Report on the Experimental Language X10

VERSION 1.01

PLEASE SEND COMMENTS TO VIJAY SARASWAT AT vsaraswa@us.ibm.com

Dec 8, 2006

SUMMARY

This draft report provides an initial description of the programming language X10. X10 is a single-inheritance class-based object-oriented (OO) programming language designed for high-performance, high-productivity computing on high-end computers supporting $\approx 10^5$ hardware threads and $\approx 10^{15}$ 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 OO programmers. It is intended to support a wide variety of concurrent programming idioms.

This document provides an initial description of the language and corresponds to the first implementation of the language.

The X10 design team consists of DAVID BACON, RAJ BARIK, BOB BLAINEY, PHILIPPE CHARLES, PERRY CHENG, CHRISTOPHER DONAWA, JULIAN DOLBY, KEMAL EBCIOĞLU, ROBERT FUHRER, PATRICK GALLOP, CHRISTIAN GROTHOFF, ALLAN KIELSTRA, SRIRAM KRISHNAMOORTHY, NATHANIEL NYSTROM, ROBERT O'CALLAHAN, FILIP PIZLO, V.T. RAJAN, VIJAY SARASWAT (contact author), VIVEK SARKAR, ARMANDO SOLAR-LEZAMA, CHRISTOPH VON PRAUN, JAN VITEK, and TONG WEN.

For extended discussions and support we would like to thank: Calin Cascaval, Norman Cohen, Elmootaz Elnozahy, John Field, Bob Fuhrer, Orren Krieger, Doug Lea, John McCalpin, Paul McKenney, Ram Rajamony, R.K. Shyamasundar, Frank Tip, Pradeep Varma, and Mandana Vaziri.

We 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 acknowledge the influence of the Java™ Language Specification [6] document, as evidenced by the numerous citations in the text.

This document revises Version 0.41 of the Report, released on 7 February 2006. It documents the language corresponding to the first version of the implementation. The implementation was done by RAJ BARIK, PHILIPPE CHARLES, CHRISTOPHER DONAWA, ROBERT FUHRER, CHRISTIAN GROTHOFF, NATHANIEL NYSTROM, VIJAY SARASWAT, VIVEK SARKAR, CHRISTOPH VON PRAUN, and KEMAL EBCIOĞLU. TONG WEN has written many application programs in X10.

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. It is becoming increasingly difficult to mange chip power and heat. 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 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 are made to interact via a relatively language-neutral message-passing format such as MPI [10]. 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* (GAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [2, 11]. These languages permit the programmer to think of a single computation running across 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 modern object-oriented programming language in the GAS family. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity transformational programming for high-end 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^{TM} Grande forum, [8, 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), supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [3]) and supports IEEE-standard floating point arithmetic. Some limited operator overloading is provided for a few "built in" classes in the `x10.lang` package. Future versions of the language will support user-definable operator overloading.

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the GAS model to the *globally asynchronous, locally synchronous* (GALS) model originally developed in hardware and embedded software research. X10 introduces *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 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. An activity may synchronously (and *atomically*) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology. To access or update memory at other places, it must spawn activities asynchronously (either explicitly or implicitly). X10 provides weaker ordering guarantees for inter-place data access, enabling applications to scale. *Immutable* data needs no consistency management and may be freely copied by the implementation between places. One or more *clocks* may be used to order activities running in multiple places. (Multi-dimensional) Arrays may be distributed across multiple places. Arrays support parallel collective operations. A novel exception flow model ensures that exceptions thrown by asynchronous activities can be caught at a suitable parent activity. The type system tracks which memory accesses are local. The programmer may introduce placecasts which verify the access is local at runtime. Linking with native code is supported.

X10 v1.01 adds to Version 0.4 the following features. A dependent type system is introduced: types may be formed by placing constraints on the values of *properties* (`final` instance fields) of the type. In particular, reference types must be qualified with a constraint specifying the place at which instances of the type are located. Various behavior annotations are introduced e.g. `local`, `seq`) – these specify behavioral properties of code (all accesses to mutable data are to local data, no asynchronous activities are spawned) and are verified by the compiler. Method inheritance is required to respect these annotations.

We anticipate a revision of the language in the next several months. The revision will fully define generics for X10, and introduce general closures. We believe both these features will make code more concise and readable.

X10 is an experimental language. Several representative concurrent idioms have already 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. Future versions of the language are expected to support user-definable operators and permit the specification of generic classes and methods.

Two papers have been published on X10. [9] presents a formal semantics for the concurrency and distributed aspects of X10. [4] presents some case studies, and discusses some X10 programming idioms.

1. Overview of X10

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. Places and activities

An X10 program is intended to run on a wide range of computers, from uniprocessors to large clusters of parallel processors supporting millions of concurrent operations. To support this scale, X10 introduces the central concept of *place* (§ 6). Conceptually, a place is a "virtual shared-memory multi-processor": a computational unit with a finite (though perhaps changing) number of hardware threads and a bounded amount of shared memory, uniformly accessible by all threads.

An X10 computation acts on *data objects*(§ 3.6) through the execution of lightweight threads called *activities*(§ 7). 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 and stays at that place throughout its lifetime. 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. The distribution of an aggregate object remains unchanged throughout the computation. X10 assumes an underlying garbage collector will dispose of (scalar and aggregate) objects and reclaim the memory associated with them once it can be determined that these objects are no longer accessible from the current state of the computation. (There are no operations in the language to allow a programmer to explicitly release memory.)

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. To read a remote location, an activity must spawn another activitiy *asynchronously* (§ 7.2). This operation returns immediately, leaving the spawning activity with a *future* (§ 7.5) for the result. 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 blocks X10 introduces statements of the form `atomic S` where `S` is a statement. The type system ensures that such a statement will dynamically access only local data. (The statement may 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.

Asynchronous activities An asynchronous activity is created by a statement `async (P) S` where `P` is a place expression and `S` is a statement. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`.

An asynchronous expression of type `future<T>` has the form `future (P) E` where `E` is an expression of type `T`. `E` may reference `final` variables declared in the lexically enclosing environment. It executess the expression `E` at the place `P` as an asynchronous 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 asynchronous activity.

1.2. Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic network of (possibly diverse) activities in an X10 computation. Instead, it becomes necessary to allow a computation to use multiple barriers. X10 *clocks* (§ 8) 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.

1.3. Interfaces and Classes

Programmers write X10 code by writing *generic interfaces* (§ 9) and *generic classes* (§ 10). 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.

Note: The use of generic types is restricted in this version of the language. The only generic type constructors permitted are the pre-defined constructors for futures and arrays. This restriction will be lifted in future versions of the language.

1.4. Scalar classes

An X10 scalar class (§ 10) 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.

There are two kinds of scalar classes: *reference* classes (§ 10.1) and *value* classes (§ 10.2).

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 (§ 10.2) 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`, `double`, `complex` and `String`. The user may defined additional arithmetic value classes using the facilities of the language.

1.5. Arrays, Regions and Distributions

An X10 array is a function from a *distribution* (§ 11.2) to a base type (which may itself be an array type).

A distribution is a map from a *region* (§ 11.1) to a subset of places. A region is a collection of *points* or *indices*. For instance, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`. Points are used in array index expressions to pick out a particular array element.

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.

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.

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

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.

1.7. 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 (§ 12.4.1) 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`.

1.8. 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 blocks 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.

1.9. Summary and future work

1.9.1. Design for scalability

X10 is designed for scalability. An activity may atomically access only multiple locations in the current place. Unconditional atomic blocks are statically guaranteed to be non-blocking, and may be implemented using non-blocking techniques that avoid mutual exclusion bottlenecks. Dataflow synchronization permits point-to-point coordination between reader/writer activities, obviating the need for barrier-based or lock-based synchronization in many cases.

1.9.2. Design for productivity

X10 is designed for productivity.

Safety and correctness. Programs written in X10 are guaranteed to be statically *type safe*, *memory safe* and *pointer safe*. Static type safety guarantees that at runtime a location contains only those values who's dynamic type satisfies the constraints imposed by the location's static type and every runtime operation performed on the value in a location is permitted by the static type of the location.

Memory safety guarantees that an object may only access memory within its representation, and other objects it has a reference to. X10 supports no pointer arithmetic, and bound-checks array accesses dynamically if necessary. X10 uses dynamic garbage collection to collect objects no longer referenced by the computation. X10 guarantees that no object can retain a reference to an object whose memory has been reclaimed. Further, X10 guarantees that every location is initialized at runtime before it is read, and every value read from a location has previously been written into that location.

Pointer safety guarantees that a null pointer exception cannot be thrown by an operation on a value of a non-nullable type.

Because places are reflected in the type system, static type safety also implies *place safety*: a location may contain references to only those objects whose location satisfies the restrictions of the static place type of the location.

X10 programs that use only clocks and unconditional atomic blocks are guaranteed not to deadlock. Unconditional atomic blocks are non-blocking, hence cannot introduce deadlocks (assuming the implementation is correct).

Many concurrent programs can be shown to be determinate (hence race-free) statically.

Integration. A key issue for any new programming language is how well it can be integrated with existing (external) languages, system environments, libraries and tools.

We believe that X10, like Java, will be able to support a large number of libraries and tools. An area where we expect future versions of X10 to improve on Java like languages is *native integration* (§ **??**). Specifically, X10 will permit permit multi-dimensional local arrays to be operated on natively by native code.

1.9.3. Conclusion

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.

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.

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

DESCRIPTION OF THE LANGUAGE

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:

```
activitylocal async    ateach    atomic
await        clocked   current   foreach
finish       future    here      next
nullable     or        placelocal reference
value        when
```

Literals X10 v1.01 defines literal syntax in the same way as Java does.

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* (§ 3.6. 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* (§ 10.1). They may contain mutable fields and must stay resident in the place in which they were created. Value objects are instances of *value classes* (§ 10.2). 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).

An X10 type is either a *reference type* or a *value type*. Each 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. Types are constructed through the application of *type constructors* (§ 3.1).

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

A variable is a storage location (§ 3.5). 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.

3.1. Type constructors

An X10 type is a pair specifying a *datatype* and a *placetype*. Semantically, a datatype specifies a set of values and a placetype specifies the set of places at which these values may live. Thus taken together, a type specifies both the kind of value permitted and its location.

```
509  Type ::= DataType PlaceTypeSpecifieropt
510    | nullable Type
511    | future < Type >
512  DataType ::= PrimitiveType
513  DataType ::= ClassOrInterfaceType
514    | ArrayType
```

[1] In order to allow this version of the language to focus on the core new ideas, X10 v1.01 does not have user-definable classloaders, though there is no technical reason why they could not have been added.

For simplicity, this version of X10 does not permit the specification of generic classes or interfaces. This is expected to be remedied in future versions of the language.

Every class and interface definition in X10 defines a type with the same name. Additionally, X10 specifies three *type constructors*: `nullable`, the `future`, and array type constructors. We discuss these constructors and place types in detail in the secions that follow; here we briefly discss interface and class declarations.

Interface declarations. An interface declaration specifies a name, a list of extended interfaces, and constants (`public static final` fields) and method signatures associated with the interface. Each interface declaration introduces a type with the same name as the declaration. Semantically, the data type is the set of all objects which are instances of (value or reference) classes that implement the interface. A class implements an interface if it says it does and if it implements all the methods defined in the interface.

The *interface declaration* (§ 9) 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. Each such declaration introduces a data type.

```
426  DataType ::= ClassOrInterfaceType
433  ClassOrInterfaceType ::= TypeName
13   ClassType ::= TypeName
15   TypeName ::= identifier
16    | TypeName . identifier
```

Reference class declarations. The *reference class declaration* (§ 10.1) takes as argument a reference class (the *extended class*), one or more interfaces (the *implemented interfaces*), the definition of fields, methods and inner types, and returns a class of the named type (§ 10.1). Each such declaration introduces a data type. Semantically, the data type is the set of all objects which are instances of (subclasses of) the class.

Value class declarations. The *value class declaration* (§ 10.2) 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.

3.2. The Nullable Type Constructor

X10 supports the prefix type constructor, `nullable`. For any type `T`, the type `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 `nullable T` for all types `T`.

