blocks of size `R` as follows. Let p equal `R div N` and q equal `R mod N`, where `N` is the size of `Q`. The first q places get successive blocks of size $(p+1)$ and the remaining places get blocks of size p.

The distribution `block(R)` is the same distribution as `block(R, place.places)`.

Cyclic distributions. The distribution `cyclic(R, Q)` distributes the points in `R` cyclically across places in `Q` in order.

The distribution `cyclic(R)` is the same distribution as `cyclic(R, place.places)`.

Thus the distribution `cyclic(place.MAX_PLACES)` provides a $1-1$ mapping from the region `place.MAX_PLACES` to the set of all places and is the same as the distribution `unique(place.places)`.

Block distributions. The distribution `block(R, N, Q)` distributes the elements of `R` (in order) over the set of places `Q` in blocks of size `N` as follows. Let p equal `R div N` and q equal `R mod N`, where `N` is the size of `Q`. The first q places get successive blocks of size $(p+1)$ and the remaining places get blocks of size p.

The distribution `block(R, N)` is the same distribution as `block(R, N, place.places)`.

Block cyclic distributions. The distribution `blockCyclic(R, N, Q)` distributes the elements of `R` cyclically over the set of places `Q` in blocks of size `N`.

The distribution `blockCyclic(R, N)` is the same distribution as `block(R, N, place.places)`.

Arbitrary distributions. The distribution `arbitrary(R,Q)` arbitrarily allocates points in `R` to `Q`. As above, `arbitrary(R)` is the same distribution as `arbitrary(R, place.places)`.

Domain Restriction. If `D` is a distribution and `R` is a sub-region of `D.domain`, then `D | R` represents the restriction of `D` to `R`. The compiler throws an error if it cannot determine that `R` is a sub-region of `D.domain`.

Range Restriction. If `D` is a distribution and `P` a place expression, the term `D | P` denotes the sub-distribution of `D` defined over all the points in the domain of `D` mapped to `P`.

Note that `D | here` does not necessarily contain adjacent points. For instance, if `D` is a cyclic distribution, `D | here` will typically contain points that are `P` apart, where `P` is the number of places.

10.2.2. User-defined distributions

Future versions of X10 may provide user-defined distributions, in a way that supports static reasoning.

10.2.3. Operations on Distributions

A *sub-distribution* of `D` is any distribution `E` defined on some subset of the domain of `D`, which agrees with `D` on all points in its domain. We also say that `D` is a *super-distribution* of `E`. A distribution `D1` *is larger than* `D2` if `D1` is a super-distribution of `D2`.

Let `D1` and `D2` be two distributions.

Intersection of distributions. `D1 && D2`, the intersection of `D1` and `D2`, is the largest common sub-distribution of `D1` and `D2`.

Asymmetric union of distributions. `D1[D2]`, the asymmetric union of `D1` and `D2`, is the distribution whose domain is the union of the regions of `D1` and `D2`, and whose value at each point `p` in its domain is `D1[p]` if `p` lies in `D.domain` otherwise it is `D2[p]`.

Disjoint union of distributions. `D1 || D2`, the disjoint union of `D1` and `D2`, is defined only if the domains of `D1` and `D2` are disjoint. Its value is `D1[D2]` (or equivalently `D2[D1]`. (It is the least super-distribution of `D1` and `D2`.)

Difference of distributions. `D1 - D2` is the largest sub-distribution of `D1` whose domain is disjoint from that of `D2`.

10.2.4. Example

```
<distribution D(1)> public static
    T[D] dotProduct(T a[D], T b[D]) {
    return (new T[D.places] (j) {
        return (new T[D | here] (i) {
            return a[i]*b[i];
        }) >> (+);
   }) >> (+);
}
```

This code returns the inner product of two `T` vectors defined over the same (otherwise unknown) distribution. The result is the sum reduction of an array of `T` with one element at each place in the range of `D`. The value of this array at each point is the sum reduction of the array formed by multiplying the corresponding elements of `a` and `b` in the local sub-array at the current place.