The visibility of the type `nullable T` is the same as the visibility of `T`. The members of the type `nullable T` are the same as those of type `T`. Note that because of this `nullable` may not be regarded as a generic class; rather it is a special type constructor.

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). It may be applied to value types, reference types or aggregate types. It may not be used in an `extends` clause or an `implements` clause in a class or interface declaration. It may not be used in a new expression – a new expression is used to construct

If `T` is a value (respectively, reference) type, then `nullable T` is defined to be a value (respectively, reference) type.

An immediate consequence of the definition of `nullable` is that for any type `T`, the type `nullable nullable T` is equal to the type `nullable T`.

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

An expression `e` of type `nullable 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 `nullable T`, for some `T`.)

Conversions `null` can be passed as an argument to a method call whose corresponding formal parameter is of type `nullable 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 `nullable T` for some type `T`.

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

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

Correspondingly, a value `e` of type `nullable T` can be cast to the type `T`, resulting in a `NullPointerException` if `e` is `null` and `nullable 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 `nullable T`; otherwise it returns `null` at type `T`. Thus it may be used to check whether `T=nullable T`.

Arrays of nullary type The nullary type constructor may also be used in (aggregate) instance creation expressions (e.g. `new (nullable T)[R]`). In such a case `T` must designate a class. Each member of the array is initialized to `null`, unless an explicit array initializer is specified.

Implementation notes A value of type `nullable 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 `nullable 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 `nullable 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 `nullable nullable T` is distinct from `nullable 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 `nullable nullable T` is distinct from `nullable T` because it has one more `null` value. Explicit injection/projection functions (of signature `T -> nullable T` to `nullable 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. As discussed below, this results in no loss of expressiveness because towers can be obtained through explicit programming.

Examples Consider the following class:

```
final value Box {
  public nullable Object datum;
  public Box(nullable Object v) { this.datum = v; }
}
```

Now one may use a variable `x` at type `nullable Box` to distinguish between the `null` at type `nullable Box` and at type `nullable Object`. In the first case the value of `x` will be `null`, in the second case the value of `x.datum` will be `null`.

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

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

3.3. Future type constructor

For any type `t`, `future<t>` is a type:

```
424   Type ::=   future < Type >
```

The type represents a value which when forced will return a value of type `t`. Thus the type makes available the following methods:

```
public Type force()
```

3.4. Array Type Constructors

X10 v1.01 does not have array class declarations (§ 11). 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* (§ 11).

The array type constructor takes as argument a type (the *base type*), an optional distribution (§ 11.2), and optionally either the keyword `reference` or `value` (the default is reference):

```
18   ArrayType ::= Type [ ]
438  ArrayType ::= X10ArrayType
439  X10ArrayType ::= Type [ . ]
440    | Type reference [ . ]
441    | Type value [ . ]
442    | Type [ DepParameterExpr ]
443    | Type reference [ DepParameterExpr ]
444    | Type value [ DepParameterExpr ]
```

The array type `Type[ ]` is the type of all arrays of base type `Type` defined over the distribution `0:N -> here` for some positive integer `N`.

The qualifier `value` (`reference`) specifies that the array is a `value`(`reference`) array. The array elements of a `value` array are all `final`.[2] If the qualifier is not specified, the array is a `reference` array.

[2] Note that the base type of a `value` array can be a value class or a reference class, just as the type of a `final` variable can be a value class or a reference class.

The array type `Type reference [.]` is the type of all (reference) arrays of basetype `Type`. Such an array can take on any distribution, over any region. Similarly, `Type value [.]` is the type of all value arrays of basetype `Type`.

X10 v1.01 also allows a distribution to be specified between `[` and `]`. The distribution must be an expression of type `distribution` (e.g. a `final` variable) whose value does not depend on the value of any mutable variable.

Future extensions to X10 will support a more general syntax for arrays which allows for the specification of dependent types, e.g. `double[:rank 3]`, the type of all arrays of `double` of rank `3`.

3.5. 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 `nullable 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 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`.

X10 supports seven kinds of variables: final *class variables* (static variables), *instance variables* (the instance fields of a class), *array components*, *method parameters*, *constructor parameters*, *exception-handler parameters* and *local variables*.

3.5.1. 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.

3.5.2. Initial values of variables

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

Every class variable must be initialized before it is read, through the execution of an explicit initializer or a static block. Every instance variable must be initialized before it is read, through the execution of an explicit initializer or a constructor. An instance variable declared at a nullable type (and not declared to be `final`) is assumed to have an initializer which sets the value to `null`.

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].

3.6. 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 (§ 11.3) expression, such as an array initializer. An object that is an instance of a reference (value) type is called a *reference* (*value*) *object*.

All value and reference classes subclass from `x10.lang.Object`. This class has one `const` field `location` of type `x10.lang.place`. Thus all objects in X10 are located (have a place). However, X10 permits value objects to be freely copied from place to place because they contain no mutable state. It is permissible for a read of the `location` field of such a value to always return `here` (§ 6.0.2); therefore no space needs to be allocated in the object representation for such a field.

In X10 v1.01 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 blocks (§ 7.6) for mutual exclusion and clocks (§ 8) 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 (§7.6).

Note that the creation of a remote async activity (§ 7.2) `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 implementation should try to ensure that the cost of communicating the values of final fields of an object from the place where it is hosted to any other place is not incurred more than once for each target place.

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). The implementation should try to ensure that the 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 attempt to ensure 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 (§ 12.4.1). An activity can perform this operation on local or remote objects, and should not incur communication costs (to bring over type information) more than once per place.

- `instanceof` operator (§ 12.4.2). An activity can perform this operation on local or remote objects, and should not incur communication costs (to bring over type information) more than once per place.

- The stable equality operator == and `!=` (§ 12.4.3). An activity can perform these operations on local or remote objects, and should not incur communication costs (to bring over relevant information) more than once per place.

3.7. Built-in types

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

3.7.1. The class `Object`

The class `x10.lang.Object` is a superclass of all other classes. A variable of this type can hold a reference to an instance of any scalar or array type.

```
package x10.lang;
public class Object {
  public String toString() {...}
  public boolean equals(Object o) {...}
  public int hashCode() {...}
}
```

The method `equals` and `hashCode` are useful in hashtables, and are defined as in Java. The default implementation of `equals` is stable equality, § 12.4.3. This method may be overridden in a (value or reference) subclass.

3.7.2. The class `String`

X10 supports strings as in Java. A string object is immutable, and has a concatenation operator (+) available on it.

3.7.3. Arithmetic classes

Several value types are provided that encapsulate abstractions (such as fixed point and floating point arithmetic) commonly implemented in hardware by modern computers:

```
boolean byte short char
int long
double float
```

X10 v1.01 defines these data types in the same way as the Java language. Specifically, a program may contain literals that stand for specific instances of these classes. The syntax for literals is the same as for Java (§ refLiterals).

Future Extensions. X10 *may provide mechansims in the future to permit the programmer to specify how a specific value class is to be mapped to special hardware operations (e.g. along the lines of [1]). Similarly, mechanisms may be provided to permit the user to specify new syntax for literals.*

3.7.4. Places, distributions, regions, clocks

X10 defines several other classes in the `x10.lang` package. Please consult the API documentation for more details.

3.7.5. Java utility classes

X10 v1.01 programmers may import and use Java packages such as `java.util`, e.g. `java.util.Set`, `java.lang.System`. X10 programs should not invoke methods that use the `wait/notify/notifyAll` methods on such objects, since this may interfere with X10 synchronization. The implementation does not make imported Java classes automatically extend `x10.lang.Object`.

Future Extensions. *The above represents an* ad hoc *integration of* Java *libraries into* X10*. It has the unfortunate consequence that not every run-time value in an* X10 *program execution is an instance of a subclass of* `x10.lang.Object`.

In the future a more principled and robust scheme will be worked out. Such a scheme will need to attend to the integration of the Java and X10 type systems, and develop a notion of place for Java objects.

3.8. Conversions and Promotions

X10 v1.01 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.

Mon Jul 03 16:00:11 2006

4. Dependent types

Dependent types are a fundamental extension to the type system for Java-like languages. A dependent type records constraints on *properties* (final fields) of the type. Since types are a fundamental building block of Java-like languages, the introduction of dependent types affects many facets of the language simultaneously — the definition of types, classes, interfaces, methods, constructors, fields, inheritance, overriding, overloading, and type related operators (cast, and instanceof).

Indeed dependent types bring substantial expressiveness to Java-like languages. Just as generic classes permit a single definition for a class C to be treated as a template for an unbounded number of classes obtained from C by "instantiating" C with type arguments, so also a dependent type permits a single definition to produce a potentially unbounded family of types. That is, just as generic types permit a programmer to express and use *functions* from types to types, so also dependent types permit a programmer to express and use functions from values to types. Indeed, the family of types generated from C form a lattice of subtypes of C, one for each constraint on the properties of C expressible in the underlying constraint system, but all sharing the same "structure".

In X10 dependent types are checked statically. However, as in Java-like languages an instanceof relation is available to dynamically check that an object belongs to a particular (dependent) type. Also a cast operation is available to force the runtime system to treat an object o as belonging to a particular dependent type (the operation throws a ClassCast exception if it is not possible to do so).

Dependent types are also the basis for an implicit syntax for X10. This is discussed in the last section.

4.1. Properties

The dependent type system is built on the notion of *properties*, for types (classes and interfaces).

```
NormalClassDeclaration ::=
   ClassModifiersopt class identifier
   PropertyListopt Superopt Interfacesopt ClassBody

NormalInterfaceDeclaration ::=
   InterfaceModifiersopt interface identifier
   PropertyListopt ExtendsInterfacesopt InterfaceBody

PropertyList     ::= ( Properties WhereClauseopt )
Properties       ::= Property
                   | Properties , Property
Property         ::= Type identifier
PropertyListopt  ::= $Empty | PropertyList
```

A property has a name and a type. The declaration of a type (class or interface) introduces a sequence of defined properties for the type.

> STATIC SEMANTICS RULE: It is a compile-time error for a class defining a property `P p` to have an ancestor class that defines a property with the name `p`.

Each class `C` defining a property `P p` implicitly has a field

```
public final P p;
```

and a getter method

```
public final P p()  return p;
```

Each interface `I` defining a property `P p` implicitly has a getter method

```
public final P p()  return p;
```

STATIC SEMANTICS RULE: It is a compile-time error for a class or interface defining a property `P p` to have an existing method with the signature `P p()`.

Properties are used to build dependent types from classes, as described below (§ 4.2).

The `WhereClause` in a `PropertyList` specifies an explicit condition on the properties of the type, and is discussed further below (§ 4.3, 4.11).

STATIC SEMANTICS RULE: Every constructor for a class defining properties `P1 p1, ..., Pn pn` must ensure that each of the fields corresponding to the properties is definitely initialized (cf requirement on initialization of final fields in Java) before the constructor returns.

Example 4.1.1 A class representing immutable 2d points, with two properties `i` and `j`.

```
value class Point(int i, int j)  ...
value class point(int rank)  ...
```

□

4.2. Dependent types

A dependent type (deptype) is of the form `C(: c)` where `C` is a class or interface, and `c` is a *constraint*. (`C` is said to be *the base type* of the deptype, and `c` the *constraint* of the deptype.) A constraint is a boolean expression that can only use a predefined set of operators and methods.

```
Type  ::=  PrimitiveType
        | ClassOrInterfaceType
        | ArrayType
        | nullable < Type > DepParamtersopt
        | future < Type > DepParametersopt
PrimitiveType ::= NumericType DepParametersopt
        | boolean DepParametersopt
ClassOrInterfaceType  ::=
  TypeName DepParametersopt PlaceTypeSpecifieropt
ClassType           ::=
  TypeName DepParametersopt PlaceTypeSpecifieropt
InterfaceType       ::=
  TypeName DepParametersopt PlaceTypeSpecifieropt
PlaceTypeSpecifier    ::= ! PlaceTypeopt
PlaceType           ::= current | Expression

DepParameters   ::= ( DepParameterExpr )
DepParameterExpr ::= ArgumentList WhereClauseopt
WhereClause     ::= : Constraint
Constraint      ::= Expression
ArgumentList    ::= Expression
   | ArgumentList , Expression
DepParametersopt ::= null | DepParameters
WhereClauseopt  ::= null | WhereClause
PlaceTypeopt    ::= null | PlaceType
```

STATIC SEMANTICS: VARIABLE OCCURRENCE In a deptype T=C(:c), the only variables that may occur in c are (a) self, (b) properties visible at T, (c) final local variables, final method parameters or final constructor parameters visible at T, (d) final fields visible at T's lexical place in the source program.