10.3. Array initializer

An array initializer creating a new array with distribution `D` may optionally take a parametrized block of the form `(ind1,..., indk){S}`. Here, `k` may be zero; in this case the statement is written as just `{S}`. For instance:

```
_ data = new int value [1000]
    (i){ return i*i; };
_ data2 = new int[1000->current]@threadlocal
    { return 1; };
```

The first declaration stores in `data` an (immutable) array whose distribution is `(1..1000)-> here`, which is created `here`, and which is initialized with the value `i*i` at index `i`.

The second declaration stores in `data2` a reference to a mutable array (allocated in the `threadlocal` region of the current activity) with `1000` elements each of which is located in the same place as the array (hence is `threadlocal`). Each array component is initialized to `1`.

In general the expression

```
_ data =  new T[D]@P (ind1, ..., indk) { S }
```

should be thought of as creating a new array located at `P` with a `k`-dimensional distribution `D` such that the elements of the array are initialized as if by execution of the code:

```
ateach(ind1, ..., indk : D) {
    A[ind1, ..., indk] =
  (new Object { T val(D ind1,...,indk) {S}})
  .val(ind1, ..., indk);
}
```

Notice that in the method declaration `D` is used as a type.

Other examples:

```
_ data = new int[1000](i){return i*i; };
float[D] d = new float[D] (i){return 10.0*i; };
float[D] d2 = new float[D] (i){return i*i; };
float[D] result = new float[D]
      (i) {return d[i] + d2[i]; };
```

10.4. Operations on arrays

In the following let `a` be an array with distribution `D` and base type `T`. `a` may be mutable or immutable, unless indicated otherwise.

10.4.1. Element operations

The value of `a` at a point `p` in its region of definition is obtained by using the indexing operation `a[p]`. This operation may be used on the left hand side of an assignment operation to update the value.

10.4.2. Constant promotion

For a distribution `D` and a constant or final variable `v` of type `T` the expression `D v` denotes the mutable array with distribution `D` and base type `T` initialized with `v`.

10.4.3. Restriction of an array

Let `D1` be a sub-distribution of `D`. Then `a[D1]` represents the sub-array of `a` with the distribution `D1`.

Recall that a rich set of operators are available on distributions (§ 10.2) to obtain sub-distributions (e.g. restricting to a subregion, to a specific place etc).

10.4.4. Assembling an array

Let `a1,a2` be arrays of the same base type `T` defined over distributions `D1` and `D2` respectively. Assume that both arrays are value or reference arrays.

Assembling arrays over disjoint regions If `D1` and `D2` are disjoint then the expression `a1 || a2` denotes the unique array of base type `T` defined over the distribution `D1 || D2` such that its value at point `p` is `a1[p]` if `p` lies in `D1` and `a2[p]` otherwise. This array is a reference (value) array if `a1` is.

Overlaying an array on another The expression `a1.over(a2)` (read: the array `a1` *overlaid on* `a2`) represents an array whose underlying region is the union of that of `a1` and `a2` and whose distribution maps each point `p` in this region to `D1[p]` if that is defined and to `D2[p]` otherwise. The value `a1.over(a2)[p]` is `a1[p]` if it is defined and `a2[p]` otherwise.

This array is a reference (value) array if `a1` is.

10.4.5. Global operations

Pointwise operations Suppose that `m` is an operation defined on type `T` that takes an argument of type `S` and returns a value of type `R`. Such an operation can be lifted pointwise to operate on a `T` array and an `S` array defined over the same distribution `D` to return an `R` array defined over `D`.

The syntax for such pointwise application is `a.m(b)` where `a` and `b` are `D` arrays.

Reductions Let `m` be a reduction operator (§ 8) defined on type `T`. Let `a` be a value or reference array over base type `T`. Then the operation `a>>m()` returns a value of type `T` obtained by performing `m` on all points in `a` in some order.

This operation involves communication between the places over which the array is distributed. The `X10` implementation guarantees that only one value of type `T` is communicated from a place as part of this reduction process.

Scans Let `m` be a reduction operator (§ 8) defined on type `T`. Let `a` be a value or reference array over base type `T` and distribution `D`. Then the operation `a||m()` returns an array of base type `T` and distribution `D` whose `i`th element (in canonical order) is obtained by performing the reduction `m` on the first `i` elements of `a` (in canonical order).

This operation involves communication between the places over which the array is distributed. The `X10` implementation will endeavour to minimize the communication between places to implement this operation.

11. The Nullable Type Constructor

`X10` supports the prefix type constructor `?`, read as `nullable`. For any type `T`, the type `?T` (read: "`nullable T`") contains all the values of type `T`, and a special `null` value, unless `T` already contains `null`. This value is designated by the literal `null`, which is special in that it has the type `?T` for all types `T`.

This type constructor can be used in any type expression used to declare variables (e.g. local variables, method parameters, class fields, iterator parameters, try/catch parameters etc), or in a new expression (e.g. `new ?T()`. It may be applied to value types, reference types, aggregate types or type parameters. It may not be used in an `extends` clause or an `implements` clause in a class or interface declaration. If `T` is a value (reference) type, then `?T` is defined to be a value (reference) type.

An immediate consequence of the definition of `nullable` is that for any type `T`, the type `??T` is equal to the type `?T`. (The type `??T` can arise when the body of a generic class contains a type `?X` where `X` is a type parameter, and the generic class is instantiated with a type `?T`.)

Any attempt to access a field or invoke a method on the value `null` results in a `NullPointerException` being thrown.

An expression `e` of type `?T` may be checked for nullity using the expression `e==null`. (It is a compile time error for the static type of `e` to not be `?T`, for some `T`.)

Conversions `null` can be passed as an argument to a method call whose corresponding formal parameter is of type `?T` for some type `T`. (This is a widening reference conversion, per [6, Sec 5.1.4].) Similarly it may be returned from a method call of return type `?T` for some type `T`.

For any value `v` of type `T`, the class cast expression `(?T) v` succeeds and specifies a value of type `?T` that may be seen as the "boxed" version of `v`.

X10 permits the widening reference conversion from any type `T` to the type `?T1` if `T` can be widened to the type `T1`. Thus, the type `T` is a subtype of the type `?T`.

Correspondingly, a value `e` of type `?T` can be cast to the type `T`, resulting in a `NullPointerException` if `e` is `null` and `?T` is not equal to `T`, and in the corresponding value of type `T` otherwise. If `T` is a value type this may be seen as the "unboxing" operator.

The expression `(T)null` throws a `ClassCastException` if `T` is not equal to `?T`; otherwise it returns `null` at type `T`. Thus it may be used to check whether `T=?T`.

Arrays of nullary type The nullary type constructor may also be used in (aggregate) instance creation expressions (e.g. `new ?T[R]`). In such a case `T` must designate a (possibly generic) class. Each member of the array is initialized to `null`. (See § 9 for a discussion of how type parameters may be specified to have constructors.)

Implementation notes A value of type `?T` may be implemented by boxing a value of type `T` unless the value is already boxed. The literal `null` may be represented as the unique null reference.

Java compatibility Java provides a somewhat different treatment of `null`. A class definition extends a nullable type to produce a nullable type, whereas primitive types such as `int` are not nullable — the programmer has to explicitly use a boxed version of `int`, `Integer`, to get the effect of `?int`. Wherever Java uses a variable at reference type `S`, and at runtime the variable may carry the value `null`, the X10 programmer should declare the variable at type `?S`. However, there are many situations in Java in which a variable at reference type `S` can be statically determined to not carry null as a value. Such variables should be declared at type `S` in X10

Design rationale The need for nullable arose because X10 has value types and reference types, and arguably the ability to add a `null` value to a type is orthogonal to whether the type is a value type or a reference type. This argues for the notion of nullability as a type constructor.

The key question that remains is whether it should be possible to define "towers", that is, define the type constructor in such a way that `??T` is distinct from `?T`. Here one would think of nullable as a disjoint sum type constructor that adds a value `null` to the interpretation of its argument type even if it already has that value. Thus `??T` is distinct from `?T` because it has one more `null` value. Explicit injection/projection functions (of signature `T -> ?T` to `?T ->T`) would need to be provided.

The designers of X10 felt that while such a definition might be mathematically tenable, and programmatically interesting, it was likely to be too confusing for programmers. More importantly, it would be a deviation from current practice that is not forced by the core focus of X10 (concurrency and distribution). Hence the decision to collapse the tower by requiring that `??T` be equal to `?T`. As discussed below, towers can be obtained through explicit programming.