STATIC SEMANTICS: THIS RESTRICTIONS The special variable `this` may be used in a depclause for a type `T` only if (a) it occurs in a property declaration for a class, (b) it occurs in an instance method, (c) it occurs in an instance field, (d) it occurs in an instance initializer.

In particular it may not be used in types that occur in a static context, or in the arguments, body or return type of a constructor or in the extends or implements clauses of class and interface definitions. In these contexts the object that `this` would correspond to has either not been formed or is not well defined.

STATIC SEMANTICS: VARIABLE VISIBILITY If a type `T` occurs in a field, method or constructor declaration, then all variables used in `T` must have at least the same visibility as the declaration. The relation "at least the same visibility as" is given by the transitive closure of:

```
public > protected, protected > private
public > package, package > private
```

All inherited properties of a type `T` are visible in the property list of `T`, and the body of `T`.

In general local variables/parameters/properties/fields are visible at `T` if they are defined before `T` in the program. This rule applies to types in property lists as well as parameter lists (for methods and constructors). An exception is made for the return type of a method: all the arguments to the method are considered to be visible, even though they occur lexically after the return type (given the Java syntactic convention that the return type for a method precedes the argument list for the method).

We permit variable declarations `T v` where `T` is obtained from a dependent type `C(:c)` by replacing one or more occurrences of `self` in `c` by `v`. (If such a declaration `T v` is type-correct, it must be the case that the variable `v` is not visible at the type `T`. Hence we can always recover the underlying deptype `C(:c)` by replacing all occurrences of `v` in the constraint of `T` by `self`.)

For instance, `int(: v > 0) v` is shorthand for `int(: self > 0) v`.

STATIC SEMANTICS: CONSTRAINT TYPE The type of c must be boolean.

A variable occuring in the constraint `c` of a deptype, other than `self` or a property of `self`, is said to be a *parameter* of `c`.

An instance `o` of `C` is said to be of type `C (:c)` (or: *belong to* `C(:c)`) if the predicate `c` evaluates to `true` in the current lexical environment, augmented with the binding `[self |-> o]`. We shall use the function $[\![C(:c)]\!]$ to denote the set of objects that belong to `C(:c)`.

4.3. Type definition

A class definition

```
ClassModifiersopt class identifier
   PropertyListopt Superopt Interfacesopt ClassBody
```

and an interface definition

```
InterfaceModifiersopt interface identifier
  PropertyListopt ExtendsInterfacesopt InterfaceBody
```

may reference several deptypes. The types of properties, the specification of the super clause and the specification of interfaces may each involve deptypes.

```
Super ::= extends ClassType
Superopt ::= null | Super
ClassType ::=
  TypeName DepParametersopt PlaceTypeSpecifieropt

Interfaces ::= implements InterfaceTypeList
InterfaceTypeList ::= InterfaceType
      | InterfaceTypeList , InterfaceType
Interfacesopt ::= null | Interfaces
InterfaceType ::=
 TypeName DepParametersopt PlaceTypeSpecifieropt
```

4.4. Where clauses

There is a general recipe for constructing a list of parameters or properties `T1(:c1) x1, ... , Tk(:ck) xk` that must satisfy a given (satisfiable) constraint `c`.

```
class Foo (T1 (: (T2 x2; ...; Tk xk;  c) x1,
      T2 (: (T3 x3; ...; Tk xk;  c) x2,
       ...
      Tk (:  c) xk)         ...
```

The first type `T1 (:T2 x2;...;Tk xk; c) x1` is consistent iff `(exists T1 x1, T2 x2,..., Tk xk) c` is consistent. The second is consistent iff

```
forall T1(: exists (T2 x2,..., Tk xk) c) x1
exists T2 x2. (exists T3 x2,..., Tk xk) c
```

But this is always true. Similarly for the conditions for the other properties.

Thus logically every satisfiable constraint `c` on a list of parameters `x1,..., xk` can be expressed using the dependent types of xi, provided that the constraint language is rich enough to permit existential quantifiers.

Nevertheless we will find it convenient to permit the programmer to explicitly specify a depclause after the list of properties, thus:

```
class Point(int i, int j) { ... }
class Line(Point start, Point end :  end != start)
  { ... }
class Triangle (Line a, Line b, Line c
      : a.end == b.start && b.end == c.start &&
        c.end == a.start) { ... }
class SolvableQuad(int a, int b, int c
                : a*x*x+b*x+c==0)  { ... }
class Circle (int r, int x, int y
           : r > 0 && r*r==x*x+y*y){ ... }
class NonEmptyList extends List(: n > 0) {...}
```

Consider the definition of the class `Line`. This may be thought of as saying: the class `Line` has two fields, `Point start` and `Point end`. Further, every instance of Line must satisfy the constraint that `end !=start`. Similarly for the other class definitions.

In the general case, the production for `NormalClassDeclaration` specifies that the list of properties may be followed by a `WhereClause`:

```
NormalClassDeclaration ::=
    ClassModifiersopt class identifier
    PropertyListopt Superopt Interfacesopt ClassBody

NormalInterfaceDeclaration ::=
   InterfaceModifiersopt interface identifier
   PropertyListopt ExtendsInterfacesopt InterfaceBody

PropertyList    ::= ( Properties WhereClauseopt )
```

All the properties in the list, together with inherited properties, may appear in the `WhereClause`. A property list `T1 x1, ..., Tn xn : c` for a class `Foo` is said to be consistent if each of the `Ti` are consistent and the constraint

```
exists  T1 x1, ..., Tn xn, Foo self . c
```

is valid (always true).

4.5. Type invariants

With every defined class or interface `T` we associate a *type invariant* `i(T)` as follows. The type invariant associated with `x10.lang.Object` is the proposition

```
nullable<place> self.loc
```

The type invariant associated with any interface `I` that extends interfaces `I1,..., Ik` and defines properties `P1 x1, ..., Pn xn` and specifies a where clause `c` is given by:

```
ti(I1) && ... && tk(Ik) && P1 self.x1
&& ... && Pn self.xn && c
```

Similarly the type invariant associated with any class `C` that implements interfaces `I1, ..., Ik`, extends class `D` and defines properties `P1 x1, ..., Pn xn` and specifies a where clause `c` is given by:

```
i(D) && P1 self.x1 && ... && Pn self.xn && c
```

The INT IMPLEMENTS Static Semantic rule below requires that the type invariant associated with a class entail the type invariants of each interface that it implements.

The static semantics rules below guarantee that for any variable `v` of type `T(:c)` (where `T` is an interface name or a class name) the only objects `o` that may be stored in `v` are such that `o` satisfies `i(T)[o/this] && c[o/self]`.

4.6. Consistency of deptypes

A dependent type `C(:c)` may contain zero or more parameters. We require that a type never be empty – so that it is possible for a variable of the type to contain a value. This is accomplished by requiring that the constraint c must be satisfiable *regardless* of the value assumed by parameters to the contraint (if any). Formally, consider a type `T=C(: c)`, with the variables `F1 f1, ..., Fk fk` free in `c`. Let `S=F1 f1, .., Fk fk, Fk+1 fk+1, ... Fn fn` be the smallest set of declarations containing `F1 f1, ..., Fk fk` and closed under the rule: `F f` in `S` if a reference to variable `f` (which is declared as `F f`) occurs in a type in `S`.

(NOTE: The syntax rules for the language ensure that `S` is always finite. The type for a variable `v` cannot reference a variable whose type depends on `v`.)

We say that `T=C(:c)` is *parametrically consistent* (in brief: *consistent*) if

- Each type `F1, ..., Fn` is (recursively) parametrically consistent, and
- It can be established that `forall F1 f1, .., Fn fn. exists C self. c && i(C).`

where `i(C)` is the invariant associated with the type `C` (§ 4.5). Note by definition of `S` the formula on the above has no free variables.

> STATIC SEMANTICS RULE: For a declaration `T v` to be type-correct, `T` must be parametrically consistent. The compiler issues an error if it cannot determine the type is parametrically consistent.

Example 4.6.1 A class that represents a line has two distinct points:

```
class Array(int  rank,
   region(:rank==this.rank) region) {...}
```

□

One can use deptypes to define other closed geometric figures as well.

Example 4.6.2 Here is an example:

```
class Point(int x, int y) {...}
class Line( Point start,
      Point(: self != this.start) end)
{...}
```

□

To see that the declaration `Point(: self != start) end` is parametrically consistent, note that the following formula is valid:

```
forall Line this.
  exists Point self. self != this.start
```

since the set of all `Points` has more than one element.

Example 4.6.3 A triangle has three lines sharing three vertices.

```
class Triangle
 (Line a,
  Line(: a.end == b.start) b,
  Line(: b.end == c.start && c.end == a.start) c)
 { ...}
```

□

Given `Line a`, the type `Line(: a.end == b.start) b` is consistent, and given the two, the type `Line(: b.end == c.start && c.end == a.start) c` is consistent.

4.7. Equivalence of deptypes

Two dependent types `C(:c)` and `C(:d)` are said to be *equivalent* if `c` is true whenever `d` is, and vice versa. Thus, $[\![C(:c)]\!] = [\![C(:d)]\!]$.

Note that two deptypes that are syntactically different may be equivalent. For instance, `int(:self >= 0)` and `int(:self == 0 || self > 0)` are equivalent though they are syntactically distinct. The Java type system is essentially a nominal system – two types are the same if and only if they have the same name. The X10 type system extends the nominal type system of Java to permit constraint-based equivalence.

A dependent type `C(:c)` is said to refine a type `C(:d)` if `c` implies `d`. In such a case we have $[\![C(:c)]\!]$ is a subset of $[\![C(:d)]\!]$. All dependent types defined on `C` refine `C` since `C` is equivalent to `C(:true)`.

4.8. Type checking rules

4.8.1. Class definitions

Consider a class definition

```
ClassModifiersopt
 class C (P1 x1,..., Pn xn)  extends D(:d)
   implements I1(:c1),..., Ik(:ck)
 ClassBody
```

Each of the following static semantics rules must be satisfied:

STATIC SEMANTICS: INT-IMPLEMENTS The type invariant `i(C)` of `C` must entail `ci[this/self]` for each `i` in `1:k`.

STATIC SEMANTICS: SUPER-EXTENDS The return type `c` of each constructor in `ClassBody` must entail `d`.

4.8.2. Constructor definitions

A constructor for a class `C` is guaranteed to return an object of the class on successful termination. This object must satisfy i(C), the class invariant associated with `C` (§ 4.5). However, often the objects returned by a constructor may satisfy *stronger* properties than the class invariant. X10's dependent type system permits these extra properties to be asserted with the constructor in the form of a deptype (the "return type" of the constructor):

```
ConstructorDeclarator ::=
  SimpleTypeName DepParametersopt
 ( FormalParameterListopt WhereClauseopt )
```

As with method declarations, the parameter list for the constructor may specify a where clause that is to be satisfied by the parameters to the list.

Example 4.8.1 Here is another example.

```
public class List(int(:n >=0) n) {
 protected nullable<Object>   value;
 protected nullable<List(n-1)>  tail;
 public List(t.n+1)(Object o, final List t) {
     n=t.n+1;
     tail=t;
     value=o;
 }
 public List(0) () {
     n=0;
     value=null;
     tail=null;
 }
 ...
}
```

The second constructor returns a `List` that is guaranteed to have `n==0`; the first constructor is guaranteed to return a List with `n>0` (in fact, `n==t.n+1`, where the argument to the constructor is `t`). This is recorded by the programmer in the deptype associated with the constructor. □

STATIC SEMANTICS: SUPER-INVOKE Let `C` be a class with properties `P1 p1, ..., Pn pn`, invariant `c` extending the deptype `D(:d)` (where `D` is the name of a class).