Examples Consider the following class:

```
final value Box<T> {
  public T datum;
  public Box(T v) { this.datum = v; }
 }
```

Now one may use a variable `x` at type `?Box<V>` to distinguish between the `null` at type `?Box<V>` and at type `V` (if `V` is a nullable type). In the first case the value of `x` will be `null`, in the second case `x.datum` will be `null`.

Such a type may be used to define efficient generic code for memoization:

```
abstract class Memo <V> {
  ?Box<V>[]  values;
  Memo(int n) {
    // initialized to all nulls
    values = new ?Box<V>[n];
  }
  V compute(int key);
  V lookup(int key) {
   if (values[key] != null)
     return values[key].datum;
   V val = compute(key);
   values[key] = new Box<V>(val);
   return val;
  }
}
```

12. Statements and Expressions

X10 inherits all the standard statements of Java, with the expected semantics:

```
EmptyStatement      LabeledStatement
ExpressionStatement IfStatement
SwitchStatement     WhileDo
DoWhile             ForLoop
BreakStatement      ContinueStatement
ReturnStatement   ThrowStatement
TryStatement
```

We focus on the new statements in X10.

12.1. Assignment

It is often the case that an X10 variable is assigned to only once. The user may declare such variables as `final`. However, this is sometimes syntactically cumbersome.

X10 supports the syntax `l := r` for assignment to mutable variables. The user is strongly enouraged to use this syntax to assign variables that are intended to be assigned to more than once. The X10 compiler may issue a warning if it detects code that uses `=` assignment statements on `mutable` variables.

X10 supports assignment `l = r` and `l := r` to array variables. In this case `r` must have the same distribution `D` as `l`. This statement involves control communication between the sites hosting `D`. Each site performs the assignment(s) of array components locally. The assignment terminates when assignment has terminated at all sites hosting `D`.

12.2. Remote Method Invocation

We also introduce shorthand for asynchronous remote method invocation:

```
o -> m(a1,...,ak)
```

is taken as shorthand for one of

```
async (o) {o.m(a1,...,ak);}
future (o){o.m(a1, ..., ak);}
```

based on whether the return type of methodname is `void`.

12.3. Iteration

We introduce k-dimensional versions of iteration operations `for` and `foreach`:

```
foreach(ind1, ..., indk : R) {S}
```

Here `R` is a region.[1] Let `[T1,..., Tk]` be the convex closure of the region. Then the execution of this statement results in the parallel execution of an activity

```
{ final T1 Ind1 = i1;
  ...
  final Tk Indk = ik;
  S
}
```

for each value `[i1,..., ik]` in the region. Note that the number of activities is equal to the cardinality of the region, not its convex closure.

The syntax for the sequential iterator `for` is the same as for `forall`. However each statement is executed in the same activity, one after the other in canonical order.

In a similar fashion we introduce the syntax:

```
ateach( Ind1, ..., Indk : A) {S}
```

to stand for

```
foreach (ind1, ..., indk : A)
 async (A[ind1,...,indk]) {S}
```

In method definitions, for a region `R` we allow the syntax

```
R i1,...,ik;
```

to introduce `k` new parameters which range over the component enumerations of the convex closure of `R`.

12.4. Expressions

X10 inherits all the standard expressions of Java [6, § 15] – as modified to permit generics [4] – with the expected semantics, unless otherwise mentioned below:

Assignment MethodInvocation
Cast Class
ClassInstanceCreationExpression FieldAccessExpression
ArrayCreationExpression ArrayAccessExpression
PostfixExpression PrefixExpression
InfixExpression UnaryOperators
MultiplicativeOperators AdditiveOperators
ShiftOperators RelationalOperators
EqualityOperators BitwiseOperators
ConditionalOperators AssignmentOperators

Expressions are evaluated in the same order as they would in Java (primarily left-to-right).

We focus on the expressions in X10 which have a different semantics.

The classcast operator The classcast operation may be used to cast an expression to a given type:

```
CastExpression::
  ‘(’ValueType‘)’ Expression
  ‘(’ReferenceDataType@PlaceType‘)’ Expression
```

The result of this operation is a value of the given type if the cast is permissible at runtime. For value types whether or not a cast is permissible at runtime is determined as for the Java language [6, §5.5]. For reference types a cast is permissible if the place type of the expression is the given `PlaceType`, and the value of the expression can be cast to the given reference data type per Java rules.