For every constructor in `C` the compiler checks that the call to super invokes a constructor for D whose return type is strong enough to entail d. Specifically, if the call to super is of the form `super(e1, ..., ek)` and the static type of each expression ei is Si, and the invocation is statically resolved to a constructor `D(:d1) (T1 x1,..., Tk xk : c)` then it must be the case that

```
S1 x1, ..., Si xi |- Ti xi  (for i in 1:k)
S1 x1, ..., Sk xk |- c
d1[a/self] && S1 x1 ... && Sk xk |- d[a/self]
```

where `a` is a constant that does not appear in `S1 x1 && ... && Sk xk`.

`Static Semantics: Constructor return` The compiler checks that every constructor for `C` ensures that the properties `p1,..., pn` are initialized with values which satisfy `t(C)`, and its own return type `c'` as follows. In each constructor, the compiler checks that the static types `Ti` of the expressions `ei` assigned to pi are such that the following is true

```
T1 p1, ...., Tn pn |- t(C) && c'
```

(Note that for the assignment of ei to pi to be type-correct it must be the case that Ti pi —- Pi pi.)

STATIC SEMANTICS: CONSTRUCTOR INVOCATION The compiler must check that every invocation `C(e1,..., en)` to a constructor is type correct: each argument `ei` must have a static type that is a subtype of the declared type `Ti` for the `i`th argument of the constructor, and the conjunction of static types of the argument must entail the `WhereClause` in the parameter-list of the constructor.

4.9. Field definitions

Not every instance of a class needs to have every field defined on the class. In Java-like languages this is ensured by conditionally setting fields to a default value, such as `null`, in those instances where the fields are not needed.

Consider the class `List` used earlier. Here all instances of `List` returned by the second constructor do not need the fields `value` and `tail`; their value is set to null.

X10 permits a much cleaner solution that does not require default values such as null to be stored in such fields. X10 permits fields to be *guarded*, that is defined only if a certain constraint on the properties of the class, called the `guard` of that field, is true.

```
FieldDeclaration  ::=
   FieldModifiersopt ThisClauseopt
   Type VariableDeclarators ;
ThisClause      ::= this DepParameters
ThisClauseopt    ::= null | ThisClause
```

It is illegal for code to access a guarded field through a reference whose static type does not satisfy the associated guard, even implicitly (i.e. through an implicit `this`). Rather the source program should contain an explict cast, e.g. `C(:c) me = (C(:c)) this`.

STATIC SEMANTICS RULE: Let `f` be a field defined in class `C` with guard `this(:c)`. The compiler declares an error if field `f` is accessed through a reference `o` whose static type is not a subtype of `C(:c)`.

Example 4.9.1 We may now rewrite the List example:

```
public class List(int(:n >=0) n) {
  protected this(:n>0) Object  value;
  protected this(:n>0) List(n-1)  tail;
  public List(t.n+1)(Object o, final List t){
     n=t.n+1;
     List(:n>0) me = (List(:n>0)) this;
     me.tail=t;
     me.value=o;
  }
  public List(0) () {
     n=0;
  }
  ...
}
```

The fields `value` and `tail` do not exist for instances of the class `List(0)`. □

It is a compile-time error for a class to have two fields of the same name, even if their `ThisClauses` are different. A class `C` with a field named `f` is said to *hide* a field in a superclass named `f`.

STATIC SEMANTICS RULE: A class may not declare two fields with the same name.

4.9.1. Field hiding

The definition of field hiding does not take `ThisClauses` in account. Suppose a class `C` has a field

```
this(:c) Foo f;
```

and a subclass `D` of `C` has a field

```
this(:d) Fum f;
```

We will say that `D.f` hides `C.f`, *regardless* of the constraints `c` and `d`. This is in keeping with Java, and permits a naive implementation which always allocates space for a conditional field.

DESIGN RATIONALE It might seem attractive to require that `D.f` hides `C.f` only if `d` entails `c`. This would seem to necessitate a rather complex implementation structure for classes, requiring some kind of a heterogenous translation for deptypes of `C` and `D`. This bears further investigation.

4.10. Method definitions

X10 permits guarded method definitions, similar to guarded field definitions. Additionally, the parameter list for a method may contain a WhereClause.

```
MethodHeader ::=
  MethodModifiersopt ResultType
  MethodDeclarator Throwsopt
MethodDeclarator ::=
  ThisClauseopt identifier
  ( FormalParameterListopt WhereClauseopt )
 | MethodDeclarator [ ]

ResultType ::= Type | void
```

The guard (specified by `ThisClause`) speciifes a constraint `c` on the properties of the class `C` on which the method is being defined. The method exists only for those instances of `C` which satisfy `c`. It is illegal for code to invoke the method on objects whose static type is not a subtype of `C(:c)`.

We relax the rules of lexical visibility and finality for variable references in deptypes for method parameters. Method parameters not necessarily declared to be final are permitted to occur in the types of parameters that occur after them in textual order. Method parameters may also occur in the ReturnType for the method, as long as they are declared final. (Even though the ReturnType occurs lexically before the parameter list, it is considered to lie in the scope of the declarations in the parameter list.)

STATIC SEMANTICS RULE: The compiler checks that every method invocation `o.m(e1,..., en)` for a method is type correct. Each each argument ei must have a static type Si that is a subtype of the declared type Ti for the ith argument of the method, and the conjunction of static types of the arguments must entail the WhereClause in the parameter-list of the method.

The compiler checks that in every method invocation `o.m(e1,..., en)` the static type of o, S, is a subtype of C(:c), where the method is defined in class C and the ThisClause for m is equivalent to `this(:c)`.

Finally, if the declared return type of the method is D(:d), the return type computed for the call is `D(: final S a; S1 x1; ...; Sn xn; d[a/this])`, where a is a new variable that does not occur in `d, S, S1, ... , Sn`,

and `x1,...,xn` are the formal parameters of the method.

Example 4.10.1 Consider the program:

```
public class List(int(:n >=0) n) {
  protected this(:n > 0) Object  value;
  protected this(:n > 0) List(n-1)  tail;
  public List(t.n+1)(Object o, List t) {
      n=t.n+1;
      tail = t;
      value = o;
  }
  public List(:self.n==0) () {
      n=0;
  }
  public List(:self.n==this.n+l.n) append(List l) {
      return (n==0)? l
         : new List( value, tail.append(l));
  }
  public this(:n>0)
    Object nth(final int(:k >= 1 && k <= n) k) {
      return k==1 ? value : tail.nth(k-1);
  }
}
```

The following code fragment

```
List(:self.n==3) u = ...
List(:self.n==x) t = ...;
List(:self.n==x+3) s = t.append(u);
```

will typecheck. The type of the expression `t.append(u)` is

```
List(: final List(:self.n==x) a;
       List(:self.n==3) l; self.n== a.n+l.n)
```

and this simplifies to

```
List(: final List(:self.n==x) a;
       List(:self.n==3) l; self.n== x+3)
```

which, after dropping unused local variables, reduces to:

```
List(: self.n== x+3)
```

□

4.10.1. Method overloading, overriding, hiding, shadowing and obscuring

The definitions of method overloading, overriding, hiding, shadowing and obscuring in X10 are the same as in Java, modulo the following considerations motivated by dependent types.

The definition of a method declaration `m1` "having the same signature as" a method declaration `m2` involves identity of types. Two X10 types are defined to be identical iff they are equivalent (§ 4.7). Two methods are said to have *the same signature* if (a) they have the same number of formal parameters, (b) for each parameter their types are equivalent, and (c) the constraints associated with their ThisTypes are equivalent. It is a compile-time error for there to be two methods with the same name and same signature in a class (either defined in that class or in a superclass).

STATIC SEMANTICS RULE: A class `C` may not have two declarations for a method named `m` – either defined at `C` or inherited –

```
T this(:tc) m(T1(:t1) v1,..., Tn(:tn) vn) {...}
S this(:sc) m(S1(:s1) v1,..., Sn(:sn) vn) {...}
```

if it is the case that the types `this(:tc), T1(:t1), ...., Tn(:tn)` are equivalent to the types `this(:sc), S1(:t1), ...., Tn(:tn)` respectively.

A class `C` inherits from its direct superclass and superinterfaces all their methods visible according to the access restriction modifiers public/private/protected/(package) of the superclass/superinterfaces that are not hidden or overridden. A method `M1` in a class `C` overrides a method `M2` in a superclass `D` if `M1` and `M2` have the same signature. Methods are overriden on a signature-by-signature basis.

A method invocation `o.m(e1,..., en)` is said to have the *static signature* `<T, T1,...,Tn>` where `T` is the static type of `o`, and `T1,..., Tn` are the static types of `e1,..., en` respectively. As in Java, it must be the case that the compiler can determine a single method defined on `T` with argument type `T1,..., Tn`, otherwise a compile-time error is declared. However, unlike Java, the X10 type `T` may be a dependent type `C(:c)`. Therefore, given a class definition for `C` we must determine which methods of `C` are available at a type `C(:c)`. But the answer to this question is clear: exactly those methods defined on `C` are available at the type `C(:c)` whose guard `d` is implied by `c`.

Example 4.10.2 Consider the definitions:

```
class Point(int i, int j) {...}
class Line(Point s, Point(:self != i) e) {
//m1: Both points lie in the right half of the plane
  this(:s.i>= 0 && e.i >= 0) void draw() {...}
// m2 -- Both points lie on the y-axis
  this(:s.i== 0 && e.i == 0) void draw() {...}
// m3 -- Both points lie in the top half of the plane
  this(:s.j>= 0 && e.j >= 0) void draw() {...}
  // m4  -- The general method
  void draw() {...}
}
```

Three different implementations are given for the draw method, one for the case in which the line lies in the right half of the plane, one for the case that the line lies on the y-axis and the third for the case that the line lies in the top half of the plane.

Consider the invocation Line(:s.i ¡ 0) m = ... m.draw();

This generates a compile time error because there is no applicable method definition.

Consider the invocation

```
Line(:s.i>=0 && s.j>=0 && e.i>=0 && e.j>=0)
  m = ...
m.draw();
```

This generates a compile time error because both m1 and m3 are applicable.

Consider the invocation

```
Line(:s.i>=0 && s.j>=0 && e.i>=0) m = ...
m.draw();
```

This does not generate any compile-time error since only m1 is applicable. □

In the last example, notice that at runtime `m1` will be invoked (assuming `m` contains an instance of the `Line` class, and not some subclass of `Line` that overrides this method). This will be the case even if `m` satisfies at runtime the stronger conditions for `m2` (i.e., `s.i==0 && e.i==0`). That is, dynamic method lookup will not take into account the "strongest" constraint that the receiver may satisfy, i.e. its "strongest deptype".

DESIGN RATIONALE. *The design decision that dynamic method lookup should ignore dependent type information was made to keep the design and the implementation simple and to ensure that serious errors such as method invocation errors are captured at compile-time.*

Consider the above example and the invocation

```
Line m = ...
m.draw();
```

*Statically the compiler will not report an error because m4 is the only method that is applicable. However, if dynamic method lookup were to use deptypes then we would face the problem that if m is a line that lives in the upper right quadrant then *both* m2 and m3 are applicable and one does not override the other. Hence we must report an error dynamically.*

As discussed above, the programmer can write code with `instanceof` *and classcasts that perform any application-appropriate discrimination.*

4.11. Interfaces with properties

X10 permits interfaces to have properties and specify an interface invariant. This is necessary so that programmers can build dependent types on top of interfaces and not just classes.

```
NormalInterfaceDeclaration ::=
    InterfaceModifiersopt interface identifier
      PropertyListopt ExtendsInterfacesopt InterfaceBody
PropertyList    ::= ( Properties WhereClauseopt )
Properties      ::= Property
                  | Properties , Property
Property        ::= Type identifier
PropertyListopt  ::= null | PropertyList
```

The invariant associated with an interface is the conjunction of the invariants associated with its superinterfaces and the invariant defined at the interface.

STATIC SEMANTICS RULE: The compiler declares an error if this constraint is not consistent (§ 4.6).

Each interface implicitly defines a nullary getter method `T p()` for each property `T p`.

STATIC SEMANTICS RULE: The compiler issues a warning if the programmer explicitly defines a method with this signature for an interface.

A class `C` (with properties) is said to implement an interface `I` if

- its properties contains all the properties of I,
- its class invariant, i(C), implies i(I)

4.11.1. instanceof

X10 permits Types to be used in an in instanceof expression to determine whether an object is an instance of the given type:

```
RelationalExpression ::=
  RelationalExpression instanceof Type
```

In the above expression, `Type` is any type including deptypes and "primitive" types. The expression `e instanceof T` evaluates to true if and only if the evaluation of `e` results in a value `v` which belongs to the type `T`. This determination may involve checking that the constraint associated with the type is true for the value `v`.

4.11.2. Class casts

X10 permits types to be used in a cast expression:

```
CastExpression ::=
  ( Type ) UnaryExpressionNotPlusMinus
```

In the above expression, Type is any type including deptypes and "primitive" types. The expression `((T) e)` evaluates `e` to a value `v`, and results in `v` if `v` is an instance of the type `T`. Otherwise a `ClassCastException` is thrown. The static type of the expression `(T) e` is `T`.

STATIC SEMANTICS RULE: The compiler checks that the static type of e is either a subtype of T (this situation is sometimes called a "stupid cast"), or a supertype of T. If neither is the case, it throws a compile-time error since the cast must necessarily fail at runtime.

4.11.3. Local variables

Dependent types may be used to specify the types of local variables, including loop variables and parameters for catch clauses.

4.12. Array types

```
ArrayType ::= Type [ ]
  | Type value [ ]
  | Type [ DepParameterExpr ]
  | Type  value [ DepParameterExpr ]
```

X10 has the following built in types:

```
class point(int(:rank>=0) rank)  ...
class region
  (int(:rank>=0) rank,
   // region is a product of rank-1 convex regions.
   boolean rect,
   // on each dim, the low bound is 0.
   boolean lowZero
  )  ...
class dist
  (int (:rank>=0) rank,
   boolean rect,
   boolean lowZero,
   region(: self.rank==rank&&self.rect==rect
          &&self.lowZero==lowZero) region,
   place onePlace
   )   ...
class Array<T>
  (int (:rank>=0) rank,
   boolean rect,
   boolean lowZero,
   place onePlace,
   region region,
   dist(:self.rank==rank &&self.rect=rect
        &&self.lowZero==lowZero
       &&self.region==region
       &&self.onePlace==onePlace) dist
   ) {...}
class ValueArray<T>
 (int (:rank>=0) rank,
   boolean rect,
   boolean lowZero,
   place onePlace,
   region region,
   dist(:self.rank==rank &&self.rect=rect
       &&self.lowZero==lowZero
      &&self.region==region
      &&self.onePlace==onePlace) dist
) {...}
```

The array types on the left are shorthand for the deptypes on the right:

```
Type []  => Array<Type>
Type value [] => ValueArray<Type>
Type  [ DepParameterExpr ]
   => Array<Type>( DepParameterExpr )
Type Value [ DepParameterExpr ]
   => ValueArray<Type>( DepParameterExpr )
```

For `R` is a reference type, the type `R(:c)!current[]` is interpreted as the type of all arrays which are such that the value at a point `p` in its region has the type `R(:c)!self.dist[p]`. (Recall that in X10 a distribution is itself a value array that maps a point to a place.)

Example 4.12.1 The type `double[:rail]` is the Java type `double[]`. The type `double[:rail][:rail]` is the Java type `double[][]`.

The type `double[:rank==N]` is the type of all N-dimensional arrays of doubles.

The type `double[:rank==N&&onePlace==here]` is the type of all N-dimensional arrays of doubles that are local (mapped to one place). □

4.13. Constraint system

The initial release of X10 has a very simple constraint system, permitting only conjunctions of equalities between variables and constants, and existential quantification over typed variables.

Subsquent implementations are intended to support boolean algebra, arithmetic, relational algebra etc to permit types over regions and distributions. We envision this as a major step towards removing most (if not all) dynamic array bounds and places check from X10.

4.13.1. Syntactic abbreviations

4.14. Place types

```
PlaceTypeSpecifier ::= ! PlaceType
PlaceType ::=  any | current | Expression
```

All X10 reference classes extend the class x10.lang.Ref:

```
package x10.lang;
ublic class Ref(place loc)  ...
```

Because of the importance of places in the X10 design, special syntactic support is provided for deptypes involving places.

We now consider the expansions of deptypes with place information.

Unless a deptype `T` (whose base is a reference type) has an `!` suffix, the constraint for `T` is implicitly assumed to contain the clause `self.loc==here`.

```
C( : c) => C(: self.loc==here && c )
```

The expansions for a deptype with an "`!`" suffix are:

```
C( : c)!  => C(: c )  // no self.loc clause.
C( : c)!p => C(: self.loc==p && c )
```

`Static Semantics Rule:` It is a compile time error for the `!`-annotation to be used for deptypes whose base type does not extend `x10.lang.Ref`.

4.15. Implicit Syntax

Recall that the explicit syntax for X10 requires the programmer to use asyncs/future to ensure the Locality Principle: An activity accesses only those mutable locations that reside in the same place as the activity.

Explicit syntax has the advantage that the performance model for X10 is explicit from the syntax. It has the disadvantage that the programmer has to manually reason about the placement of various objects. If the programmer reasons incorrectly then computation may abort at runtime with an exception.

The place-based type system enables the compiler to support the Locality principle. The programmer uses the type system to establish that the types of various objects are local. These assertions are checked by the compiler (as a side-effect of checking dependent types). Additionally, the programmer may now use normal variable syntax to access (read/write) variables, and invoke methods on objects. Suppose the type of the variable v is C(:c). If c establishes loc==here then the compiler generates code for performing the relevant operation on the local variable (read, write, method invocation) synchronously.

Otherwise the compiler generates code in explicit syntax as follows. If the operation is a read, the compiler generates code to perform a future/force on the variable

```
future(v)v.force();
```

If the operation is a write —v=e—, the compiler generates code to perform the write synchronously:

```
final T temp = e;
finish async (v) v = w;
```

If the operation is a read on an array variable a[p] the compiler generates the code:

```
future(a.dist[p]) a[p].force();
```

If the operation is a write —a[p]=e—, the compiler generates code to perform the write synchronously:

```
final point tp = p;
final T t = e;
finish async (a.dist[tp]) a[tp] = t;
```

If the operation is a method invocation —e.m(e1,..., en)— for a void method, the compiler generates code to perform the method invocation synchronously:

```
final T! t = e;
final P1! t1 = e1;
...
final Pn! tn = en;
finish async (t)          t.m(t1,..., tn);
```

If the operation is a method invocation —e.m(e1,...,en)— for a method that returns a value of type E, then the compiler generates the following code:

```
new Runnable()     .run()
```

5. 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 the following naming conventions. Names of `value` classes should start with a lower-case leter, and those of `reference` classes with an upper-case letter. X10 also supports the convention that fields, local variable names and method parameter names that start with an uppercase letter are automatically considered to be annotated with `final`.

6. 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.

X10 provides a built-in value class, `x10.lang.place`; all places are instances of this class. This class is `final` in X10 v1.01.

In X10 v1.01, 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 may be determined by reading (`place.MAX_PLACES`). (This number is specified from the command line/configuration information; see associated `README` documentation.)

All scalar objects created during program execution are located in one place, though they may be referenced 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 may be obtained through the `const` field `place.places`. (This set may be used to define distributions, for instance, § 11.2.)

The set of all places is totally ordered. The first place may be obtained by reading `place.FIRST_PLACE`. The initial activity for an X10 computation starts in this place (§ 7.4). For any place, the operation`next()` returns the next place in the total order (wrapping around at the end). Further details on the methods and fields available on this class may be obtained by consulting the API documentation.

Note: Future versions of the language may permit user-definable places, and the ability to dynamically create places.

Static semantics. Variables of type `place` must be initialized and are implicitly `final`.

6.0.1. Place expressions

Any expression of type `place` is called a place expression. Examples of place expressions are `this.location` (the place at which the current object lives), `place.FIRST_PLACE` (the first place in the system in canonical order).

Place expressions are used in the following contexts:

- As a target for an `async` activity or a future (§ 7.2).
- In a class cast expression (§ 12.4.1).
- In an `instanceof` expression (§ 12.4.2).
- In stable equality comparisons, at type `place`.

Like values of any other type, places may be passed as arguments to methods, returned from methods, stored in fields etc.

6.0.2. `here`

X10 supports a special indexical constant[1] `here`:

```
22 ExpressionName ::= here
```

The constant evaluates to the place at which the current activity is running. Unlike other place expressions, this constant cannot be used as the placetype of fields, since the type of a field should be independent of the activity accessing it.

Example. The code:

```
public class F {
   public void m( F a ) {
     place OldHere = here;
     async ( a ) {
      System.out.println("OldHere == here:"
                      + (OldHere == here));
     }
   }
  public static void main(String[] s) {
     new F().m(future( place.FIRST_PLACE.next())
               { new F()}.force());
  }
}
```

will print out `true` iff the computation was configured to start with the number of places set to `1`.

7. Activities

An X10 computation may have many concurrent *activities* "in flight" at any give time. We use the term activity to denote a dynamic execution instance of a piece of code (with references to data). An activity is intended to execute in parallel with other activities. An activity may be thought of as a very light-weight thread. In X10 v1.01, an activity may not be interrupted, suspended or resumed as the result of actions taken by any other activity.

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*. When the statement associated with an activity terminates normally, the activity terminates normally; when it terminates abruptly with some reason R, the activity terminates with the same reason (§ 7.1).

An activity may be long-running and may invoke recursive methods (thus may have a stack associated with it). On the other hand, an activity may be short-running, involving a fine-grained operation such as a single read or write.

An activity may have an *activitylocal* heap accessible only to the activity.

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

X10 distinguishes between *local* termination and *global* termination of a statement. The execution of a statement by an activity is said to terminate locally when the activity has finished all its computation related to that statement. (For instance the creation of an asynchronous activity terminates locally when the activity has been created.) It is said to terminate globally when it has terminated locally and all activities that it may have spawned at any place (if any) have, recursively, terminated globally.

An X10 computation is initiated as a single activity from the command line. This activity is the *root activity* for the entire computation. The entire computation terminates when (and only when) this activity globally terminates. Thus X10 does not permit the creation of so called "daemon threads" – threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§ 7.4).

Future Extensions. *We may permit the initial activity to be a daemon activity to permit reactive computations, such as webservers, that may not terminate.*

7.1. The X10 rooted exception model

The rooted nature of X10 computations permits the definition of a *rooted* exception model. In multi-threaded programming languages there is a natural parent-child relationship between a thread and a thread that it spawns. Typically the parent thread continues execution in parallel with the child thread. Therefore the parent thread cannot serve to catch any exceptions thrown by the child thread.

The presence of a root activity permits X10 to adopt a different model. In any state of the computation, say that an activity A is *a root of* an activity B if A is an ancestor of B and A is suspended at a statement (such as the `finish` statement § 7.3) awaiting the termination of B (and possibly other activities).

[1] An indexical constant is one whose value depends on its context of use.

For every X10 computation, the `root-of` relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B, the path from A to B is called the *activation path* for the activity.[1]

We may now state the exception model for X10. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the `root-of` tree when the activity terminates (based on the semantics of the statement being executed by the activity).[2] Thus, unlike concurrent languages such as Java no exception is "thrown on the floor".

7.2. Spawning an activity

Asynchronous activities serve as a single abstraction for supporting a wide range of concurrency constructs such as message passing, threads, DMA, streaming, data prefetching. (In general, asynchronous operations are better suited for supporting scalability than synchronous operations.)

An activity is created by executing the statement:

```
463 Statement ::= AsyncStatement
473 StatementNoShortIf ::= AsyncStatementNoShortIf
481 AsyncStatement ::=
     async PlaceExpressionSingleListopt Statement
491 AsyncStatementNoShortIf ::=
     async PlaceExpressionSingleListopt
       StatementNoShortIf
524   PlaceExpressionSingleListopt ::=
525       | PlaceExpressionSingleList
499   PlaceExpressionSingleList ::=
        ( PlaceExpression )
500   PlaceExpression ::= Expression
```

The place expression `e` is expected to be of type `place`, e.g. `here` or `place.FIRST_PLACE` or `d[p]` for some distribution `d` and point `p` (§ 6). If not, the compiler replaces `e` with `e.location`. (Recall that every expression in X10 has a type; this type is a subtype of the root class `x10.lang.Object`. This class has a field `location` of type `place` recording the place at which the value resides. See the documentation for `x10.lang.Object`.)

Note specifically that the expression `a[i]` when used as a place expression will evaluate to `a[i].location`, which may not be the same place as `a.distribution[i]`. The programmer must be careful to choose the right expression, appropriate for the statement. Accesses to `a[i]` within `Statement` should typically be guarded by the place expression `a.distribution[i]`.

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. (The place must be such that the statement can be executed safely, without generating a `BadPlaceException`.) In such cases the programmer may omit the place designator; the compiler will throw an error if it cannot determine the unique designated place.[3]

An activity A executes the statement `async (P) S` by launching a new activity B at the designated place, to execute the specified statement. The statement terminates locally as soon as B is launched. The activation path for B is that of A, augmented with information about the line number at which B was spawned. B terminates normally when S terminates normally. It terminates abruptly if S throws an (uncaught) exception. The exception is propagated to A if A is a root activity (see § 7.3), otherwise through A to A's root activity. Note that while an activity is running, exceptions thrown by activities it has already generated may propagate through it up to its root activity.

Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the pool of activities at the target place and will be executed in sequence or in parallel based on the local scheduler's decisions. If the programmer wishes to sequence their execution s/he must use X10 constructs, such as clocks and `finish` to obtain the desired effect. Further, the X10 implementations are not required to have fair schedulers, though every implementation should make a best faith effort to ensure that every activity eventually gets a chance to make forward progress.

Static semantics. The statement in the body of an `async` 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, which throws no checked exceptions. As such, it may reference variables in lexically enclosing scopes (including `clock` variables, § 8) provided that such variables are (implicitly or explicitly) `final`.

7.3. Finish

The statement `finish S` converts global termination to local termination and introduces a root activity.

```
468 Statement ::= FinishStatement
478 StatementNoShortif ::=
     FinishStatementNoShortIf
488 FinishStatement ::= finish Statement
498 FinishStatementNoShortIf ::=
     finish StatementNoShortIf
```

An activity A executes `finish s` by executing `s`. The execution of `s` may spawn other asynchronous activities (here or at other places). Uncaught exceptions thrown or propagated by any activity spawned by `s` are accumulated at `finish s`.

[1] Note that depending on the state of the computation the activation path may traverse activities that are running, suspended or terminated.

[2] In X10 v1.01 the `finish` statement is the only statement that marks its activity as a root activity. Future versions of the language may introduce more such statements.

[3] X10 v1.01 does not specify a particular algorithm; this will be fixed in future versions.

`finish s` terminates locally when all activities spawned by `s` terminate globally (either abruptly or normally). If `s` terminates normally, then `finish s` terminates normally and A continues execution with the next statement after `finish s`. If `s` terminates abruptly, then `finish s` terminates abruptly and throws a single exception formed from the collection of exceptions accumulated at `finish s`.

Thus a `finish s` statement serves as a collection point for uncaught exceptions generated during the execution of `s`.

Note that repeatedly `finish`ing a statement has no effect after the first `finish`: `finish finish s` is indistinguishable from `finish s`.

Interaction with clocks. `finish s` interacts with clocks (§ 8).

While executing `s`, an activity must not spawn any `clocked` asyncs. (Asyncs spawned during the execution of `s` may spawn clocked asyncs.) A `ClockUseException` is thrown if (and when) this condition is violated.

In X10 v1.01 this condition is checked dynamically; future versions of the language will introduce type qualifiers which permit this condition to be checked statically.

Future Extensions. *The semantics of* `finish S` *is conjunctive; it terminates when all the activities created during the execution of* `S` *(recursively) terminate. In many situations (e.g. nondeterministic search) it is natural to require a statement to terminate when any* one *of the activities it has spawned succeeds. The other activities may then be safely aborted. Future versions of the language may introduce a* `finishone S` *construct to support such speculative or nondeterministic computation.*

7.4. Initial activity

An X10 computation is initiated from the command line on the presentation of a classname `C`. The class must have a `public static void main(String[] a)` method, otherwise an exception is thrown and the computation terminates. The single statement

```
finish async (place.FIRST_PLACE) {
  C.main(s);
}
```

is executed where `s` is an array of strings created from command line arguments. This single activity is the root activity for the entire computation. (See § 6 for a discussion of placs.)

7.5. Asynchronous Expression and Futures

X10 provides syntactic support for *asynchronous expressions*, also known as futures:

```
511  Primary ::= FutureExpression
515  FutureExpression ::=
       future PlaceExpressionSingleListopt
         Expression
```

Intuitively such an expression evaluates its body asynchronously at the given place. The resulting value may be obtained from the future returned by this expression, by using the `force` operation.

In more detail, in an expression `future(Q)e`, the place expression `Q` is treated as in an `async` statement. `e` is an expression of some type `T`. `e` may reference only those variables in the enclosing lexical environment which are declared to be `final`.

If the type of `e` is `T` then the type of `future (Q)e` is `future<T>`. This type `future<T>` is defined as if by:

```
public interface future<T> {
  T force();
  boolean forced();
}
```

(Here we use the syntax for generic classes. X10 v1.01 does not support generic classes in their full generality. In particular, the user may not define generic classes. This is reserved for future extensions to the language.)

Evaluation of `future (Q)e` terminates locally with the creation of a value `f` of type `future<T>`. This value may be stored in objects, passed as arguments to methods, returned from method invocation etc.

At any point, the method `forced` may be invoked on `f`. This method returns without blocking, with the value `true` if the asynchronous evaluation of `e` has terminated globally and with the value `false` if it has not.

The method invocation `force` on `f` blocks until the asynchronous evaluation of `e` has terminated globally. If the evaluation terminates successfully with value `v`, then the method invocation returns `v`. If the evaluation terminates abruptly with exception `z`, then the method throws exception `z`. Multiple invocations of `force` (by this or any other activity) do not result in multiple evaluations of `e`. The results of the first evaluation are stored in the future `f` and used to respond to all `force` queries.

```
future<T> promise
    = future (a.distribution[3]) { a[3] };
T value = promise.force();
```

7.5.1. Implementation notes

Futures are provided in X10 for convenience; they may be programmed using latches, `async` and `finish` as described in § 7.6.3.

7.6. Atomic blocks

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 block*.

A programmer may use atomic blocks 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.

7.6.1. Unconditional atomic blocks

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

```
461 Statement ::= AtomicStatement
474 StatementNoShortIf ::=
      AtomicStatementNoShortIf
482 AtomicStatement ::= atomic  Statement
492 AtomicStatementNoShortIf ::=
      atomic StatementNoShortIf
445 MethodModifier ::= atomic
```

For the sake of efficient implementation X10 v1.01 requires that the atomic block be *analyzable*, that is, the set of locations that are read and written by the `BlockStatement` are bounded and determined statically.[4] The exact algorithm to be used by the compiler to perform this analysis will be specified in future versions of the language.

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 block within a `try/finally` clause and include undo code in the finally clause. Thus the `atomic` statement only guarantees atomicity on successful execution, not on a faulty execution.

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 an `atomic` statement.

Atomic blocks are closely related to non-blocking synchronization constructs [7], and can be used to implement non-blocking concurrent algorithms.

Static semantics and dynamic checks. In `atomic s`, `s` may include method calls, conditionals etc. It may *not* include an `async` activity. It may *not* include any statement that may potentially block at runtime (e.g. `when`, `force` operations, `next` operations on clocks, `finish`).

Limitation: Not checked in the current implementation.

All locations accessed in an atomic block must reside `here` (§ 6.0.2). A `BadPlaceException` is thrown if (and when) this condition is violated.

[4] A static bound is a constant that depends only on the program text, and is independent of any runtime parameters.

Consequences. Note an important property of an (unconditonal) atomic block:

$$\texttt{atomic}\,\{\texttt{s1 atomic s2}\} \;=\; \texttt{atomic}\,\{\texttt{s1 s2}\} \quad (7.1)$$

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

Example

The following class method implements a (generic) compare and swap (CAS) operation:

```
// target defined in lexically enclosing environment.
public atomic boolean CAS( Object old,
                           Object new) {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

7.6.2. Conditional atomic blocks

Conditional atomic blocks are of the form:

```
465 Statement ::= WhenStatement
475 StatementNoShortIf ::= WhenStatementNoShortIf
483 WhenStatement ::=
        when ( Expression ) Statement
484   | WhenStatement
        or ( Expression ) Statement
```

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.

X10 does not guarantee that a conditional atomic block will execute if its condition holds only intermmitently. For, based on the vagaries of the scheduler, the precise instant at which a condition holds may be missed. Therefore the programmer is advised to ensure that conditions being tested by conditional atomic blocks are eventually stable, i.e. they will continue to hold until the block executes (the action in the body of the block may cause the condition to not hold any more).

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

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. Finally note that a `when` statement with multiple branches is behaviorally identical to a `when` statement with a single branch that checks the disjunction of the condition of each branch, and whose body contains an `if/then/else` checking each of the branch conditions.

Static semantics. For the sake of efficient implementation certain restrictions are placed on the guards and statements in a conditional atomic block.

Guards are required not to have side-effects, not to spawn asynchronous activities and to have a statically determinable upper bound on their execution. These conditions are expected to be checked statically by the compiler.

The body of a `when` statement must satisfy the conditions for the body of an `atomic` block.

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

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.

7.6.3. Examples

Bounded buffer. 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 {
  nullable Object datum = null;
  boolean filled = false;
  public
    void send(Object v) {
      when ( !filled ) {
        this.datum = v;
        this.filled = true;
    }
  }
  public
    Object receive() {
      when ( filled ) {
        Object v  = datum;
        datum = null;
        filled = false;
        return v;
      }
    }
  }
```

Implementing a future with a latch. The following class shows how to implement a *latch*. A latch is an object that is initially created in a state called the *unlatched* state. During its lifetime it may transition once to a *forced* state. Once forced, it stays forced for its lifetime. The latch may be queried to determine if it is forced, and if so, an associated value may be retrieved. Below, we will consider a latch set when some activity invokes a `setValue` method on it. This method provides two values, a normal value and an exceptional value. The method `force` blocks until the latch is set. If an exceptional value was specified when the latch was set, that value is thrown on any attempt to read the latch. Otherwise the normal value is returned.

```
public interface future {
   boolean forced();
   Object force();
}
public class Latch implements future {
  protected boolean forced = false;
  protected nullable boxed result = null;
  protected nullable exception z = null;

  public atomic
   boolean setValue( nullable Object val ) {
   return setValue( val, null);
    }
   public atomic
   boolean setValue( nullable exception z ) {
       return setValue( null, z);
    }
    public atomic
    boolean setValue( nullable Object val,
                   nullable exception z ) {
       if ( forced ) return false;
       // these assignment happens only once.
       this.result = val;
       this.z = z;
       this.forced = true;
       return true;
    }
    public atomic boolean forced() {
       return forced;
    }
    public Object force() {
       when ( forced ) {
          if (z != null) throw z;
          return result;
       }
    }
}
```

Latches, `aync` operations and `finish` operations may be used to implement futures as follows. The expression `future(P){e}` can be translated to:

```
new RunnableLatch() {
   public Latch run() {
      Latch L = new Latch();
      async ( P ) {
```

```
        Object X;
        try {
            finish X = e;
            async ( L ) {
               L.setValue( X );
            }
        } catch ( exception Z ) {
           async ( L ) {
             L.setValue( Z );
           }
        }
      }
      return l;
    }
  }.run()
```

Here we assume that `RunnableLatch` is an interface defined by:

```
public interface RunnableLatch {
   Latch run();
}
```

We use the standard Java idiom of wrapping the core translation inside an inner class definition/method invocation pair (i.e. `new RunnableLatch() .....run()`) so as to keep the resulting expression completely self-contained, while executing statements inside the evaluation of an expression.

Execution of a `future(P) {e}` causes a new latch to be created, and an `async` activity spawned at `P`. The activity attempts to `finish` the assigned `x = e`, where `x` is a local variable. This may cause new activities to be spawned, based on `e`. If the assignment terminates successfully, another activity is spawned to invoke the `setValue` method on the latch. Exceptions thrown by these activities (if any) are accumulated at the `finish` statement and thrown after global termination of all activities spawned by `x=e`. The exception will be caught by the `catch` clause and stored with the latch.

A future to execute a statement. Consider an expression `onFinish {S}`. This should return a `boolean` latch which should be forced when `S` has terminated globally. Unlike `finish S`, the evaluation of `onFinish {S}` should locally terminate immediately, returning a latch. The latch may be passed around in method invocations and stored in objects. An activity may perform `force`/`forced` method invocations on the latch whenever it desires to determine whether `S` has terminated.

Such an expression can be written as:

```
new RunnableLatch() {
   public Latch run() {
      Latch L = new Latch();
      async ( here ) {
         try {
             finish S;
             L.setValue( true );
         } catch ( exception Z ) {
              L.setValue( Z );
         }
      }
      return L;
   }
}.run()
```

7.7. Iteration

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

```
189 Statement ::= ForStatement
206 StatementNoShortIf ::=
     ForStatementNoShortIf
236 ForStatement ::= EnhancedForStatement
239 ForStatementNoShortIf ::=
     EnhancedForStatementNoShortIf
466 Statement ::= ForEachStatement
476 StatementNoShortIf ::=
     ForEachStatementNoShortIf
487 EnhancedForStatement ::=
      for ( FormalParameter : Expression )
        Statement
487 EnhancedForStatementNoShortIf ::=
      for ( FormalParameter : Expression )
         StatementNoShortIf
485  ForEachStatement ::=
      foreach ( FormalParameter : Expression )
         Statement
495  ForEachStatementNoShortIf ::=
       foreach ( FormalParameter : Expression )
         StatementNoShortIf
```

In both statements, the expression is intended to be of type `region`. Expressions `e` of type `distribution` and `array` are also accepted, and treated as if they were `e.region`. The compiler throws a type error in all other cases.

The formal parameter must be of type `point`. Exploded syntax may be used (§ 12.3). The parameter is considered implicitly final, as are all the exploded variables.

An activity executes a `for` statement by enumerating the points in the region in canonical order. The activity executes the body of the loop with the formal parameter(s) bound to the given point. If the body locally terminates successfully, the activity continues with the next iteration, terminating successfully when all points have been visited. If an iteration throws an exception then the for statement throws an exception and terminats abruptly.

An activity executes a `foreach` statement in a similar fashion except that separate `async` activities are launched in parallel in the local place for each point in the region. The statement terminates locally when all the activities have been spawned. It never throws an exception, though exceptions thrown by the spawned activities are propagated through to the root activity.

In a similar fashion we introduce the syntax:

```
467 Statement ::= AtEachStatement
477 StatementNoShortIf ::=
     AtEachStatementNoShortIf
486 AtEachStatement ::=
```

```
      ateach ( FormalParameter : Expression )
         Statement
496 AtEachStatementNoShortIf ::=
      ateach ( FormalParameter : Expression )
         StatementNoShortIf
```

Here the expression is intended to be of type `distribution`. Expressions `e` of type `array` are also accepted, and treated as if they were `e.distribution`. The compiler throws a type error in all other cases. This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, `ateach( point p[i1,...,ik]: A) S` may be thought of as standing for:

```
foreach (point p[i1,...,ik] : A)
  async (A.distribution[p]) {S}
```

8. 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. Clocks are a generalization of *barriers*. They permit dynamically created activities to register and deregister. An activity may be registered with multiple clocks at the same time. In particular, nested clocks are permitted: an activity may create a nested clock and within one phase of the outer clock schedule activities to run to completion on the nested clock. Neverthless the design of clocks ensures that deadlock cannot be introduced by using clock operations, and that clock operations do not introduce any races.

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

The key invariants associated with clocks are as follows. At any stage of the computation, a clock has zero or more *registered* activities. An activity may perform operations only on those clocks it is registered with (these clocks constitute its *clock set*). An activity is registered with one or more clocks 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 register itself with a clock it discovers by reading a data-structure.

An activity may perform the following operations on a clock `c`. It may *unregister* with `c` by executing `c.drop();`. After this, it may perform no further actions on `c` for its lifetime. It may *check* to see if it is unregistered on a clock. It may *register* a newly forked activity with `c`. It may *resume* the clock by executing `c.resume();`. This indicates to `c` that it has finished posting all statements it wishes to perform in the current phase. Finally, it may *block* (by executing `next;`) on all the clocks that it is registered with. (This operation implicitly `resume`'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 `resume` operation on that clock and all statements posted 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 resumes 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 resumed. But this implies that all activities blocked on `next` may continue and the configuration is not stuck. The only other possibility is that an activity may be stuck on `finish`. But the interaction rule between `finish` and clocks (§ 7.3) guarantees that this cannot cause a cycle in the wait-for graph. A more rigorous proof may be found in [9].

8.1. Clock operations

The special statements introduced for clock operations are listed below.

```
462 Statement ::= ClockedStatement
472 StatementNoShortIf ::=
      ClockedStatementNoShortIf
480 ClockedStatement ::=
      clocked ( ClockList ) Statement
490 ClockedStatementNoShortIf ::=
      clocked ( ClockList )
        StatementNoShortIf
501 NextStatement ::= next ;
```

Note that `x10.lang.clock` provides several useful methods on clocks (e.g. `drop`).

8.1.1. Creating new clocks

Clocks are created using the nullary constructor for `x10.lang.clock` via a factory method:

```
clock timeSynchronizer = clock.factory.clock();
```

All clocked variables are implicitly final. The initializer for a local variable declaration of type `clock` must be a new clock expression. Thus X10 does not permit aliasing of clocks. Clocks are created in the place global heap and hence outlive the lifetime of the creating activity. Clocks are instances of value classes, hence may be freely copied from place to place. (Clock instances typically contain references to mutable state that maintains the current state of the clock.)

The current activity is automatically registered with the newly created clock. It may deregister using the `drop` method on clocks (see the documentation of `x10.lang.clock`). All activities are automatically deregistered from all clocks they are registered with on termination (normal or abrupt).

8.1.2. Registering new activities on clocks

The programmer may specify which clocks a new activity is to be registered with using the `clocked` clause.

An activity may transmit only those clocks that is registered with and has not quiesced on (§ 8.1.3). A `ClockUseException` is thrown if (and when) this condition is violated.

An activity may check that it is registered on a clock `c` by executing:

```
c.registered()
```

This call returns the `boolean` value `true` iff the activity is registered on `c`; otherwise it returns `false`.

Note. X10 does not contain a "register" statement that would allow an activity to discover a clock in a datastructure and register itself on it. Therefore, while clocks may be stored in a datastructure by one activity and read from that by another, the new activity cannot "use" the clock unless it is already registered with it.

8.1.3. Resuming clocks

X10 permits *split phase* clocks. An activity may wish to indicate that it has completed whatever work it wishes to perform in the current phase of a clock `c` it is registered with, without suspending all activity. It may do so by executing the method invocation:

```
c.resume();
```

An activity may invoke this method only on a clock it is registered with, and has not yet dropped (§ 8.1.5). A `ClockUseException` is thrown if (and when) this condition is violated. Nothing happens if the activity has already invoked a `resume` on this clock in the current phase. Otherwise execution of this statement indicates that the activity will not transmit `c` to an `async` (through a `clocked` clause), until it terminates, drops `c` or executes a `next`.

8.1.4. Advancing clocks

An activity may execute the statement

```
next;
```

Execution of this statement blocks until all the clocks that the activity is registered with (if any) have advanced. (The activity implicitly issues a `resume` 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 resumed `c`. Note that once a computation 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.

8.1.5. Dropping clocks

An activity may drop a clock by executing:

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

The activity is no longer considered registered with this clock. A `ClockUseException` is thrown if the activity has already dropped `c`.

8.1.6. Program equivalences

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

$$\texttt{c.resume();next;} = \texttt{next;} \quad (8.1)$$

$$\texttt{c.resume();d.resume();} = \texttt{d.resume();c.resume();} \quad (8.2)$$

$$\texttt{c.resume();c.resume();} = \texttt{c.resume();} \quad (8.3)$$

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

9. Interfaces

X10 v1.01 interfaces are essentially the same Java interfaces [6, §9]. An interface primarily specifies signatures for public methods. It may extend multiple interfaces.

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.

10. 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. X10 does not permit mutable static state, so the role of static methods and initializers is quite limited. Instead programmers should use singleton classes to carry mutable static state.

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.

10.1. 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 v1.01 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.

10.2. 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` (§ 3.5.1), 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. For brevity, the X10 compiler allows the programmer to omit the keyword `class` after `value` in a value class declaration.

```
447  ClassDeclaration ::= ValueClassDeclaration
448  ValueClassDeclaration ::=
       ClassModifiersopt value identifier Superopt
         Interfacesopt ClassBody
449  | ClassModifiersopt value class identifier
         Superopt Interfacesopt ClassBody
```

The `nullable` type-constructor (§ 3.2) can be used to declare variables whose value may be `null` or a value type.

Stable equality for value types is defined through a deep walk, bottoming out in fields of reference types (§ 12.4.3).

Static semantics. 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. All fields of a value class are implicitly declared `final`.

10.2.1. 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).

10.2.2. Example

A functional `LinkedList` program may be written as follows:

```
value LinkedList  {
  Object first;
  nullable LinkedList rest;
  public
     LinkedList(Object first) {
     this(first, null);
  }
  public
    LinkedList(Object first,
               nullable LinkedList rest) {
    this.first = first;
    this.rest = rest;
  }
```

```
  public
    Object first() {
    return first;
  }
  public
    nullable LinkedList rest() {
    return rest;
  }
  public
    LinkedList append(LinkedList l) {
    return (this.rest == null)
        ? new LinkedList(this.first, l)
        : this.rest.append(l);
  }
}
```

Similarly, a `Complex` class may be implemented as follows:

```
value Complex  {
  double re;
  double im;
  public
     Complex(double re, double im) {
     this.re=re;
     this.im=im;
  }
  public Complex add(Complex other) {
    return new Complex(this.re+other.re,
                       this.im+other.im);
  }
  public Complex mult(Complex other) {
    return new Complex(this.re^2-other.re^2,
                       2*this.im*other.im);
  }
  ...
}
```

10.3. Method annotations

10.3.1. `atomic` annotation

A method may be declared `atomic`.

```
445   MethodModifier ::= atomic
```

Such a method is treated as if the statement in its body is wrapped implicitly in an `atomic` statement.

11. 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 through the field `a.distribution`.

11.1. 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 v1.01; 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`. Regions are first-class objects – they may be stored in fields of objects, passed as arguments to methods, returned from methods etc.

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

Here are several examples of region declarations:

```
region Null = [];  // Empty 0-dimensional region
region R1 = 1:100; // 1-dim region with extent 1..100.
region R1 = [1:100]; // Same as above.
region R2 = [0:99, -1:MAX_HEIGHT];
region R3 = region.factory.upperTriangular(N);
region R4 = region.factory.banded(N, K);
   // A square region.
region R5 = [E, E];
   // Same region as above.
region R6 = [100, 100];
   // 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 regions (§ 12.2) within square brackets, as above and represents the Cartesian product of each of the arguments. The bound of a dimension may be any final variable of a fixed-point numeric type. X10 v1.01 does not support hierarchical regions.

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

- `region.factory.upperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular `N x N` matrix.
- `region.factory.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` (§ 7.7) 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.

11.1.1. 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 equal (==) if they represent the same set of points.

11.2. 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 v1.01; 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 = [1:100]
distribution D = distribtion.factory.block(R);
distribution D = distribution.factory.cyclic(R);
distribution D = R -> here;
distribution D = distribution.factory.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`. The evaluation of the expression `D[p]` throws an `ArrayIndexOutofBoundsException` if `p` does not lie in the underlying region.

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`.

11.2.1. 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 may be performed on `distribution.factory`.

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 as follows. Let p equal `|R| div N` and q equal `|R| mod N`, where `N` is the size of `Q`, and `|R|` is the size of `R`. 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 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`.

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 in D.region. 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. An implementation may find a way to still represent them in contiguous memory, e.g. using a complex arithmetic function to map from the region index to an index into the array.

11.2.2. User-defined distributions

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

11.2.3. 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.overlay(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 D2[p] if p lies in D.domain otherwise it is D1[p]. (D1 provides the defaults.)

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.overlay(D2) (or equivalently D2.overlay(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.

11.2.4. Example

```
    double[D] dotProduct(T[D] a, T[D] b) {
    return (new T[1:D.places] (point j) {
        return (new T[D | here] (point i) {
            return a[i]*b[i];
        }).sum();
    }).sum();
}
```

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.

11.3. Array initializer

```
450 ArrayCreationExpression ::=
      new ArrayBaseType Unsafeopt []
        ArrayInitializer
451   | new ArrayBaseType Unsafeopt
      [ Expression ]
452   | new ArrayBaseType Unsafeopt
      [ Expression ] Expression
453   | new ArrayBaseType Unsafeopt
      [ Expression ]
      (FormalParameter) MethodBody
454   | new ArrayBaseType value Unsafeopt
      [ Expression ]
455   | new ArrayBaseType value Unsafeopt
      [ Expression ] Expression
456   | new ArrayBaseType value Unsafeopt
      [ Expression ]
      ( FormalParameter ) MethodBody
457 ArrayBaseType ::= PrimitiveType
458   | ClassOrInterfaceType
530 Unsafeopt ::=
531   | unsafe
```

An array may be declared unsafe if it is intended to be allocated in an unmanaged region (e.g. for communication with native code). A value array is an immutable array. An array creation must take either an int as an argument or a distribution. In the first case an array is created over the distribution [0:N-1]-> here; in the second over the given distribution.

An array creation operation may also specify an initializer using the abbreviated formalparameter/methodbody functional syntax. The formal parameter may contain exploded parameters (Section 12.3). The function is applied in parallel at all points in the domain of the distribution. The array construction operation terminates locally only when the array has been fully created and initialized (at all places in the range of the distribution).

For instance:

```
int[.] data
    = new int[1000->here]
        new intArray.pointwiseOp(){
            public int apply(point p[i]){
              return i;
            }
     };
int[.] data2
    = new int value [[1:1000,1:1000]->here]
        (point p[i,j]){ return i*j; };
```

The first declaration stores in data a reference to a mutable array with 1000 elements each of which is located in the same place as the array. Each array component is initialized to i.

The second declaration stores in `data2` an (immutable) 2-d array over `[1:1000, 1:1000]` initialized with `i*j` at point `[i,j]`. It uses a more abbreviated form to specify the array initializer function.

Other examples:

```
int[.] data
   = new int[1000](point [i]){return i*i;};
float[D] d
   = new float[D] (point [i]){return 10.0*i;};
float[D] result
   = new float[D]
     (point [i,j]) {return i + j; };
```

11.4. 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.

11.4.1. 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. The operator assignments `a[i] op= e` are also available in X10.

11.4.2. Constant promotion

For a distribution `D` and a constant or final variable `v` of type `T` the expression `new T[D](point p) { return v;}` `D v` denotes the mutable array with distribution `D` and base type `T` initialized with `v`.

11.4.3. 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 (§ 11.2) to obtain sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

11.4.4. 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.overlay(a2)` (read: the array `a1` *overlaid with* `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 `D2[p]` if that is defined and to `D1[p]` otherwise. The value `a1.overlay(a2)[p]` is `a2[p]` if it is defined and `a1[p]` otherwise.

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

The expression `a1.update(a2)` updates the array `a1` in place with the result of `a1.overlay(a2)`.

11.4.5. 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`, using the `lift` operation, `a.lift(f, b)`.

Reductions Let `f` be a `binaryOp` defined on type `T` (e.g. see the specification of the classes `x10.lang.intArray`). Let `a` be a value or reference array over base type `T`. Then the operation `a.reduce(f)` returns a value of type `T` obtained by performing `m` on all points in `a` in some order, and in parallel.

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 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.

Other operations on arrays may be found in `x10.lang.intArray` and other related classes.

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

X10 supports assignment `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. Point and region construction

X10 specifies a simple syntax for the construction of points and regions.

```
281   ArgumentList ::= Expression
282      | ArgumentList , Expression
512   Primary ::= [ ArgumentList ]
```

Each element in the argument list must be either of type `int` or of type `region`. In the former case the expression `[ a1,..., ak ]` is treated as syntactic shorthand for

```
point.factory.point(a1,..., ak)
```

and in the latter case as shorthand for

```
region.factory.region(a1,..., ak)
```

12.3. Exploded variable declarations

X10 permits a richer form of specification for variable declarators in method arguments, local variables and loop variables (the "exploded" or *destructuring* syntax).

```
81    VariableDeclaratorId ::=
          identifier [ IdentifierList ]
82      | [ IdentifierList ]
```

In X10 v1.01 the `VariableDeclaratorId` must be declared at type `x10.lang.point`. Intuitively, this syntax allows a point to be "destructured" into its corresponding `int` indices in a pattern-matching style. The kth identifier in the `IdentifierList` is treated as a `final` variable of type `int` that is initialized with the value of the kth index of the point. The second form of the syntax (Rule 82) permits the specification of only the index variables.

Future versions of the language may allow destructuring syntax for all value classes.

Example. The following example succeeds when executed.

```
public class Array1Exploded {
  public int select(point p[i,j], point [k,l]) {
      return i+k;
  }
  public boolean run() {
    distribution d =  [1:10, 1:10] -> here;
    int[.] ia = new int[d];
    for(point p[i,j]: [1:10,1:10]) {
        if(ia[p]!=0) return false;
        ia[p] = i+j;
    }
    for(point p[i,j]: d) {
      point q1 = [i,j];
      if (i != q1[0]) return false;
      if ( j != q1[1]) return false;
      if(ia[i,j]!= i+j) return false;
      if(ia[i,j]!=ia[p]) return false;
      if(ia[q1]!=ia[p]) return false;
   }
    if (! (4 == select([1,2],[3,4]))) return false;
     return true;
   }

  public static void main(String args[]) {
     boolean b= (new Array1Exploded()).run();
     System.out.println("++++++ "
                        + (b? "Test succeeded."
                           :"Test failed."));
     System.exit(b?0:1);
  }
}
```

12.4. Expressions

X10 inherits all the standard expressions of Java [6, § 15] – as modified to permit generics [5] – 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.

12.4.1. The classcast operator

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

```
306 UnaryExpressionNotPlusMinus ::=
     CastExpression
506 CastExpression ::=
     ( Type ) UnaryExpressionNotPlusMinus
```

The result of this operation is a value of the given type if the cast is permissible at runtime. Both the data type and place type of the value are checked. Data type conversion is checked according to the rules of the Java language (e.g.[6, §5.5]). If the value cannot be cast to the appropriate data type, a `ClassCast` exception is thrown. Otherwise, if the value cannot be cast to the appropriate place type a `BadPlaceException` is thrown.

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.

12.4.2. `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`.

12.4.3. 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 (even if the object has no mutable fields).

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.

The predicates == and != may not be overridden by the programmer.

13. Annotations and Compiler Plugins

X10 provides an an annotation system and compiler plugin system for to allow the compiler to be extended with new static analyses and new transformations.

Annotations are interface types that decorate the abstract syntax tree of an X10 program. The X10 type-checker ensures that an annotation is a legal interface type. In X10, interfaces may declare both methods and properties. Therefore, like any interface type, an annotation may instantiate one or more of its interface's properties. Unlike with Java annotations, property initializers need not be compile-time constants; however, a given compiler plugin may do additional checks to constrain the allowable initializer expressions. The X10 type-checker does not check that all properties of an annotation are initialized, although this could be enforced by a compiler plugin.

13.1. Annotation syntax

The annotation syntax consists of an "@" followed by an interface type.

```
532  Annotation ::= @ InterfaceType
533  Annotations ::= Annotation
534    | Annotations Annotation
535  Annotationsopt ::=
536    | Annotations
```

Annotations can be applied to most syntactic constructs in the language including class declarations, constructors, methods, field declarations, local variable declarations and formal parameters, statements, expressions, and types. Multiple occurrences of the same annotation (i.e., multiple annotations with the same interface type) on the same entity are permitted.

```
537  ClassModifier ::= Annotation
538  InterfaceModifier ::= Annotation
539  FieldModifier ::= Annotation
540  MethodModifier ::= Annotation
541  VariableModifier ::= Annotation
542  ConstructorModifier ::= Annotation
543  AbstractMethodModifier ::= Annotation
544  ConstantModifier ::= Annotation
545  Type ::= AnnotatedType
546  AnnotatedType ::= Type Annotations
547  Statement ::= AnnotatedStatement
548  AnnotatedStatement ::= Annotation Statement
549  Expression ::= AnnotatedExpression
550  AnnotatedExpression ::= ( Annotations ) Expression
```

Recall that interface types may have dependent parameters.

The following examples illustrate the syntax:

- Declaration annotations:

```
// class annotation
@Value
class Cons { ... }

// method annotation
```

```
@PreCondition(0 <= i && i < this.size)
public Object get(int i) { ... }

// constructor annotation
@Where(x != null)
C(T x) { ... }

// constructor return type annotation
C@Initialized(T x) { ... }

// variable annotation
@Unique A x;
```

- Type annotations:

```
List@Nonempty

int@Range(1,4)

double[][]@Size(n * n)
```

- Expression annotations:

```
(@RemoteCall) m()

3 == (@Bits(2)) 15
```

- Statement annotations:

```
@Atomic { ... }

@MinIterations(1)
@MaxIterations(n)
for (int i = 0; i < n; i++) { ... }

// An annotated empty statement ;
@Assert(x < y);
```

13.2. Annotation declarations

Annotations are declared as interfaces. They must be subtypes of `x10.lang.annotation.Annotation`. Annotations on types, expressions, statements, classes, fields, methods, constructors, and local variable declarations (or formal parameters) must extend `ExpressionAnnotation`, `StatementAnnotation`, `ClassAnnotation`, `FieldAnnotation`, `MethodAnnotation`, `ConstructorAnnotation`, and `VariableAnnotation`, respectively.

13.3. Compiler plugins

After the base X10 semantic checking is completed, compiler plugins are loaded and run. Plugins may perform any number of compiler passes to implement additional semantic checking and code transformations, including transformations using the abstract syntax of the annotations themselves. Plugins should output valid X10 abstract syntax trees.

Plugins are implemented in java as Polyglot [**?**] passes applied to the AST after normal base X10 type checking. Plugins to run are specified on the command-line. The order of execution is determined by the Polyglot pass scheduler.

To run compiler plugins, add the command-line option:

```
-PLUGINS=P1,P2,...,Pn
```

where `P1`, `P2`, ..., `Pn` are classes that implement the `CompilerPlugin` interface:

```
package polyglot.ext.x10.plugin;

import polyglot.ext.x10.ExtensionInfo;
import polyglot.frontend.Job;
import polyglot.frontend.goals.Goal;

public interface CompilerPlugin {
    public Goal register(ExtensionInfo extInfo, Job job);
}
```

The `Goal` object returned by the `register` method specifies dependencies on other passes. Documentation for Polyglot can be found at:

```
http://www.cs.cornell.edu/Projects/polyglot
```

Most plugins should implement either `SimpleOnePassPlugin` or `SimpleVisitorPlugin`.

The compiler loads plugin classes from the x10c classpath.

Plugins are given access to a Polyglot AST and type system. Annotations are represented in the AST as `Nodes` with the following interface:

```
package polyglot.ext.x10.ast;

public interface AnnotationNode extends Node {
  X10ClassType annotation();
}
```

Annotations for a `Node` object `n` can be accessed through the node's extension object as follows:

```
List<AnnotationNode> annotations =
  ((X10Ext) n.ext()).annotations();
List<X10ClassType> annotationTypes =
  ((X10Ext) n.ext()).annotationInterfaces();
```

In the type system, `X10TypeObject` has the following method for accessing annotations:

```
List<X10ClassType> annotations();
```

BIBLIOGRAPHY

[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 Yelick. 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] P. Charles, C. Grothoff, C. Donawa, K. Ebcioglu, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA*, 2005. To appear.

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

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

[7] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[8] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.

[9] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur '05, to appear*, 2005.

[10] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[11] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.