Any attempt to cast an expression of a reference type to a value type (or vice versa) results in a compile-time error. Some casts – such as those that seek to cast a value of a subtype to a supertype – are known to succeed at compile-time. Such casts should not cause extra computational overhead at runtime.

[1] We also permit `R` to be a distribution (array); in this case `R` is taken to stand for `R.region` (`R.distribution.region`).

instanceof operator This operator takes two arguments; the first should be a RelationalExpression and the second a Type. At run time, the result of this operator is true if the RelationalExpression can be cast to Type without a ClassCastException being thrown. Otherwise the result is false.

Stable equality. Reference equality (==, !=) is replaced in X10 by the notion of stable equality so that it can apply uniformly to value and reference types.

Two values may be compared with the infix predicate ==. The call returns the value true if and only if no action taken by any user program can distinguish between the two values. In more detail the rules are as follows.

If the values have a reference type, then both must be references to the same object.

If the values have a value type then they must be structurally equal, that is, they must be instances of the same value class or value array data type and all their fields or components must be ==.

If one of the values is null then the predicate succeeds iff the other value is also null.

The predicate != returns true (false) on two arguments if and only if the predicate == returns false (true) on the same arguments.

EXAMPLE

This example illustrates Jacobi Iteration over a generic field.

This program assumes that the user will supply the distribution. No assumptions are made about the distribution. For instance, the distribution may allocate each item of the array to a different place.

Start with an array a[1..N,1..N] of some arithmetic type. Extend with 0 boundary elements on all sides of the square array. At each step of the iteration, replace the value of a cell with the average of its adjacent cells in the (i,j) dimensions. Compute the error at each iteration as the max of the changes in value across the whole array. Continue the iteration until the error falls below a pre-given bound.

Assume that Field defines 0, 1, +, / and a max operation.

```
public
    class Jacobi<T implements Field,
                 distribution D[1..N,1..N]> {
  final T epsilon;
  final T four = 1'T + 1'T + 1'T + 1'T;
  public Jacobi(T eps) {
    this.epsilon = eps;
  }
  void T[D] jacobi(final T[D] a) {
    clock l = new clock();
    clocked(l) final D[0..N+1,0..N+1] b
       = (D[0,1..N]0'T
         || D[N+1,1..N]0'T
         || D[1..N,0] 0'T
         || D[1..N,N+1] 0'T
         || D[1..N,1..N]a);

    do {
      next l;
      final _ temp
        = new T[D[1..N,1..N]] (i,j) {
          now (l) {
                return
                  ! future {b[i+1,j]}
                + ! future {b[i-1,j]}
                + ! future {b[i,j-1]}
                + ! future {b[i,j+1]} /four;
          }
       };
      next l;
      // Use a reduce operation over the given
      // distribution to compute error.
      T err = (b-temp) >> max();

     // Set up the value of the clocked final
     // variable b for the next phase of the clock
     next(l) b = temp;
    } while (err >= delta);
    return a;
  }
}
```

REFERENCES

[1] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.

[2] William Carlson, Tarek El-Ghazawi, Bob Numrich, and Kathy Yellick. Programming in the Partitioned Global Address Space Model, 2003. Presentation at SC 2003, http://www.gwu.edu/ upc/tutorials.html.

[3] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[4] Gilad Bracha et al. Adding Generics to the Java Programming Language, 2001. JSR 014, http://www.jcp.org/en/jsr/detail?id=014.

[5] Vijay Saraswat et al. The x10 Standard Class Library. Technical report, IBM TJ Watson Research Center, New York, 2004. Forthcoming.

[6] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, 2000.

[7] Sun Microsystems. Extending the Java Programming Language with Eenumerations, Autoboxing, Enhanced for loops and Static Import, 2004. JSR 201, http://www.jcp.org/en/jsr/detail?id=201.

[8] Vijay Saraswat, Kenneth Kahn, Udi Shapiro, and David Weinbaum. Detecting stable properties of networks in concurrent logic programming languages. In *Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 210–222, 1988.

[9] Anthony Skjellum, Ewing Lusk, and William Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface.* MIT Press, 1999.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES