Report on the Programming Language X10

Version 2.0.4

DRAFT — June 16, 2010

Vijay Saraswat and Bard Bloom

Please send comments to `bardb@us.ibm.com`

June 16, 2010

This report provides a description of the programming language X10. X10 is a class-based object-oriented 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.

The X10 design team consists of Bard Bloom, Ganesh Bikshandi, David Cunningham, Robert Fuhrer, David Grove, Sreedhar Kodali, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Olivier Tardieu.

Past members include Shivali Agarwal, David Bacon, Raj Barik, Bob Blainey, Philippe Charles, Perry Cheng, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Patrick Gallop, Christian Grothoff, Allan Kielstra, Sriram Krishnamoorthy, Bruce Lucas, Vivek Sarkar, Armando Solar-Lezama, S. Alexander Spoon, Sayantan Sur, Christoph von Praun, Pradeep Varma, Krishna Nandivada Venkata, Jan Vitek, and Tong Wen.

For extended discussions and support we would like to thank: Gheorghe Almasi, Robert Blackmore, Robert Callahan, Calin Cascaval, Norman Cohen, Elmootaz Elnozahy, John Field, Kevin Gildea, Chulho Kim, Orren Krieger, Doug Lea, John McCalpin, Paul McKenney, Andrew Myers, Filip Pizlo, Ram Rajamony, R. K. Shyamasundar, V. T. Rajan, Frank Tip, Mandana Vaziri, and Hanhong Xue.

We thank Jonathan Rhees and William Clinger with help in obtaining the LaTeX style file and macros used in producing the Scheme report, on which this document is based. We acknowledge the influence of the Java™ Language Specification [5].

This document revises Version 1.7 of the Report, released in September 2008. It documents the language corresponding to Version 2.0 of the implementation. Version 1.7 of the report was co-authored by Nathaniel Nystrom. The design of structs in X10 was led by Olivier Tardieu and Nathaniel Nystrom.

Earlier implementations benefited from significant contributions by Raj Barik, Philippe Charles, David Cunningham, Christopher Donawa, Robert Fuhrer, Christian Grothoff, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Vivek Sarkar, Olivier Tardieu, Pradeep Varma, Krishna Nandivada Venkata, and Christoph von Praun. Tong Wen has written many application programs in X10. Guojing Cong has helped in the development of many applications. The implementation of generics in X10 was influenced by the implementation of PolyJ [2] by Andrew Myers and Michael Clarkson.

Contents

1 Introduction

Background

Larger computational problems require more powerful 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 message-passing format is integrated into the host language by means of an application-programming interface (API), the programmer must explicitly represent and manage the interaction between multiple processes and choreograph their data exchange; large data-structures (such as distributed arrays, graphs, hash-tables) that are conceptually unitary must be thought of as fragmented across different nodes; all processors must generally execute the same code (in an SPMD fashion) etc.

One response to this problem has been the advent of the *partitioned global address space* (PGAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [3, 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 PGAS 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 Grande forum, [7, 1]) on how to adapt such languages to the context of high-performance numerical computing. Thus X10 provides support for user-defined *struct types* (such as `Int`, `Float`, `Complex` etc), supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [4]) and supports IEEE-standard floating point arithmetic. Some capabilities for supporting operator overloading are also provided.

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the PGAS model with *asynchrony* (yielding the *APGAS* programming model). 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. 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 place casts which verify the access is local at run time. Linking with native code is supported.

X10 v2.0 builds on v1.7 to support the following features: *structs* (i.e., “headerless”, inlinable objects), type rules for preventing escape of `this` from a constructor, the introduction of a global object model, permitting user-specified (immutable) fields to be replicated with the object reference. `value` classes are no longer supported; their functionality is accomplished by using structs or global fields and methods.

Several representative idioms for concurrency and communication 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.

2 Overview of X10

X10 is a statically typed object-oriented language, extending a sequential core language with *places*, *activities*, *clocks*, (distributed, multi-dimensional) *arrays* and *struct* types. All these changes are motivated by the desire to use the new language for high-end, high-performance, high-productivity computing.

2.1 Object-oriented features

The sequential core of X10 is a *container-based* object-oriented language similar to Java and C++, and more recent language such as Scala. Programmers write X10 code by defining containers for data and behavior called *interfaces* (§7), *classes* (§8) and *structs* (§9). X10 provides inheritance and subtyping in fairly traditional ways.

Example 2.1.1 `Normed` describes entities with a `norm()` method. `Normed` is intended to be used for entities with a position in some coordinate system, and `norm()` gives the distance between the entity and the origin. A `Slider` is an object which can be moved around on a line; a `PlanePoint` is a fixed position in a plane. Both `Slider`s and `PlanePoint`s have a sensible `norm()` method, and implement `Normed`.

```
interface Normed {
  def norm():Double;
}
class Slider implements Normed {
  var x : Double = 0;
  public def norm() = Math.abs(x);
  public def move(dx:Double) { x += dx; }
```

```
}
struct PlanePoint implements Normed {
  val x : Double, y:Double;
  public def this(x:Double, y:Double) {
    this.x = x; this.y = y;
  }
  public def norm() = Math.sqrt(x*x+y*y);
}
```

□

Interfaces An X10 interface specifies a collection of abstract methods; `Normed` specifies just `norm()`. Classes and structs can be specified to *implement* interfaces, as `Slider` and `PlanePoint` implement `Normed` , and, when they do so, must provide all the methods that the interface demands.

Interfaces are purely abstract. Every value of type `Normed` must be an instance of some class like `Slider` or some struct like `PlanePoint` which implements `Normed`; no value can be `Normed` and nothing else.

Classes and Structs There are two kinds of concrete containers: *classes* (§8.3) and *structs* (§9). Concrete containers hold data in *fields*, and give concrete implementations of methods, as `Slider` and `PlainPoint` above.

Classes are organized in a single-inheritance tree: a class may have only a single parent class, though it may implement many interfaces and have many subclasses. Classes may have mutable fields, as `Slider` does.

In contrast, structs are headerless values, lacking the internal organs which give objects their intricate behavior. This makes them less powerful than objects (*e.g.*, structs cannot inherit methods, though objects can), but also cheaper (*e.g.*, they can be inlined, and they require less space than objects). Structs are immutable, though their fields may be immutably set to objects which are themselves mutable. They behave like objects in all ways consistent with these limitations; *e.g.*, while they cannot *inherit* methods, they can have them – as `PlanePoint` does.

X10 has no primitive classes per se. However, the standard library `x10.lang` supplies structs `Boolean`, `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Complex` and `String`. The user may defined additional arithmetic structs using the facilities of the language.

Functions. X10 provides functions (§10) to allow code to be used as values. Functions are first-class data: they can be stored in lists, passed between activities, and so on. `square`, below, is a function which squares an `Int`. `of4` takes an `Int`-to-`Int` function and applies it to the number 4. So, `fourSquared` computes `of4(square)`, which is `square(4)`, which is 16, in a fairly complicated way.

```
val square = (i:Int) => i*i;
val of4 = (f: (Int)=>Int) => f(4);
val fourSquared = of4(square);
```

They are used extensively in X10 programs. For example, the normal way to construct a `Rail[Int]` – that is, a fixed-length array of numbers, like an `int[]` in Java – is to pass two arguments to a factory method: the first argument being the length of the rail, and the second being a function which computes the initial value of the $\mathtt{i}^{th}$ element. The following code constructs a rail initialized to the squares of 0,1,...,9: `r(0) == 0`, `r(5)==25`, etc.

```
val r : Rail[Int] = Rail.make[Int](10, square);
```

Constrained Types X10 containers may declare *properties*, which are fields bound immutably at the creation of the container. The static analysis system understands properties, and can work with them logically.

For example, an implementation of matrices `Mat` might have the numbers of rows and columns as properties. A little bit of care in definitions allows the definition of a + operation that works on matrices of the same shape, and * that works on matrices with appropriately matching shapes The following code typechecks, but an attempt to compute `axb1 + bxc` or `bxc * axb1` would result in a compile-time type error:

```
static def example(a:Int, b:Int, c:Int) {
  val axb1 : Mat(a,b) = makeMat(a,b);
  val axb2 : Mat(a,b) = makeMat(a,b);
  val bxc  : Mat(b,c) = makeMat(b,c);
  val axc  : Mat(a,c) = (axb1 +axb2) * bxc;
}
```

The “little bit of care” shows off many of the features of constrained types. The `(rows:Int, cols:Int)` in the class definition declares two properties, `rows` and `cols`.[1]

A constrained type looks like `Mat{self.rows==r && self.cols==c}`: a type name, followed by a Boolean expression in braces. The special variable `self` refers to the matrix whose number of rows and columns is being checked. The `type` declaration on the second line makes `Mat(2,3)` be a synonym for `Mat{self.rows==r && self.cols==c}`, allowing for compact types in many places.

Functions can return constrained types. The `makeMat(r,c)` method returns a `Mat(r,c)` – a matrix whose shape is given by the arguments to the method. For the sake of brevity in the example, it returns `null`; in real code, it would actually produce a matrix – which must be statically provable to have the right shape. In particular, constructors can have constrained return types to provide specific information about the constructed values.

The arguments of methods can have type constraints as well. The `operator this +` line lets `A+B` add two matrices. The type of the second argument `y` is constrained to have the same number of rows and columns as the first argument `this`. Attempts to add mismatched matrices will be flagged as type errors at compilation.

At times it is more convenient to put the constraint on the method as a whole, as seen in the `operator this *` line. Unlike for +, there is no need to constrain both dimensions; we simply need to check that the columns of the left factor match the rows of the right. This constraint is written in `{...}` after the argument list. The shape of the result is computed from the shapes of the arguments.

And that is all that is necessary for a user-defined class of matrices to have shape-checking for matrix addition and multiplication. The `example` method compiles under those definitions.

```
abstract class Mat(rows:Int, cols:Int) {
 static type Mat(r:Int, c:Int) = Mat{self.rows==r&&self.cols==c};
 static def makeMat(r:Int,c:Int) : Mat(r,c) = null;
 abstract global operator this + (y:Mat(this.rows,this.cols))
                :Mat(this.rows, this.cols);
 abstract global operator this * (y:Mat) {this.cols == y.rows}
                :Mat(this.rows, y.cols);
```

[1]The class is officially declared abstract to allow for multiple implementations, like sparse and band matrices, but in fact is abstract to avoid having to write the actual definitions of + and *.

Generic types Containers may have type parameters, permitting the definition of *generic types*. Type parameters may be instantiated by any X10 type. It is thus possible to make a list of integers `List[Int]`, a list of non-zero integers `List[Int{self != 0}]`, or a list of people `List[Person]`. In the definition of `List`, `T` is a type parameter; it can be instantiated with any type.

```
class List[T] {
    var head: T;
    var tail: List[T]!;
    def this(h: T, t: List[T]!) { head = h; tail = t; }
    def add(x: T) {
        if (this.tail == null)
            this.tail = new List(x, null);
        else
            this.tail.add(x);
    }
}
```

The constructor (`def this`) initializes the fields of the new object. The `add` method appends an element to the list. `List` is a generic type. When instances of `List` are allocated, the type parameter `T` must be bound to a concrete type. `List[Int]` is the type of lists of element type `Int`, `List[List[String]]` is the type of lists whose elements are themselves lists of string, and so on.

2.2 The sequential core of X10

The sequential aspects of X10 are mostly familiar from C and its progeny. X10 enjoys the familiar control flow constructs: `if` statements, `while` loops, `for` loops, `switch` statements, `throw` to raise exceptions and `try...catch` to handle them, and so on.

X10 has both implicit coercions and explicit conversions, and both can be defined on user-defined types. Explicit conversions are written with the `as` operation: `n as Int`. The types can be constrained: `n as Int{self != 0}` converts `n` to a non-zero integer, and throws a runtime exception if its value as an integer is zero.

2.3 Places and activities

The full power of X10 starts to emerge with concurrency. 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* (§13). A place can be thought of as 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 *values*(§8.1) through the execution of lightweight threads called *activities*(§14). Values are of three kinds. An *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, thought different aggregates may be distributed differently. Objects are garbage-collected when no longer useable; 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 it is running. It may atomically update one or more data items, but only in the current place. If it becomes necessary to read or modify an object at some other place `q`, the *place-shifting* operation `at(q)` can be used, to move part of the activity to `q`. It is easy to compute across multiple places, but the expensive operations (*e.g.*, those which require communication) are readily visible in the code.

Object and Places. Every object has a *home place*, `x.home`. `home` is a property, and the constrained type mechanisms mentioned previously can be used with it. For example, an object `x` with a mutable field `f` only allows mutations when `x.home==here`, which formalizes the concept above. If we know that `x` is located here but the typechecker doesn't, we can use a cast to tell it:

```
val xhere = x as FHolder!;
xhere.f = 12;
```

(`FHolder!` is an abbreviation for `FHolder{self.home==here}`, a `FHolder` object located here.) This will throw an exception if `x` is not located here.

Atomic blocks. X10 has a control construct `atomic S` where `S` is a statement with certain restrictions. `S` will be executed atomically, without interruption by other activities. This is a common primitive used in concurrent algorithms, though rarely provided in this degree of generality by concurrent programming languages.

More powerfully – and more expensively – X10 allows conditional atomic blocks, `when(B)S`, which are executed atomically at some point when `B` is true. Conditional atomic blocks are one of the strongest primitives used in concurrent algorithms, and one of the least-often available.

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 `P` executing statement `S`.

If the activity needs to return a value, the form `future (p) E` is often convenient. This spawns an activity at `p` evaluating `E`, and also returns a value called a *future* which is capable of accepting the value of `E` when it is ready. The caller can try to get the value from the future, which provides it immediately if it is ready and blocks to wait for it if it is not.

2.4 Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic network of heterogeneous activities in an X10 computation. X10 allows multiple barriers in a form that supports determinate, deadlock-free parallel computation, via the `Clock` type.

A single `Clock` represents a computation that occurs in phases. At any given time, an activity is *registered* with zero or more clocks. The X10 statement `next` tells all of an activity's registered clocks that the activity has finished the current phase, and causes it to wait for the next phase. Other operations allow waiting on a single clock, starting new clocks or new activities registered on an extant clock, and so on.

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.

2.5 Arrays, regions and distributions

X10 provides `DistArrays`, *distributed arrays*, which spread data across many places. An underlying `Dist` object provides the *distribution*, telling which elements of the `DistArray` go in which place. `Dist` uses subsidiary `Region` objects to abstract over the shape and even the dimensionality of arrays. Specialized X10 control statements such as `ateach` provide efficient parallel iteration over distributed arrays.

2.6 Annotations

X10 supports annotations on classes and interfaces, methods and constructors, variables, types, expressions and statements. These annotations may be processed by compiler plugins.

2.7 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.

2.8 Summary and future work

2.8.1 Design for scalability

X10 is designed for scalability, by encouraging working with local data, and limiting the ability of events at one place to delay those at another. For example, 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. Data-flow synchronization permits point-to-point coordination between reader/writer activities, obviating the need for barrier-based or lock-based synchronization in many cases.

2.8.2 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 every location contains only values whose dynamic type agrees with the location's static type. The compiler allows a choice of how to handle method calls. In strict mode, method calls are statically checked to be permitted by the static types of operands. In lax mode, dynamic checks are inserted when calls may or may not be correct, providing weaker static correctness guarantees but more programming convenience.

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 garbage collection to collect objects no longer referenced by any activity. 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 run time before it is read, and every value read from a word of memory has previously been written into that word.

Because places are reflected in the type system, static type safety also implies *place safety*. All operations that need to be performed locally are, in fact, performed locally. All data which is declared to be stored locally are, in fact, stored locally.

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. 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.

2.8.3 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.[2] 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.

[2] In this X10 is similar to more modern languages such as ZPL [4].

3 Lexical structure

In general, X10 follows Java rules [5, Chapter 3] for lexical structure.

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

Whitespace ASCII space, horizontal tab (HT), form feed (FF) and line terminators constitute white space.

Comments All text included within the ASCII characters "`/*`" and "`*/`" is considered a comment and ignored; nested comments are not allowed. All text from the ASCII characters "`//`" to the end of line is considered a comment and is ignored.

Identifiers Identifiers are defined as in Java. Identifiers consist of a single letter followed by zero or more letters or digits. Letters are defined as the characters for which the Java method `Character.isJavaIdentifierStart` returns true. Digits are defined as the ASCII characters `0` through `9`.

Keywords X10 reserves the following keywords:

```
abstract    any        as         async
at          ateach     atomic     await
break       case       catch      class
clocked     const      continue   current
def         default    do         else
extends     extern     final      finally
finish      for        foreach    future
```

`global`			
`goto`	`has`	`here`	`if`
`implements`	`import`	`in`	
`instanceof`	`interface`		
`native`	`new`	`next`	`nonblocking`
`or`	`package`	`pinned`	`private`
`protected`	`property`	`public`	`return`
`safe`	`self`	`sequential`	`shared`
`static`			
`super`	`switch`	`this`	`throw`
`throws`	`to`	`try`	`type`
`val`	`value`	`var`	`when`
`while`			

Note that the primitive types are not considered keywords. The keyword `goto` is reserved, but not used.

Literals Briefly, X10 v2.0 uses fairly standard syntax for its literals: integers, unsigned integers, floating point numbers, booleans, characters, strings, and `null`. The most exotic points are (1) unsigned numbers are marked by a `u` and cannot have a sign; (2) `true` and `false` are the literals for the booleans; and (3) floating point numbers are `Double` unless marked with an `f` for `Float`.

Less briefly, we use the following abbreviations:

$$
\begin{array}{rcl}
\delta & = & \text{one or more decimal digits} \\
\delta_8 & = & \text{one or more octal digits} \\
\delta_{16} & = & \text{one or more hexadecimal digits, using } \texttt{a-f} \text{ for 10-15} \\
\iota & = & \delta \mid \texttt{0}\delta_8 \mid \texttt{0x}\delta_{16} \mid \texttt{0X}\delta_{16} \\
\sigma & = & \text{optional + or -} \\
\beta & = & \delta \mid \delta. \mid \delta.\delta \mid .\delta \\
\xi & = & (\texttt{e} \mid \texttt{E})\sigma\delta \\
\phi & = & \beta\xi
\end{array}
$$

- `true` and `false` are the `Boolean` literals.
- `null` is a literal for the null value. It has type `Any{self==null}`.

- `Int` literals have the form $\sigma\iota$; *e.g.*, `123`, `-321` are decimal `Int`s, `0123` and `-0321` are octal `Int`s, and `0x123`, `-0X321`, `0xBED`, and `0XEBEC` are hexadecimal `Int`s.
- `Long` literals have the form $\sigma\iota$`l` or $\sigma\iota$`L`. *E.g.*, `1234567890L` and `0xBAGEL` are `Long` literals.
- `UInt` literals have the form ι`u` or ι`U`. *E.g.*, `123u`, `0123u`, and `0xBEAU` are `UInt` literals.
- `ULong` literals have the form ι`ul` or ι`lu`, or capital versions of those. For example, `123ul`, `0124567012ul`, `0xFLU`, `OXba1eful`, and `0xDecafC0ffeefUL` are `ULong` literals.
- `Float` literals have the form $\sigma\phi$`f` or $\sigma\phi$`F`. Note that the floating-point marker letter `f` is required: unmarked floating-point-looking literals are `Double`. *E.g.*, `1f`, `6.023E+32f`, `6.626068E-34F` are `Float` literals.
- `Double` literals have the form $\sigma\phi$[1], $\sigma\phi$`D`, and $\sigma\phi$`d`. *E.g.*, `0.0`, `0e100`, `229792458d`, and `314159265e-8` are `Double` literals.
- `Char` literals have one of the following forms:
  - `'`*c*`'` where *c* is any printing ASCII character other than `\` or `'`, representing the character *c* itself; *e.g.*, `'!'`;
  - `'\b'`, representing backspace;
  - `'\t'`, representing tab;
  - `'\n'`, representing newline;
  - `'\f'`, representing form feed;
  - `'\r'`, representing return;
  - `'\''`, representing single-quote;
  - `'\"'`, representing double-quote;
  - `'\\'`, representing backslash;
  - `'\`*dd*`'`, where *dd* is one or more octal digits, representing the one-byte character numbered *dd*; it is an error if *dd*>255.

[1]Except that literals like `1` which match both ι and ϕ are counted as integers, not `Double`; `Double`s require a decimal point, an exponent, or the `d` marker.

- `String` literals consist of a double-quote `"`, followed by zero or more of the contents of a `Char` literal, followed by another double quote. *E.g.*, `"hi!"`, `""`.
- There are no literals of type `Byte` or `UByte`.

Separators X10 has the following separators and delimiters:

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

Operators X10 has the following operators:

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

4 Types

X10 is a *strongly typed* object-oriented language: every variable and expression has a type that is known at compile-time. Types limit the values that variables can hold and specify the places at which these values can lie.

X10 supports three kinds of runtime entities, *objects*, *structs*, and *functions*. Objects are instances of *classes* (§8.3). They may contain mutable fields and stay resident in the place in which they were created. Objects are said to be *boxed* in that variables of a class type are implemented through a single memory location that contains a reference to the memory containing the declared state of the object (and other meta-information such as the list of methods of the object). Thus objects are represented through an extra level of indirection. A consequence of this flexibility is that every class type contains the value `null` corresponding to the invalid reference. `null` is often useful as a default value. Further, two objects may be compared for identity (==) in constant time by simply containing references to the memory used to represent the objects.

Structs are instances of *struct types* (§9). They are a restricted variant of classes, lacking meta-information; this makes them less flexible, but in many cases more efficient. When it is semantically meaningful, converting a class into a struct or vice-versa is quite easy. Structs are immutable and may be freely copied from place to place. Further, they may be allocated inline, using only as much memory as necessary to hold and align the fields of the struct.

Functions, called closures or lambda-expressions in other languages, are instances of *function types*— §10). Functions can refer to variables from the surrounding environment; *e.g.*, `(x:Int)=>x*y` is a unary integer function which multiplies its argument by the variable `y` from the surrounding block. Functions may be freely copied from place to place and may be repeatedly applied to a set of arguments.

These runtime entities are classified by *types*. Types are used in variable declarations, explicit coercions and conversions, object creation, array creation, class literals, static state and method accessors, and `instanceof` expressions.

The basic relationship between values and types is *instantiation*. For example, `1` is an instance of type of integers, `Int`. It is also an instance of type of all entities `Any`, and of type of nonzero integers `Int{self != 0}`, and many others.

The basic relationship between types is *subtyping*: `T <: U` holds if every instance of `T` is also an instance of `U`. Two important kinds of subtyping are *subclassing* and *strengthening*. Subclassing is a familiar notion from object-oriented programming. In a class hierarchy with classes `Animal` and `Cat` arranged in the usual way, every `Cat` is an `Animal`, so `Cat <: Animal` by subclassing. Strengthening is an equally familiar notion from logic. The instances of `Int{self != 0}` are all elements of `Int{true}` as well, because `self != 0` logically implies `true`; so `Int{self != 0} <: Int{true} == Int` by strengthening. X10 uses both notions of subtyping. See §4.8 for the full definition of subtyping in X10.

The Grammar of Types

Types are described by the following grammar:

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \mathit{FunctionType} \\
 & | & \mathit{ConstrainedType} \\
\mathit{FunctionType} & ::= & \mathit{TypeParameters}^{?}\ \texttt{(}\ \mathit{Formals}^{?}\ \texttt{)}\ \mathit{Constraint}^{?}\ \mathit{Throws}^{?}\ \texttt{=>}\ \mathit{TypeOrVoid} \\
\mathit{TypeParameters} & ::= & \texttt{[}\ \mathit{TypeParameter}\ (\ \texttt{,}\ \mathit{TypeParameter}\ )^{*}\ \texttt{]} \\
\mathit{TypeParameter} & ::= & \mathit{Identifier} \\
\mathit{Throws} & ::= & \texttt{throws}\ \mathit{TypeName}\ (\ \texttt{,}\ \mathit{TypeName}\ )^{*} \\
\mathit{ConstrainedType} & ::= & \mathit{Annotation}^{*}\ \mathit{BaseType}\ \mathit{Constraint}^{?}\ \mathit{PlaceConstraint}^{?} \\
\mathit{BaseType} & ::= & \mathit{ClassBaseType} \\
 & | & \mathit{InterfaceBaseType} \\
 & | & \mathit{PathType} \\
 & | & \texttt{(}\ \mathit{Type}\ \texttt{)} \\
\mathit{TypeOrVoid} & ::= & \mathit{Type} \\
 & | & \texttt{Void} \\
\mathit{ClassType} & ::= & \mathit{Annotation}^{*}\ \mathit{ClassBaseType}\ \mathit{Constraint}^{?}\ \mathit{PlaceConstraint}^{?} \\
\mathit{InterfaceType} & ::= & \mathit{Annotation}^{*}\ \mathit{InterfaceBaseType}\ \mathit{Constraint}^{?}\ \mathit{PlaceConstraint}^{?} \\
\mathit{PathType} & ::= & \mathit{Expression}\ \texttt{.}\ \mathit{Identifier} \\
\mathit{Annotation} & ::= & \texttt{@}\ \mathit{InterfaceBaseType}\ \mathit{Constraint}^{?} \\
\mathit{ClassOrInterfaceType} & ::= & \mathit{ClassType} \\
 & | & \mathit{InterfaceType} \\
\mathit{ClassBaseType} & ::= & \mathit{TypeName} \\
\mathit{InterfaceBaseType} & ::= & \mathit{TypeName}
\end{array}
$$

4.1 Classes and interfaces

4.1.1 Class types

A *class declaration* (§8) introduces a *class type* containing all instances of the class. The `Position` class below could describe the position of a slider control, for example.

```
class Position {
  private var x : Int = 0;
  public def move(dx:Int) { x += dx; }
```

```
  public def pos() : Int = x;
}
```

Class instances, also called objects, are created via constructor calls. Class instances have fields and methods, type members, and value properties bound at construction time. In addition, classes have static members: constant fields, type definitions, and member classes and member interfaces.

A class with type parameters is *generic*. A class type is instantiatable only if all of its parameters are instantiated on concrete types. The `Cell[T]` class provides a container capable of holding a value of type `T`, or being empty.

```
class Cell[T] {
  var empty : Boolean = true;
  var contents : T;
  public def putIn(t:T) {
    contents = t; empty = false;
  }
  public def emptyOut() { empty = true; }
  public def isEmpty() = empty;
  public def getOut():T throws Exception {
     if (empty) throw new Exception("Empty!");
     return contents ;
  }
}
```

X10 does not permit mutable static state. A fundamental principle of the X10 model of computation is that all mutable state be local to some place (§13), and, as static variables are globally available, they cannot be mutable. When mutable global state is necessary, programmers should use singleton classes, putting the state in an object and using place-shifting commands (§14.3) and atomicity (§14.11) as necessary to mutate it safely.

Classes are structured in a single-inheritance hierarchy. All classes extend the class `x10.lang.Object`, directly or indirectly. Each class other than `Object` extends a single parent class. `Object` provides no behavior of its own, beyond that required by `Any`.

Variables of class type may contain the value `null`.

4.1.2 Interface types

An *interface declaration* (§7) defines an *interface type*, specifying a set of methods, type members, and properties which must be provided by any class declared to implement the interface.

Interfaces can also have static members: constant fields, type definitions, and member classes and interfaces. However, interfaces cannot specify that implementing classes must provide static members.

An interface may extend multiple interfaces.

```
interface Named {
  def name():String;
}
interface Mobile {
  def move(howFar:Int):Void;
}
interface Person extends Named, Mobile {}
interface NamedPoint extends Named, Mobile{}
```

Classes may be declared to implement multiple interfaces. Semantically, the interface type is the set of all objects that are instances of classes that implement the interface. A class implements an interface if it is declared to and if it implements all the methods and properties defined in the interface. For example, `KimThePoint` implements `Person`, and hence `Named` and `Mobile`. It would be a static error if `KimThePoint` had no `name` method.

```
class KimThePoint implements Person {
   var pos : Int = 0;
   public def name() = "Kim (" + pos + ")";
   public def move(dPos:Int) { pos += dPos; }
}
```

4.1.3 Properties

Classes, interfaces, and structs may have *properties*, public `val` instance fields bound on object creation. For example, the following code declares a class named `Coords` with properties `x` and `y` and a `move` method. The properties are bound using the `property` statement in the constructor.

```
class Coords(x: Int, y: Int) {
  def this(x: Int, y: Int) : Coords{self.x==x, self.y==y}
    = { property(x, y); }
  def move(dx: Int, dy: Int) = new Coords(x+dx, y+dy);
}
```

Properties, unlike other public `val` fields, can be used at compile time in *constraints*. This allows us to specify subtypes based on properties, by appending a boolean expression to the type. For example, the type `Coords{x==0}` is the set of all points whose `x` property is `0`. Details of this substantial topic are found in §4.5.

4.2 Type parameters and Generic Types

A class, interface, method, closure, or type definition may have type parameters. Type parameters can be used as types, and will be bound to types on instantiation. For example, a generic stack class may be defined as `Stack[T]{...}`. Stacks can hold values of any type; *e.g.*, `Stack[Int]` is a stack of integers, and `Stack[Point{self!=null}]`is a stack of non-null `Points`. Generics *must* be instantiated when they are used: `Stack`, by itself, is not a valid type. Type parameters may be constrained by a guard on the declaration (§4.4, §8.4.1,§10.2).

A *generic type* is a class, struct, interface, or type declared with one or more type parameters. When instantiated with concrete (*viz.*, non-generic) types for its parameters, a generic type becomes a concrete type and can be used like any other type. For example, `Stack` is a generic type, `Stack[Int]` is a concrete type, and can be used as one: `var stack : Stack[Int];`

A `Cell[T]` is a generic object, capable of holding a value of type `T`. For example, a `Cell[Int]` can hold an `Int`, and a `Cell[Cell[Int]{self!=0}]` can hold a `Cell[Int]` which in turn can only hold non-zero numbers. `Cells` are actually useful in situations where values must be bound immutably for one reason, but need to be mutable.

```
class Cell[T] {
    var x: T;
    def this(x: T) { this.x = x; }
    def get(): T = x;
    def set(x: T) = { this.x = x; }
```

```
}
```

Cell[Int] is the type of Int-holding cells. The get method on a Cell[Int] returns an Int; the set method takes an Int as argument. Note that Cell alone is not a legal type because the parameter is not bound.

4.2.1 Variance of Type Parameters

Consider classes Person :> Child. Every child is a person, but there are people who are not children. What is the relationship between Cell[Person] and Cell[Child]?

Why Variance Is Necessary

In this case, Cell[Person] and Cell[Child] should be unrelated. If we had Cell[Person] :> Cell[Child], the following code would let us assign a old (a Person but not a Child) to a variable young of type Child, thereby breaking the type system:

```
// INCORRECTLY assuming Cell[Person] :> Cell[Child]
val cc : Cell[Child] = new Cell[Child]();
val cp : Cell[Person] = cc; // legal upcast
cp.set(old);       // legal since old : Person
val young : Child = cc.get();
```

Similarly, if Cell[Person] <: Cell[Child]:

```
// INCORRECTLY assuming Cell[Person] <: Cell[Child]
val cp : Cell[Person] = new Cell[Person];
val cc : Cell[Child] = cp; // legal upcast
val cp.set(old);
val young : Child = cc.get();
```

So, there cannot be a subtyping relationship in either direction between the two. And indeed, neither of these programs passes the X10 typechecker.

Legitimate Variance

The Cell[Person]-vs-Cell[Child] problems occur because it is possible to both store and retrieve values from the same object. However, entities with only one of the two capabilities *can* sensibly have some subtyping relations. Furthermore, both sorts of entity are useful. An entity which can store values but not retrieve them can nonetheless summarize them. An object which can retrieve values but not store values can be constructed with an initial value, providing a read-only cell.

So, X10 provides *variance* to support these options. Type parameters may be defined in one of three forms.

1. *invariant*: Given a definition `class C[T]{...}`, `C[Person]` and `C[Child]` are unrelated classes; neither is a subclass of the other.

2. *covariant*: Given a definition `class C[+T]{...}` (the + indicates covariance), `C[Person] :> C[Child]`. This is appropriate when `C` allows retrieving values but not setting them.

3. *contravariant*: Given a definition `class C[-T]{...}` (the - indicates contravariance), `C[Person] <: C[Child]`. This is appropriate when `C` allows storing values but not retrieving them.

The `T` parameter of `Cell` above is invariant.

A typical example of covariance is `Get`. As the `example()` method shows, a `Get[T]` must be constructed with its value, and will return that value whenever desired.

```
class Get[+T] {
  var x: T;
  def this(x: T) { this.x = x; }
  def get(): T = x;
  static def example() {
     val g : Get[Int]! = new Get[Int](31);
     val n : Int = g.get();
     x10.io.Console.OUT.print("It’s " + n);
     x10.io.Console.OUT.print("It’s still " + g.get());
  }
}
```

A typical example of contravariance is `Set`. As the `example()` method shows, a variety of objects[1] can be put into a `Set[Object]`. While the object itself cannot be retrieved, some summary information about it – in this case, its `typeName` – can be.

```
class Set[-T] {
  var x: T;
  def this(x: T) { this.x = x; }
  def set(x: T) = { this.x = x; }
  def summary(): String = this.x.typeName();
  static def example() {
    val s : Set[Object]! = new Set[Object](new Throwable());
    s.summary(); // == "x10.lang.Throwable"
    s.set("A String");
    s.summary(); // == "x10.lang.String";
  }
}
```

Given types `S` and `T`:

- If the parameter of `Get` is covariant, then `Get[S]` is a subtype of `Get[T]` if `S` is a *subtype* of `T`.
- If the parameter of `Set` is contravariant, then `Set[S]` is a subtype of `Set[T]` if `S` is a *supertype* of `T`.
- If the parameter of `Cell` is invariant, then `Cell[S]` is a subtype of `Cell[T]` if `S` is a *equal* to `T`.

In order to make types marked as covariant and contravariant semantically sound, X10 performs extra checks. A covariant type parameter is permitted to appear only in covariant type positions, and a contravariant type parameter in contravariant positions.

- The return type of a method is a covariant position.
- The argument types of a method are contravariant positions.

[1]Objects but no structs. If we had wanted structs too, we could have used a `Cell[Any]`.

- Whether a type argument position of a generic class, interface or struct type `C` is covariant or contravariant is determined by the `+` or `-` annotation at that position in the declaration of `C`.

There are similar restrictions on use of covariant and contravariant values.

4.3 Function Types

For every sequence of types `T1,..., Tn,T`, and `n` distinct variables `x1,...,xn` and constraint `c`, the expression `(x1:T1,...,xn:Tn){c}=>T` is a *function type*. It stands for the set of all functions `f` which can be applied to a list of values `(v1,...,vn)` provided that the constraint `c[v1,...,vn,p/x1,...,xn]` is true, and which returns a value of type `T[v1,...vn/x1,...,xn]`. When `c` is true, the clause `{c}` can be omitted. When `x1,...,xn` do not occur in `c` or `T`, they can be omitted. Thus the type `(T1,...,Tn)=>T` is actually shorthand for `(x1:T1,...,xn:Tn){true}=>T`, for some variables `x1,...,xn`.

4.4 Type definitions

With value arguments, type arguments, and constraints, the syntax for X10 types can often be verbose. For example, a non-null list of non-null strings is `List[String{self!=null}]{self!=null}`. X10 provides *type definitions* to allow users to give short names to long types, and to commonly-used combinations of types. We could name that type:

```
type LnSn = List[String{self!=null}]{self!=null};
```

Or, we could abstract it somewhat, defining a type constructor `Nonnull[T]` for the type of `T`'s which are not null:

```
type Nonnull[T] = T{self!=null};
type LnSn = Nonnull[List[Nonnull[String]]];
var example : LnSn;
```

Type definitions can also refer to values, in particular, inside of constraints. The type of `n`-element `Rails[Int]s` is `Rail[Int]{self.length == n}` but it is often convenient to give a shorter name:

```
type Rail(n:Int) = Rail[Int]{self.length == n};
var example : Rail(78);
```

Type definitions, like many other X10 abstractions, can have constraints on their use.

Type definitions have the following syntax:

TypeDefinition ::= `type` *Identifier* (`[` *TypeParameters* `]`)$^?$
(`(` *Formals* `)`)$^?$ *Constraint*$^?$ = *Type*

A type definition can be thought of as a type-valued function, mapping type parameters and value parameters to a concrete type. The following examples are legal type definitions, given `import x10.util.*`:

```
type StringSet = Set[String];
type MapToList[K,V] = Map[K,List[V]];
type Int(x: Int) = Int{self==x};
type Dist(r: Int) = Dist{self.rank==r};
type Dist(r: Region) = Dist{self.region==r};
type Redund(n:Int, r:Region){r.rank==n} = Dist{rank==n && region==r};
```

As the two definitions of `Dist` demonstrate, type definitions may be overloaded: two type definitions with different numbers of type parameters or with different types of value parameters, according to the method overloading rules (§8.4.3), define distinct type constructors.

Type definitions may appear as (static) class or interface member or in a block statement.

Type definitions are applicative, not generative; that is, they define aliases for types but do not introduce new types. Thus, the following code is legal:

```
type A = Int;
type B = String;
type C = String;
a: A = 3;
b: B = new C("Hi");
c: C = b + ", Mom!";
```

If a type definition has no type parameters and no value parameters and is an alias for a class type, a `new` expression may be used to create an instance of the class using the type definition's name. Given the following type definition:

```
type A = C[T₁, ..., Tₖ]{c};
```

where `C[T`$_1$`, ..., T`$_k$`]` is a class type, a constructor of `C` may be invoked with `new A(e`$_1$`, ..., e`$_n$`)`, if the invocation `new C[T`$_1$`, ..., T`$_k$`](e`$_1$`, ..., e`$_n$`)` is legal and if the constructor return type is a subtype of `A`.

The collection of type definitions in `x10.lang.\_` is automatically imported in every compilation unit.

4.5 Constrained types

Basic types, like `Int` and `List[String]`, provide useful descriptions of data. Indeed, most typed programming languages get by with no more specific descriptions.

However, there are a lot of things that one frequently wants to say about data. One might want to know that a `String` variable is not `null`, or that a matrix is square, or that one matrix has the same number of columns that another has rows (so they can be multiplied). In the multicore setting, one might wish to know that two values are located at the same processor, or that one is located at the same place as the current computation.

In most languages, there is simply no way to say these things statically. Programmers must made do with comments, `assert` statements, and dynamic tests. X10 can do better, with *constraints* on types (and methods and other things).

A constraint is a boolean expression `e` attached to a basic type `T`, written `T{e}`. (Only a limited selection of boolean expressions is available.) The values of type `T{e}` are the values of `T` for which `e` is true. For example:

- `String{self != null}` is the type of non-null strings. `self` is a special variable available only in constraints; it refers to the datum being constrained.
- If `Matrix` has properties `rows` and `cols`, `Matrix{rows == cols}` is the type of square matrices.
- One way to say that `a` has the same number of columns that `b` has rows (so that `a*b` is a valid matrix product), one could say:

```
val a : Matrix = someMatrix() ;
var b : Matrix{b.rows == a.cols} ;
```

- Every object has a `home` property telling where it is located. One way to say that objects `c` and `d` are located at the same place is:

```
val a : Object = someObject();
var b : Object{a.home == b.home};
```

- As explained in §13, certain operations can only be performed at an object's home, so having this expressible as a type is crucial. One way to say that `e` is located `here`, *viz.*, the same place as the current computation, is:

```
val e : Object{self.home == here} = someObject();
```

When constraining a value of type `T`, `self` refers to the object of type `T` which is being constrained. For example, `Int{self == 4}` is the type of `Int`s which are equal to 4 – the best possible description of `4`, and a very difficult type to express without using `self`.

`T{e}` is a *dependent type*, that is, a type dependent on values. The type `T` is called the *base type* and `e` is called the *constraint*. If the constraint is omitted, it is `true`—that is, the base type is unconstrained.

Constraints may refer to values in the local environment:

```
val n = 1;
var p : Point{rank == n};
```

Indeed, there is technically no need for a constraint to refer to the properties of its type; it can refer entirely to the environment, thus:

```
val m = 1;
val n = 2;
var p : Point{m != n};
```

Constraints on properties induce a natural subtyping relationship: `C{c}` is a subtype of `D{d}` if `C` is a subclass of `D` and `c` entails `d`. For example:

- `Int{self == 3} <: Int{self != 14}`. The only value of `Int{self ==3}` is 3. All integers but 14 are members of `Int{self != 14}`, and in particular 3 is.

- Suppose we have classes `Child <: Person`, and `Person` has a long `ssn` property. If `rhys : Child{ssn == 123456789}`, then `rhys` is also a `Person` and still has `ssn==123456789`, so `rhys : Person{ssn==123456789}` as well. So, `Child{ssn == 123456789} <: Person{ssn == 123456789}`.
- Furthermore, since `123456789 != 555555555`, `rhys : Person{ssn != 555555555}`. So, `Child{ssn == 123456789} <: Person{ssn != 555555555}`.
- `T{e} <: T` for any type `T`. That is, if you have a value `v` of some base type `T` which satisfied `e`, then `v` is of that base type `T` (with the constraint ignored).
- If `A <: B`, then `A{c} <: B{c}` for every constraint `{c}` for which `A{c}` and `B{c}` are defined. That is, if every `A` is also a `B`, and `a : A{c}`, then `a` is an `A` and `c` is true of it. So `a` is also a `B` (and `c` is still true of it), so `a : B{c}`.

4.5.1 Constraint Expressions

Only a few kinds of expressions can appear in constraints. For fundamental reasons of mathematical logic, the more kinds of expressions that can appear in constraints, the harder it is to compute the essential properties of constrained type – in particular, the harder it is to compute `A{c} <: B{d}`. It doesn't take much to make this basic fact undecidable. In order to make sure that it stays decidable, X10 places quite stringent restrictions on constraints.

Only the following forms of expression are allowed in constraints.

Value expressions in constraints may be:

1. Literal constants, like `3` and `true`;
2. Accessible and immutable variables and parameters;
3. Accessible and immutable fields of the containing object;
4. Properties of the type being constrained;
5. Property methods;
6. `this`, if the constraint is in a place where `this` is defined.

7. `here`
8. `self`

Boolean expressions in constraints may be any of the following, where all value expressions are of the forms which may appear in constraints:

1. Equalities `e == f`,
2. Inequalities of the form `e != f`.[2]
3. Conjunctions of Boolean expressions
4. Subtyping and supertyping expressions: `T <: U` and `T :> U`.
5. Type equalities and inequalities: `T == U` and `T != U`.
6. Boolean value expressions.

All variables appearing in a constraint expression must be visible wherever that expression can used. *E.g.*, properties and public fields of an object are always permitted, but private fields of an object can only constrain private members. (Consider a class `PriVio` with a private field `p` and a public method `m(x: Int{self != p})`, and a call `ob.m(10)` made outside of the class. Since `p` is only visible inside the class, there is no way to tell if `10` is of type `Int{self != p}` at the call site.)

The static constraint checker approximates computational reality in some cases. For example, it assumes that built-in types are infinite. This is a good approximation for `Int`. It is a poor approximation for `Boolean`, as the checker believes that `a != b && a != c && b != c` is satisfiable over `Boolean`, which it is not. However, the checker is always correct when computing the truth or falsehood of a constraint.

Acyclicity restriction

To ensure that type-checking is decidable, we require that property graphs be acyclic. The property graph, at an instant in an X10 execution, is the graph whose

[2] Currently inequalities of the form `e < f` are not supported.

nodes are all objects in existence at that instance, with an edge from x to y if x is an object with a property whose value is y.

Currently this restriction is not checked by the compiler. Future versions of the compiler will check this restriction by introducing rules on escaping of `this` (§8.8.5) before the invocation of `property` calls.

4.5.2 Place constraints

An X10 computation spans multiple places (§13). Much data can only be accessed from the proper place, and often it is preferable to determine this statically. So, X10 has special syntax for working with places. `T!` is a value of type `T` located at the right place for the current computation, and `T!p` is one located at place `p`.

PlaceConstraint	::=	`!` *Place*$^?$
Place	::=	*Expression*

More specifically, All X10 classes extend the class `x10.lang.Object`, which defines a property `home` of type `Place`. `T!p`, when `T` is a class, is `T{self.home==p}`. If `p` is omitted, it defaults to `here`. `T!` is far and away the most common usage of `!`.

Structs don't have `home`; they are available everywhere. For structs, `T!` and `T!p` are synonyms for `T`. Since `T` is available everywhere, it is available `here` and at `p`.

`!` may be combined with other constraints. `T{c}!` is the type of values of `T!` which satisfy `c`; it is `T{c && self.home==here}` for an object type and `T{c}` for a struct type. `T{c}!p` is the type of values of `T!p` which satisfy `c`; it is `T{c && self.home==p}` for an object type and `T{c}` for a struct type.

4.5.3 Variables in Constraints

X10 permits a `val` variable to appear in constraints on its own type as it is being declared. For example, `val nz: Int{nz != 0} = 1;` declares a non-zero variable `nz`.

4.5.4 Operations on Constrained Types

An instance `o` of `C` is said to be of type `C{c}` (or: *belong to* `C{c}`) if the constraint `c` evaluates to `true` in the current lexical environment, augmented with the binding `self` ↦ `o`.

The `instanceof` operation lets programs test type membership. `e instanceof C{c}` returns true if `e` belongs to `C{c}`, and false otherwise. `1 instanceof Int{self != 2}` returns true, and `x instanceof Int{self == 1}` returns false if `x==2`. However, it is a static error if `e` cannot possibly be an instance of `C{c}`; the compiler will reject `1 instanceof Int{self == 2}` because 1 can never satisfy `Int{self == 2}`. Similarly, `1 instanceof String` is a static error, rather than an expression always returning false.

Limitation *X10 does not currently handle* `instanceof` *of generics in the way you might expect. For example,* `elf != 0r instanceof ValRail[Ints]` *does not test that every element of* `r` *is non-zero.*

The `as` operation attempts to convert a value to a given constrained type, as described in §11.22. As with `instanceof`, it may succeed, fail with a dynamic error, or, in the case where the cast is impossible, fail to compile.

4.5.5 Example of Constraints

Example 4.5.1 Constraints can be used to express simple relationships between objects, enforcing some class invariants statically. For example, in geometry, a line is determined by two *distinct* points; a `Line` class can specify the distinctness in a type constraint:[3]

```
class Position(x: Int, y: Int) {
   def this(x:Int,y:Int){property(x,y);}
   }
class Line(start: Position,
           end: Position{self != start}) {}
```

Extending this concept, a `Triangle` can be defined as a figure with three line segments which match up end-to-end. Note that the degenerate case in which two or three of the triangle's vertices coincide is excluded by the constraint on `Line`. However, not all degenerate cases can be excluded by the type system; in particular, it is impossible to check that the three vertices are not collinear.

```
class Triangle
 (a: Line,
```

[3]We call them `Position` to avoid confusion with the built-in class `Point`

```
b: Line{a.end == b.start},
c: Line{b.end == c.start && c.end == a.start})  {
 def this(a:Line,
          b: Line{a.end == b.start},
          c: Line{b.end == c.start && c.end == a.start})
 {property(a,b,c);}
}
```

□

4.6 Function types

X10 functions, like mathematical functions, take some arguments and produce a result. X10 functions, like other X10 code, can change mutable state and throw exceptions. Closures (§10) and method selectors (§10.3) are of function type. Typical functions are the reciprocal function:

```
val recip = (x : Double) => 1/x;
```

and a function which increments element `i` of a rail `r`, or throws an exception if there is no such element, where, for the sake of example, we constrain the type of `i`:

```
val inc = (r:Rail[Int]!, i: Int{i != r.length}) => {
  if (i < 0 || i >= r.length) throw new DoomExn();
  r(i)++;
};
```

So, in general, a function type needs to list the types T_i of all the formal parameters, and their distinct names x_i in case other types refer to them; a constraint `c` on the function as a whole; a return type `T`; and the exceptions EX_j that the function might throw when applied:

Limitation *The* `throws` *clause is not currently implemented. Also, some method modifiers (*`safe`*,* `atomic`*, etc.) will apply to function types as well.*

$$(x_1\texttt{: } T_1\texttt{, } \ldots\texttt{, } x_n\texttt{: } T_n)\texttt{\{c\} => T}$$
$$\texttt{throws } EX_1\texttt{, } \ldots\texttt{, } EX_k$$

The names $\mathtt{x}_i$ of the formal parameters are not relevant. Types which differ only in the names of formals (following the usual rules for renaming of variables, as in α-renaming in the λ calculus) are considered equal. *E.g.*, `(a:Int, b:Rail[String]{b.length==a}) => Boolean` and `(b:Int, a:Rail[String]{a.length==b}) => Boolean` are equivalent types.

The formal parameter names are in scope from the point of definition to the end of the function type—they may be used in the types of other formal parameters and in the return type. Value parameters names may be omitted if they are not used; the type of the reciprocal function can be written as `(Double)=>Double`.

FunctionType	::=	*TypeParameters*? `(` *Formals*? `)` *Constraint*? => *Type Throws*?
TypeParameters	::=	`[` *TypeParameter* (`,` *TypeParameter*)* `]`
TypeParameter	::=	*Identifier*
Formals	::=	*Formal* (`,` *Formal*)*

A function type is covariant in its result type and contravariant in each of its argument types. That is, let `S1,...,Sn,S,T1,...Tn,T` be any types satisfying `Si <: Ti` and `S <: T`. Then `(x1:T1,...,xn:Tn){c}=>S` is a subtype of `(x1:S1,...,xn:Sn){c}=>T`.

A class or struct definition may use a function type `F = (x1:T1,...,xn:Tn){c}=>T` in its implements clause; this is equivalent to implementing an interface requiring the single method `def apply(x1:T1,...,xn:Tn){c}:T`. Similarly, an interface definition may specify a function type `F` in its `extends` clause. Values of a class or struct implementing `F` can be used as functions of type `F` in all ways. In particular, applying one to suitable arguments calls the `apply` method.

A function type `F` is not a class type in that it does not extend any type or implement any interfaces, or support equality tests. `F` may be implemented, but not extended, by a class or function type. Nor is it a struct type, for it has no predefined notion of equality.

4.7 Annotated types

Any X10 type may be annotated with zero or more user-defined *type annotations* (§17).

Annotations are defined as (constrained) interface types and are processed by compiler plugins, which may interpret the annotation symbolically.

A type `T` is annotated by interface types $\mathtt{A}_1, \ldots, \mathtt{A}_n$ using the syntax $\mathtt{@A}_1\ \ldots\ \mathtt{@A}_n$ `T`.

4.8 Subtyping and type equivalence

Intuitively, type $\mathtt{T}_1$ is a subtype of type $\mathtt{T}_2$, written $\mathtt{T}_1$ `<:` $\mathtt{T}_2$, if every instance of $\mathtt{T}_1$ is also an instance of $\mathtt{T}_2$. For example, `Child` is a subtype of `Person` (assuming a suitably defined class hierarchy): every child is a person. Similarly, `Int{self != 0}` is a subtype of `Int` – every non-zero integer is an integer.

This section formalizes the concept of subtyping. Subtyping of types depends on a *type context*, *viz.*. a set of constraints which may say something about types. For example:

```
class ConsTy[T,U] {
   def upcast(t:T){T <: U} :U = t;
}
```

Inside `upcast`, `T` is constrained to be a subtype of `U`, and so `T <: U` is true, and `t` can be treated as a value of type `U`. Outside of `upcast`, there is no reason to expect any relationship between them, and `T <: U` may be false. However, subtyping of types that have no free variables does not depend on the context. `Int{self != 0} <: Int` is always true, regardless of what else is going on.

Limitation *Subtyping of type variables does not currently work.*

- **Reflexivity:** Every type `T` is a subtype of itself: `T <: T`.
- **Transitivity:** If `T <: U` and `U <: V`, then `T <: V`.
- **Direct Subclassing:** Let $\vec{X}$ be a (possibly empty) vector of type variables, and $\vec{Y}$, $\vec{Y_i}$ be vectors of type terms over $\vec{X}$. Let $\vec{T}$ be an instantiation of $\vec{X}$, and $\vec{U}$, $\vec{U_i}$ the corresponding instantiation of $\vec{Y}$, $\vec{Y_i}$. Let `c` be a constraint, and `c'` be the corresponding instantiation. We elide properties, and interpret empty vectors as absence of the relevant clauses. Suppose that `C` is declared by one of the forms:
  1. $\mathtt{class\ C}[\vec{X}]\mathtt{\{c\}\ extends\ D}[\vec{Y}]\mathtt{\{d\}\ implements\ I}_1[\vec{Y_1}]\mathtt{\{i}_1\mathtt{\},\ldots,I}_n[\vec{Y_n}]\mathtt{\{i}_n\mathtt{\}\{}$
  2. $\mathtt{interface\ C}[\vec{X}]\mathtt{\{c\}\ extends\ I}_1[\vec{Y_1}]\mathtt{\{i}_1\mathtt{\},\ldots,I}_n[\vec{Y_n}]\mathtt{\{i}_n\mathtt{\}\{}$

3. `struct C[`$\vec{X}$`]{c} implements I`$_1[\vec{Y_1}]$`{i`$_1$`},...,I`$_n[\vec{Y_n}]$`{i`$_n$`}{`

Then:

1. `C[`$\vec{T}$`] <: D[`$\vec{U}$`]{d}` for a class
2. `C[`$\vec{T}$`] <: I`$_i$`[`$\vec{U_i}$`]{i`$_i$`}` for all cases.
3. `C[`$\vec{T}$`] <: C[`$\vec{T}$`]{c′}` for all cases.

- **Function types:** `(x`$_1$`: T`$_1$`, ..., x`$_n$`: T`$_n$`){c} => T throws EX`$_1$`, ...,EX`$_k$ is a subtype of `(x′`$_1$`: T′`$_1$`, ..., x′`$_n$`: T′`$_n$`){c′} => T′ throws EX′`$_1$`, ..., EX′`$_{k'}$ if:

1. Each `T`$_i$ `<: T′`$_i$;
2. `c` entails `c′`;
3. `T′ <: T`;
4. Each `EX`$_j$ `<: EX′`$_{j'}$.

- **Constrained types:** `T{c}` is a subtype of `T{d}` if `c` entails `d`.

- **Any:** Every type `T` is a subtype of `x10.lang.Any`.

- **Type Variables:** Inside the scope of a constraint `c` which entails `A <: B`, we have `A <: B`. *e.g.*, `upcast` above.

- **Covariant Generic Types:** If `C` is a generic type whose ith type parameter is covariant, and `T′`$_i$ `<: T`$_i$ and `T′`$_j$ `== T`$_j$ for all $j \neq i$, then `C[T′`$_1$`, ..., T′`$_n$`] <: C[T′`$_1$`, ..., T′`$_n$`]`. *E.g.*, `class C[T1, +T2, T3]` with $i = 2$, and `U2 <: T2`, then `C[T1,U2,T3] <: C[T1,T2,T3]`.

- **Contravariant Generic Types:** If `C` is a generic type whose ith type parameter is contravariant, and `T′`$_i$ `<: T`$_i$ and `T′`$_j$ `== T`$_j$ for all $j \neq i$, then `C[T′`$_1$`, ..., T′`$_n$`] :> C[T′`$_1$`, ..., T′`$_n$`]`. *E.g.*, `class C[T1, -T2, T3]` with $i = 2$, and `U2 <: T2`, then `C[T1,U2,T3] :> C[T1,T2,T3]`.

Two types are *equivalent*, `T == U`, if `T <: U` and `U <: T`.

4.9 Common ancestors of types

There are several situations where X10 must find a type `T` that describes values of two or more different types. This arises when X10 is trying to find a good type to describe:

- Conditional expressions, like `test ? 0 : "non-zero"` or even `test ? 0 : 1`;
- ValRail construction, like `[0, "non-zero"]` and `[0,1]`;
- Functions with multiple returns, like

  ```
  def f(a:Int) {
    if (a == 0) return 0;
    else return "non-zero";
  }
  ```

In some cases, there is a unique best type describing the expression. For example, if `B` and `C` are direct subclasses of `A`, `pick` will have return type `A`:

```
static def pick(t:Boolean, b:B, c:C) = t ? b : c;
```

However, in many common cases, there is no unique best type describing the expression. For example, consider the expression $E =$ `b ? 0 : 1`. The best type of `0` is `Int{self==0}`, and the best type of 1 is `Int{self==1}`. Certainly E could be given the type `Int`, or even `Any`, and that would describe all possible results. However, we actually know more. `Int{self != 2}` is a better description of the type of E—certainly the result of E can never be `2`. `Int{self != 2, self != 3}` is an even better description; E can't be `3` either. We can continue this process forever, adding integers which E will definitely not return and getting better and better approximations. (If the constraint sublanguage had `||`, we could give it the type `Int{self == 0 || self == 1`, which would be nearly perfect. But `||` makes typechecking far more expensive, so it is excluded.) No X10 type is the best description of E; there is always a better one.

Similarly, consider two unrelated interfaces:

```
interface I1 {}
interface I2 {}
class A implements I1, I2 {}
```

```
class B implements I1, I2 {}
class C {
  static def example(t:Boolean, a:A, b:B) = t ? a : b;
}
```

`I1` and `I2` are both perfectly good descriptions of `t ? a : b`, but neither one is better than the other, and there is no single X10 type which is better than both. (Some languages have *conjunctive types*, and could say that the return type of `example` was `I1 && I2`. This, too, complicates typechecking.)

So, when confronted with expressions like this, X10 computes *some* satisfactory type for the expression, but not necessarily the *best* type. X10 provides certain guarantees about the common type `V{v}` computed for `T{t}` and `U{u}`:

- If `T{t} == U{u}`, then `V{v} == T{t} == U{u}`. So, if X10's algorithm produces an utterly untenable type for `a ? b : c`, and you want the result to have type `T{t}`, you can (in the worst case) rewrite it to `a ? b as T{t} : c as T{t}`.
- If `T == U`, then `V == T == U`. For example, X10 will compute the type of `b ? 0 : 1` as `Int{c}` for some constraint `c`—perhaps simply picking `Int{true}`, *viz.*, `Int`.
- X10 preserves place information, because it is so important. If both `t` and `u` entail `self.home==p`, then `v` will also entail `self.home==p`. In particular, the common type for `T!` and `U!` has the form `V!`.
- X10 similarly preserves nullity information. If `t` and `u` both entail `x == null` or `x != null` for some variable `x`, then `v` will also entail it as well.

4.10 Fundamental types

Certain types are used in fundamental ways by X10.

4.10.1 The interface Any

It is quite convenient to have a type which all values are instances of; that is, a supertype of all types.[4] X10's universal supertype is the interface Any.

```
package x10.lang;
public interface Any {
  property def home():Place;
  property def at(p:Object):Boolean;
  property def at(p:Place):Boolean;
  global safe def toString():String;
  global safe def typeName():String;
  global safe def equals(Any):Boolean;
  global safe def hashCode():Int;
}
```

Any provides a handful of essential methods that make sense and are useful for everything.[5] a.toString() produces a string representation of a, and a.typeName() the string representation of its type; both are useful for debugging. aequals(b) is the programmer-overridable equality test, and a.hashCode() an integer useful for hashing. at() and home() are used in multi-place computing.

4.10.2 The class Object

The class x10.lang.Object is the supertype of all classes. A variable of this type can hold a reference to any object. Object implements Any. It also has a property home:Place, described more in §13.

4.11 Type inference

X10 v2.0 supports limited local type inference, permitting certain variable types and return types to be elided. It is a static error if an omitted type cannot be inferred or uniquely determined.

[4] Java, for one, suffers a number of inconveniences because some built-in types like int and char aren't subtypes of anything else.

[5] The behavioral annotation property is explained in §4.1.3; safe in §8.4.4, and global in §8.1.1.

4.11.1 Variable declarations

The type of a `val` variable declaration can be omitted if the declaration has an initializer. The inferred type of the variable is the computed type of the initializer. For example, `val seven = 7;` is identical to `val seven: Int{self==7} = 7;` Note that type inference gives the most precise X10 type, which might be more specific than the type that a programmer would write.

Limitation *At the moment, only* `val` *declarations can have their types elided in this way.*

4.11.2 Return types

The return type of a method can be omitted if the method has a body (*i.e.*, is not `abstract` or `native`). The inferred return type is the computed type of the body. In the following example, the return type inferred for `isTriangle` is `Boolean{self==false}`

```
class Shape {
  def isTriangle() = false;
}
```

Note that, as with other type inference, methods are given the most specific type. In many cases, this interferes with subtyping. For example, if one tried to write:

```
class Triangle extends Shape {
  def isTriangle() = true;
}
```

the X10 compiler would reject this program for attempting to override `isTriangle()` by a method with the wrong type, *viz.*, `Boolean{self==true}`. In this case, supply the type that is actually intended for `isTriangle`, such as `def isTriangle() :Boolean =false;`.

The return type of a closure can be omitted. The inferred return type is the computed type of the body.

The return type of a constructor can be omitted if the constructor has a body. The inferred return type is the enclosing class type with properties bound to the arguments in the constructor's `property` statement, if any, or to the unconstrained class type. For example, the `Spot` class has two constructors, the first of which has inferred return type `Spot{x==0}` and the second of which has inferred return type `Spot{x==xx}`.

```
class Spot(x:Int) {
  def this() {property(0);}
  def this(xx: Int) { property(xx); }
}
```

A method or closure that has expression-free `return` statements (`return;` rather than `return e;`) is said to return `Void`. `Void` is not a type; there are no `Void` values, nor can `Void` be used as the argument of a generic type. However, `Void` takes the syntactic place of a type. A method returning `Void` can be specified by `def m():Void`:

```
val f : () => Void = () => {return;};
```

By a convenient abuse of language, `Void` is sometimes lumped in with types; *e.g.*, we may say "return type of a method" rather than the formally correct but rather more awkward "return type of a method, or `Void`". Despite this informal usage, `Void` is not a type. For example, given

```
static def eval[T] (f:()=>T):T = f();
```

The call `eval[Void](f)` does *not* typecheck; `Void` is not a type and thus cannot be used as a type argument. There is no way in X10 to write a generic function which works with both functions which return a value and functions which do not. In most cases, functions which have no sensible return value can be provided with a dummy return value.

4.11.3 Type arguments

Limitation *This does not seem to work at all currently.*

A call to a polymorphic method may omit the explicit type arguments. If the method has a type parameter `T`, the type argument corresponding to `T` is inferred to be the least common ancestor of the types of any formal parameters of type `T`.

Consider the following method:

```
def choose[T](a: T, b: T): T { ... }
```

Given `Set[T] <: Collection[T]`, `List[T] <: Collection[T]`, and `SubClass <: SuperClass`, in the following snippet, the algorithm will infer the type `Collection[Any]` for `x`.

```
def m(intSet: Set[Int], stringList: List[String]) {
  val x = choose(intSet, stringList);
...
}
```

And in this snippet, the algorithm should infer the type `Collection[Int]` for `y`.

```
def m(intSet: Set[Int], intList: List[Int]) {
  val y = choose(intSet, intList);
  ...
}
```

Finally, in this snippet, the algorithm should infer the type `Collection{T <: SuperClass}` for `z`.

```
def m(intSet: Set[SubClass], numList: List{T <: SuperClass}) {
  val z = choose(intSet, numList);
  ...
}
```

5 Variables

A *variable* is an X10 identifier associated with a value within some context. Variable bindings have these essential properties:

- **Type:** What sorts of values can be bound to the identifier;
- **Scope:** The region of code in which the identifier is associated with the entity;
- **Lifetime:** The interval of time in which the identifier is associated with the entity.
- **Visibility:** Which parts of the program can read or manipulate the value through the variable.

X10 has many varieties of variables, used for a number of purposes. They will be described in more detail in this chapter.

- Class variables, also known as the static fields of a class, which hold their values for the lifetime of the class.
- Instance variables, which hold their values for the lifetime of an object;
- Array elements, which are not individually named and hold their values for the lifetime of an array;
- Formal parameters to methods, functions, and constructors, which hold their values for the duration of method (etc.) invocation;
- Local variables, which hold their values for the duration of execution of a block.

- Exception-handler parameters, which hold their values for the execution of the exception being handled.

A few other kinds of things are called variables for historical reasons; *e.g.*, type parameters are often called type variables, despite not being variables in this sense because they do not refer to X10 values. Other named entities, such as classes and methods, are not called variables. However, all name-to-whatever bindings enjoy similar concepts of scope and visibility.

In the following example, `n` is an instance variable, and `next` is a local variable defined within the method `bump`.[1]

```
class Counter {
  private var n : Int = 0;
  public def bump() : Int {
    val next = n+1;
    n = next;
    return next;
    }
}
```

Both variables have type `Int` (or perhaps something more specific). The scope of `n` is the body of `Counter`; the scope of `next` is the body of `bump`. The lifetime of `n` is the lifetime of the `Counter` object holding it; the lifetime of `next` is the duration of the call to `bump`. Neither variable can be seen from outside of its scope.

Variables whose value may not be changed after initialization are said to be *immutable*, or *constants* (§5.1), or simply `val` variables. Variables whose value may change are *mutable* or simply `var` variables. `var` variables are declared by the `var` keyword. `val` variables may be declared by the `val` keyword; when a variable declaration does not include either `var` or `val`, it is considered `val`.

```
val a : Int = 0;                // Full 'val' syntax
b : Int = 0;                    // 'val' implied
val c = 0;                      // Type inferred
var d : Int = 0;                // Full 'var' syntax
var e : Int;                    // Not initialized
var f : Int{self != 100} = 0;   // Constrained type
```

[1]This code is unnecessarily turgid for the sake of the example. One would generally write `public def bump() = ++n;`.

5.1 Immutable variables

Immutable variables can be given values (by initialization or assignment) at most once, and must be given values before they are used. Usually this is achieved by declaring and initializing the variable in a single statement.

```
val a : Int = 10;
val b = (a+1)*(a-1);
```

`a` and `b` cannot be assigned to further.

In other cases, the declaration and assignment are separate. One such case is how constructors give values to `val` fields of objects. The `Example` class has an immutable field `n`, which is given different values depending on which constructor was called. `n` can't be given its value by initialization when it is declared, since it is not knowable which constructor is called at that point.

```
class Example {
  val n : Int; // not initialized here
  def this() { n = 1; }
  def this(dummy:Boolean) { n = 2;}
}
```

Another common case of separating declaration and assignment is in function and method call. The formal parameters are bound to the corresponding actual parameters, but the binding does not happen until the function is called. In the code below, `x` is initialized to `3` in the first call and `4` in the second.

```
val sq = (x:Int) => x*x;
x10.io.Console.OUT.println("3 squared = " + sq(3));
x10.io.Console.OUT.println("4 squared = " + sq(4));
```

5.2 Initial values of variables

Every assignment, binding, or initialization to a variable of type `T{c}` must be an instance of type `T` satisfying the constraint `{c}`. Variables must be given a value before they are used. This may be done by initialization, which is the only way for immutable (`val`) variables and one option for mutable (`var`) ones:

```
val immut : Int = 3;
var mutab : Int = immut;
val use = immut + mutab;
```

Or, for mutable variables, it may be done by a later assignment.

```
var muta2 : Int;
muta2 = 4;
val use = muta2 * 10;
```

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. Mutable instance variables of class type are initialized to to `null`. Mutable instance variables of struct type are assumed to have an initializer that sets the value to the result of invoking the nullary constructor on the class. An initializer is required if the default initial value of the variable's type is not assignable to the variable's type, *e.g.*, `Int` variables are initialized to zero, but that doesn't work for `val x:Int{x!=0}`.

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 [5, § 16].

5.3 Destructuring syntax

X10 permits a *destructuring* syntax for local variable declarations and formal parameters of type `Point`, §16.1. (Future versions of X10 may allow destructuring of other types as well.) A point is a sequence of $r \geq 0$ `Int`-valued coordinates. It is often useful to get at the coordinates directly, in variables.

The following code makes an anonymous point with one coordinate `11`, and binds `i` to `11`. Then it makes a point with coordinates `22` and `33`, binds `p` to that point, and `j` and `k` to `22` and `33` respectively.

```
val (i) : Point = Point.make(11);
val p(j,k) = Point.make(22,33);
```

A useful idiom for iterating over a range of numbers is:

```
var sum : Int = 0;
for ((i) in 1..100) sum += i;
```

The parentheses in `(i)` introduce destructuring, making X10 treat `i` as an `Int`; without them, it would be a `Point`.

In general, a pattern of the form $(\mathtt{i}_1,\ldots,\mathtt{i}_n)$ matches a point with n coordinates, binding $\mathtt{i}_j$ to coordinate j. A pattern of the form $\mathtt{p}(\mathtt{i}_1,\ldots,\mathtt{i}_n)$ does the same, , but also binds `p` to the point.

5.4 Formal parameters

Formal parameters are the variables which hold values transmitted into a method or function. They are always declared with a type. (Type inference is not advisable, because there is no single expression to deduce a type from.) The variable name can be omitted if it is not to be used in the scope of the declaration, as in the type of the method `static def main(Rail[String])` executed at the start of a program that does not use its command-line arguments.

Formal	::=	*FormalModifier** `var` *VarDeclaratorWithType*
	|	*FormalModifier** `val` *VarDeclaratorWithType*
	|	*FormalModifier** *VarDeclaratorWithType*
	|	*Type*
FormalModifier	::=	*Annotation*
	|	`shared`

`var`, `val`, and `shared` behave just as they do for local variables, §5.5. In particular, the following `inc` method is allowed, but, unlike some languages, does *not* increment its actual parameter. `inc(j)` creates a new local variable `i` for the method call, initializes `i` with the value of `j`, increments `i`, and then returns. `j` is never changed.

```
static def inc(var i:Int) { i += 1; }
```

5.5 Local variables

Local variables are declared in a limited scope, and, dynamically, keep their values only for so long as the scope is being executed. They may be `var` or `val`. They may have initializer expressions: `var i:Int = 1;` introduces a variable `i` and initializes it to 1. If the variable is immutable (`val`) the type may be omitted and inferred from the initializer type (§4.11). Variables marked `shared` can be used by many activities at once; see §14.10.

The variable declaration `val x:T=e;` confirms that `e`'s value is of type `T`, and then introduces the variable `x` with type `T`. For example,

```
val t : Tub = new Tub();
```

produces a variable `t` of type `Tub`, even though the expression `new Tub()` produces a value of type `Tub!` – that is, a `Tub` located `here`. This can be inconvenient if, *e.g.*, it is desired to make method calls upon `t`.

Including type information in variable declarations is generally good programming practice: it explains to both the compiler and human readers something of the intent of the variable. However, including types in `val t:T=e` can obliterate helpful information. So, X10 allows a *documentation type declaration*, written `val t <: T = e`. This has the same effect as `val t = e`, giving `t` the full type inferred from `e`; but it also confirms statically that that type is at least `T`. For example, the following gives `t` the type `Tub!` as desired:

```
val t <: Tub = new Tub();
```

However, replacing `Tub` by `Int` would result in a compilation error.

Variables do not need to be initialized at the time of definition – not even `val`s. They must be initialized by the time of use, and `val`s may only be assigned to once. The X10 compiler performs static checks guaranteeing this restriction. The following is correct, albeit obtuse:

```
static def main(r: Rail[String]) {
  val a : Int;
  a = r.length;
  val b : String;
  if (a == 5) b = "five?"; else b = "" + a + " args";
  // ...
```

5.6 Fields

Like most other kinds of variables in X10, the fields of an object can be either `val` or `var`. Fields can be `static`,`global`, or `property`; see §8.2 and §8.3. Field declarations may have optional initializer expressions, as for local variables, §5.5. `var` fields without an initializer are initialized with the default value of their type. `val` fields without an initializer must be initialized by each constructor.

For `val` fields, as for `val` local variables, the type may be omitted and inferred from the initializer type (§4.11). `var` files, like `var` local variables, must be declared with a type.

FieldDeclaration	::=	*FieldModifier** `var` *FieldDeclaratorsWithType* (`,` *FieldDeclaratorsWithType*)*
	|	*FieldModifier** `val` *FieldDeclarators* (`,` *FieldDeclarators*)*
	|	*FieldModifier** *FieldDeclaratorsWithType* (`,` *FieldDeclaratorsWithType*)*
FieldDeclarators	::=	*FieldDeclaratorsWithType*
	::=	*FieldDeclaratorWithInit*
FieldDeclaratorId	::=	*Identifier*
FieldDeclaratorWithInit	::=	*FieldDeclaratorId Init*
	|	*FieldDeclaratorId ResultType Init*
FieldDeclaratorsWithType	::=	*FieldDeclaratorId* (`,` *FieldDeclaratorId*)* *ResultType*
FieldModifier	::=	*Annotation*
	|	`static`
	|	`property`
	|	`global`

6 Names and packages

X10 supports mechanisms for names and packages in the style of Java [5, §6,§7], including `public`, `protected`, `private` and package-specific access control.

6.1 Packages

A package is a named collection of top-level type declarations, *viz.*, class, interface, and struct declarations. Package names are sequences of identifiers, like `x10.lang` and `com.ibm.museum`. The multiple names are simply a convenience. Packages `a`, `a.b`, and `a.c` have only a very tenuous relationship, despite the similarity of their names.

Packages and protection modifiers determine which top-level names can be used where. Only the `public` members of package `pack.age` can be accessed outside of `pack.age` itself.

```
package pack.age;
class Deal {
  public def make() {}
}
public class Stimulus {
  private def taxCut() = true;
  protected def benefits() = true;
  public def jobCreation() = true;
  /*package*/ def jumpstart() = true;
}
```

The class `Stimulus` can be referred to from anywhere outside of `pack.age` by its full name of `pack.age.Stimulus`, or can be imported and referred to simply as

`Stimulus`. The public `jobCreation()` method of a `Stimulus` can be referred to from anywhere as well; the other methods have smaller visibility. The non-`public` class `Deal` cannot be used from outside of `pack.age`.

6.1.1 Name Collisions

It is a static error for a package to have two members, or apparent members, with the same name. For example, package `pack.age` cannot define two classes both named `Crash`, nor a class and an interface with that name.

Furthermore, `pack.age` cannot define a member `Crash` if there is another package named `pack.age.Crash`, nor vice-versa. (This prohibition is the only actual relationship between the two packages.) This prevents the ambiguity of whether `pack.age.Crash` refers to the class or the package. Note that the naming convention that package names are lower-case and package members are capitalized prevents such collisions.

6.2 `import` Declarations

Any public member of a package can be referred to from anywhere through a fully-qualified name: `pack.age.Stimulus`.

Often, this is too awkward. X10 has two ways to allow code outside of a class to refer to the class by its short name (`Stimulus`): single-type imports and on-demand imports.

Imports of either kind appear at the start of the file, immediately after the `package` directive if there is one; their scope is the whole file.

6.2.1 Single-Type Import

The declaration `import` *TypeName* `;` imports a single type into the namespace. The type it imports must be a fully-qualified name of an extant type, and it must either be in the same package (in which case the `import` is redundant) or be declared `public`.

Furthermore, when importing `pack.age.T`, there must not be another type named `T` at that point: neither a `T` declared in `pack.age`, nor a `inst.ant.T` imported from some other package.

6.2.2 Automatic Import

The automatic import `import pack.age.*;`, loosely, imports all the public members of `pack.age`. In fact, it does so somewhat carefully, avoiding certain errors that could occur if it were done naively. Types defined in the current package, and those imported by single-type imports, shadow those imported by automatic imports.

6.2.3 Implicit Imports

The packages `x10.lang` and `x10.array` are imported in all files without need for further specification.

6.3 Conventions on Type Names

TypeName	::=	*Identifier*
	|	*TypeName* `.` *Identifier*
	|	*PackageName* `.` *Identifier*
PackageName	::=	*Identifier*
	|	*PackageName* `.` *Identifier*

While not enforced by the compiler, classes and interfaces in the X10 library follow the following naming conventions. Names of types—including classes, type parameters, and types specified by type definitions—are in CamelCase and begin with an uppercase letter. (Type variables are often single capital letters, such as `T`.) For backward compatibility with languages such as C and Java, type definitions are provided to allow primitive types such as `int` and `boolean` to be written in lowercase. Names of methods, fields, value properties, and packages are in camelCase and begin with a lowercase letter. Names of `static val` fields are in all uppercase with words separated by '`_`''s.

7 Interfaces

X10 v2.0 interfaces are generally modelled on Java interfaces [5, §9]. An interface specifies signatures for public methods, properties, `static val`s, and an invariant. It may extend several interfaces, giving X10 a large fraction of the power of multiple inheritance at a tiny fraction of the cost.

The following puny example illustrates all these features:

```
interface Pushable(text:String, prio:Int) {
  def push(): Void;
  static val MAX_PRIO = 100;
}
class MessageButton(text:String, prio:Int)
  implements Pushable{self.prio==Pushable.MAX_PRIO} {
  public def push() {
    x10.io.Console.OUT.println(text + " pushed");
  }
}
```

`Pushable` defines two properties, a method, and a static value. `MessageButton` implements a constrained version of `Pushable`, *viz.* one with maximum priority. It also has `Pushable`'s properties. It defines the `push()` method given in the interface, as a `public` method—interface methods are implicitly `public`.

A concrete type—a class or struct—can *implement* an interface, typically by having all the methods and properties that the interface requires.

A variable may be declared to be of interface type. Such a variable has all the fields and methods declared (directly or indirectly) by the interface; nothing else is statically available. Values of a concrete type which implement the interface may be stored in the variable.

NormalInterfaceDeclaration ::= *InterfaceModifiers*$^?$ `interface` *Identifier* *TypePropertyList*$^?$ *PropertyList*$^?$ *Constraint*$^?$ *ExtendsInterfaces*$^?$ *InterfaceBody*

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.

Each interface implicitly defines a nullary getter method `def p(): T` for each property `p: T`. The interface may not have another definition of a method `p()`.

A class `C` is said to implement an interface `I` if

- `I`, or a subtype of `I`, appears in the `implements` list of `C`,
- `C`'s properties include all the properties of `I`,
- `C`'s class invariant inv(`C`) implies inv(`I`).
- Each method `m` defined by `I` is also a method of `C` – *with the* `public` modifier added. These methods may be `abstract` if `C` is `abstract`.

7.1 Field Definitions

An interface may declare a `val` field, with a value. This field is implicitly `public static val`:

```
interface KnowsPi {
  PI = 3.14159265358;
}
```

Classes and structs implementing such an interface get the interface's fields as `public static` fields. Unlike properties and methods, there is no need for the implementing class to declare them.

```
class Circle implements KnowsPi {
  static def area(r:Double) = PI * r * r;
}
```

7.1.1 Fine Points of Fields

It can happen that two parent interfaces give fields of the same name. In that case, those fields must be referred to by qualified names.

```
interface E1 {static val a = 1;}
interface E2 {static val a = 2;}
interface E3 extends E1, E2{}
class Example implements E3 {
  def example() = E1.a + E2.a;
}
```

If the *same* field `a` is inherited through many paths, there is no need to disambiguate it:

```
interface I1 { static val a = 1;}
interface I2 extends I1 {}
interface I3 extends I1 {}
interface I4 extends I2,I3 {}
class Example implements I4 {
  def example() = a;
}
```

7.2 Interfaces Specifying Properties

Interfaces may specify properties.

8 Classes

8.1 Principles of X10 Objects

8.1.1 Basic Design

Objects are instances of classes: the most common and most powerful sort of value in X10. The other kinds of values, structs and functions, are more specialized, better in some circumstances but not in all. `x10.lang.Object` is the most general class; all other classes inherit from it, directly or indirectly.

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.

Unlike Java, X10 objects do not have locks associated with them. Programmers should use atomic blocks (§14.11) for mutual exclusion and clocks (§15) for sequencing multiple parallel operations.

`Object` has one property `home` of type `x10.lang.Place`, telling where the object is located. Objects stay resident at the place at which they were created for their entire lifetime. However, the programmer may designate certain immutable field of an object as `global`, accessible from everywhere.

Activities at places other than `ob.home` are allowed to have references to object `ob`. These *remote references* do not grant all the privileges of a local reference; only `ob`'s `global` methods and fields are usable remotely. 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 carries with it the values of `global val` fields of the object. The implementation should try to ensure that the cost of communicating the values of `val` 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 (§11.4). All fields can be accessed locally; only `global` fields can be accessed remotely. `global` fields are more expensive than non-`global` ones, and should be used sparingly as needed. Properties are global fields.

- Method invocation (§11.6). All methods can be accessed locally; only `global` methods can be accessed remotely. `global` methods are no more expensive than local ones, and can be used freely where possible. However, `global` methods can only perform a subset of operations (§8.4.4).

- Casting (§11.22) and instance testing with `instanceof` (§11.23) Objects can be cast or tested either locally or remotely. These operations involve the class and the object's properties, both of which are available globally.

- The equality operators == and `!=` (§11.20). On creation, each object is associated with a globally unique identifier. Two object references are == iff they refer to objects with the same identifier.

8.1.2 Distributed Object Model

The state of an object is partitioned into *global* state (a programmer defined subset of `val` fields, §8.2.3) and *non-global* state. Global state is available from anywhere without communication at the time of reference; non-global state is only available directly from the object's home place.

- Field definitions are marked with the qualifier `global` if they are intended to be included in the global state.

- If the `global` qualifier is omitted, the field is considered non-global.

- Properties and static fields are implicitly marked `global`.
- `var` fields cannot be marked `global`.

Similarly, the methods of an object may be qualified as `global`(§8.4.4) ; if they are not global they are said to be *non-global*. Global methods cannot be overridden by non-global methods.

Consider the execution of an `at (P) S` statement at a place `Q` different from `P`. Suppose `x` is an in-scope immutable local variable and contains a reference to an object `o` created at `Q`. Then within `S`, `x` is said to be a *remote reference* to `o`. References to `o` from place `Q` are said to be *local references*. X10 permits `global` fields to be read and `global` methods to be invoked through a remote reference.

Like local references, remote references are first-class entities: they may be passed as arguments to methods, returned from methods, stored in fields of objects.

Remote references may also be compared for equality (==). Two remote reference are equal if they are references to the same object. Equality is guaranteed to be a constant-time operation and not involve any communication.

When a remote reference to an object `o` located at place `P` is transmitted to `P` it automatically becomes a local reference to `o`. Therefore the situation in which a local reference can be compared to a remote reference simply cannot arise.

The X10 compiler ensures that non-global methods on `o` can only be invoked in a place where `here == o.home()`, i.e. the place where `o` was created.

Implementation notes Remote references to an object `o` are intended to be implemented by serializing the global state of `o` and sending it across the network, together with a globally unique id. The data is deserialized at the receiver to create an implementation-level entity that is the remote reference. There is no requirement that a remote reference use only as much space as a local reference; *e.g.*, it is possible that remote references have space for global data, and local references do not.

8.1.3 Examples of Local and Remote References

Given the class declarations:

```
class C { }
class D {
   var f:C=null;
}
```

Let `P` and `Q` be different places, neither equal to `here`. Then all of the `truly(b)`'s in the following code have `b==true`. The locality of references is described in comments.

```
val x = new C();
// C object o1 created, local reference stored in x.
at (P) {
      // In the body x contains a remote reference to o1
      val d = new D();
      d.f  = x; // remote reference stored in d.f
      truly(d.f == x);
      truly(x == x);
      at (Q) {
         // x continues to be a remote reference to o1.
         at (P) {
             truly(d.f == x);
             truly(x == x);
         }
      }
}
```

Here is another example.

```
val x = new C();
// C object o created, reference stored in x.
// The type inferred for x is C!
at (P) {
    val x1 = x;
    // The type inferred for x1 is C, not C!;
    // the change is due to the place shift 'at(P)'
  at (x.home) {
        // x is now bound to o through a local reference. So is x1.
    Console.OUT.println(x1==x); // Must print true.
      // non-global methods can be invoked on x or x1 and will
```

```
        // execute locally on o
        // type of both x and x1 is C!.
    }
}
```

8.1.4 Class Declaration Syntax

The *class declaration* has a list of type parameters, properties, a constraint (the *class invariant*), a single superclass, zero or more interfaces, and a class body containing the the definition of fields, properties, methods, and member types. Each such declaration introduces a class type (§4.1).

8.2 Fields

Objects may have *instance fields*, or simply *fields*: places to store data that is pertinent to the object. Fields, like variables, may be either mutable (`val`) or immutable (`var`).

Class may have *static fields*, which store data pertinent to the entire class of objects. Static fields are global, *viz.* they may be accessed from any place. Like all globals, static fields must be immutable (`val`). See §8.7 for more information.

No two fields of the same class may have the same name. To avoid an ambiguity, it is a static error for a class to declare a field with a function type (§4.6) with the same name and signature as a method of the same class. (Consider the class

```
class Crash {
  val f : (Int) => Boolean = (Int)=>true;
  def f(Int) = false;
}
```

Then `crash.f(3)` might either mean "call the function `crash.f` on argument 3", or "invoke the method `f` on argument 3".)

8.2.1 Field Initialization

Fields may be given values via *field initialization expressions*: `val f1 = E;` and `var f2 : Int = F;`. There are a few restrictions on the initializer expressions

`E`:

- No checked exceptions may be thrown. (There is no place to put a `try-catch` block to deal with them.)
- Other fields of `this` may be referenced, but only those that *precede* the field being initialized. For example, this is correct, but would not be if the fields were reversed:

```
class Fld{
  val a = 1;
  val b = 2+a;
}
```

8.2.2 Field hiding

A subclass that defines a field `f` hides any field `f` declared in a superclass, regardless of their types. The superclass field `f` may be accessed within the body of the subclass via the reference `super.f`.

```
class Super{
  val f = 1;
}
class Sub extends Super {
  val f = true;
  def superf() : Int = super.f; // 1
}
```

With inner classes, it is occasionally necessary to write `Cls.super.f` to get at a hidden field `f` of an outer class `Cls`, as in

```
class A {
   global val f = 3;
}
class B extends A {
   global val f = 4;
   class C extends B {
      val f = 5;
      def foo()
```

```
            = f          // 5
            + super.f    // 4
            + B.this.f   // 4 (the "f" of the outer instance)
            + B.super.f; // 3 (the "super.f" of the outer instance)
        }
    }
```

8.2.3 Field qualifiers

The behavior of a field may be changed by a field qualifier, such as `static` or `global`.

`global` qualifier

A `val` field may be declared `global`. Global fields are available at all places, and should be thought of as being part of remote object references to the object. Global fields are good for distributing data, but increase space and communication costs; they should be used sparingly as required.

FieldModifier ::= `global`

Properties and static fields are implicitly `global`. Fields not marked `global` cannot be overridden by fields marked `global`.

`static` qualifier

A `val` field may be declared to be *static*, as described in §8.2.

8.3 Properties

The properties of an object (or struct) are global, public `val` fields usable at compile time in constraints.[1] Every object has a `home` property, telling what place

[1] In many cases, a `global val` field can be upgraded to a `property`, which entails no compile-time or runtime cost. Some cannot be, *e.g.*, in cases where cyclic structures of `val` fields are required.

the object lives in. Every array has a `rank` telling how many subscripts it takes. User-defined classes can have whatever properties are desired.

Properties are defined in parentheses, after the name of the class. They are given values by the `property` command in constructors.

```
class Proper(t:Int) {
  def this(t:Int) {property(t);}
}
```

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

A property `x:T` induces a field with the same name and type, as if defined with:

```
public global val x : T;
```

It also defines a nullary getter method,

```
public global final def x()=x;
```

(As noted in §7, interfaces can define properties too. They define the same nullary getter methods, though they do not require fields.)

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

Properties are initialized by the invocation of a special `property` statement, which must be performed in each constructor of the class:

```
property(e1,..., en);
```

The number and types of arguments to the `property` statement must match the number and types of the properties in the class declaration. Every constructor of a class with properties must invoke `property(...)` precisely once; it is a static error if X10 cannot prove that this holds.

The requirement to use the `property` statement means that all properties must be given values at the same time.

By construction, the graph whose nodes are values and whose edges are properties is acyclic. *E.g.*, there cannot be values `a` and `b` with properties `c` and `d` such that `a.c == b` and `b.d == a`. (The similar graph whose edges are `global public val` fields *can* have cycles..)

Class declarations may be used to construct class types (§4.1). Classes may have mutable fields. Instances of a class are always created in a fixed place and in X10 v2.0 stay there for the lifetime of the object. Variables declared at a class type always store a reference to the object, regardless of whether the object is local or remote.

8.4 Methods

As is common in object-oriented languages, objects can have *methods*, of two sorts. *Static methods* are functions, conceptually associated with a class and defined in its namespace. *Instance methods* are parameterized code bodies associated with an instance of the class, which execute with priveleged access to that instance's fields.

Each method has a *signature*, telling what arguments it accepts, what type it returns, what precondition it requires, and what exceptions it may throw. 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). Methods may be declared `public`, `private`, `protected`, or given default access rights.

MethodDeclaration	::=	*MethodHeader* `;`
	|	*MethodHeader* = *ClosureBody*
MethodHeader	::=	*MethodModifiers*$^?$ `def` *Identifier* *TypeParameters*$^?$ `(` *FormalParameterList*$^?$ `)` *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

A formal parameter may have a `val` or `var` modifier; `val` is the default. The body of the method is executed in an environment in which each formal parameter corresponds to a local variable (`var` iff the formal parameter is `var`) and is initialized with the value of the actual parameter.

8.4.1 Method Guards

Often, a method will only make sense to invoke under certain statically-determinable conditions. For example, `example(x)` is only well-defined when `x != null`, as `null.toString()` throws a null pointer exception:

```
class Example {
   var f : String = "";
   def example(x:Object){x != null} = {
      this.f = x.toString();
   }
}
```

(We could have used a constrained type `Object{self!=null}` instead; in most cases it is a matter of personal preference or convenience of expression which one to use.)

The requirement of having a method guard is that callers must demonstrate to the X10 compiler that the guard is satisfied. (As usual with static constraint checking, there is no runtime cost. Indeed, this code can be more efficient than usual, as it is statically provable that `x != null`.) This may require a cast:

```
def exam(e:Example!, x:Object) {
  if (x != null)
     e.example(x as Object{x != null});
  // WRONG: if (x != null) e.example(x);
}
```

The guard `{c}` in a guarded method `def m(){c} = E;` specifies 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}`.

STATIC SEMANTICS RULE: The compiler checks that every method invocation $\mathtt{o.m(e_1, \ldots, e_n)}$ for a method is type correct. Each argument $\mathtt{e}_i$ must have a static type $\mathtt{S}_i$ that is a subtype of the declared type $\mathtt{T}_i$ for the ith argument of the method, and the conjunction of the constraints on the static types of the arguments must entail the guard in the parameter list of the method.

The compiler checks that in every method invocation $\mathtt{o.m(e_1, \ldots, e_n)}$ the static type of `o`, `S`, is a subtype of `C{c}`, where the method is defined in class `C` and the guard for `m` is equivalent to `c`.

Finally, if the declared return type of the method is `D{d}`, the return type computed for the call is $\mathtt{D\{a\colon\ S;\ x_1\colon\ S_1;\ \ldots;\ x_n\colon\ S_n;\ d[a/this]\}}$, where `a` is a new variable that does not occur in $\mathtt{d,\ S,\ S_1,\ \ldots,\ S_n}$, and $\mathtt{x_1,\ \ldots,\ x_n}$ are the formal parameters of the method.

8.4.2 Property methods

A method declared with the modifier `property` may be used in constraints. A property method declared in a class must have a body and must not be `Void`. The body of the method must consist of only a single `return` statement or a single expression. It is a static error if the expression cannot be represented in the constraint system.

The expression may contain invocations of other property methods. It is the responsibility of the programmer to ensure that the evaluation of a property terminates at compile-time, otherwise the type-checker will not terminate and the program will fail to compile in a potentially most unfortunate way.

Property methods in classes are implicitly `final`; they cannot be overridden. Property methods are also implicitly `global`.

A nullary property method definition may omit the formal parameters and the `def` keyword. That is, the following are equivalent:

```
property def rail(): Boolean = rect && onePlace == here && zeroBased;
```

and

```
property rail: Boolean = rect && onePlace == here && zeroBased;
```

Similarly, nullary property methods can be inspected in constraints without `()`. `w.rail`, with either definition above, is equivalent to `w.rail()`

8.4.3 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 type parameters and dependent types.

Two or more methods of a class or interface may have the same name if they have a different number of type parameters, or they have formal parameters of different types. *E.g.*, the following is legal:

```
class Mful{
   def m() = 1;
   def m[T]() = 2;
```

```
    def m(x:Int) = 3;
    def m[T](x:Int) = 4;
}
```

X10 v2.0 does not permit overloading based on constraints. That is, the following is *not* legal, although either method definition individually is legal:

```
    def n(x:Int){x==1} = "one";
    def n(x:Int){x!=1} = "not";
```

The definition of a method declaration $\mathtt{m}_1$ "having the same signature as" a method declaration $\mathtt{m}_2$ involves identity of types.

The *constraint erasure* of a type T is defined as follows. The constraint erasure of (a) a class, interface or struct type T is T; (b) a type T{c} is the constraint erasure of T; (b) a type $\mathtt{T[S_1,\backslash ldots,S}_n\mathtt{]}$ is $\mathtt{T'[S_1',\backslash ldots,S}_n\mathtt{']}$ where each primed type is the erasure of the corresponding unprimed type. Two methods are said to have *the same signature* if (a) they have the same number of type parameters, (b) they have the same number of formal (value) parameters, and (c) for each formal parameter the constraint erasure of its types 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:

```
def m[X1, ..., Xm](v1: T1, ..., vn: Tn){tc}: T {...}
def m[X1, ..., Xm](v1: S1, ..., vn: Sn){sc}: S {...}
```

if it is the case that the constraint erasures of the types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ are equivalent to the constraint erasures of the types $\mathtt{S}_1,\ \ldots,\ \mathtt{T}_n$ respectively.

In addition, the guard of a overriding method must be no stronger than the guard of the overridden method. This ensures that any virtual call to the method satisfies the guard of the callee.

STATIC SEMANTICS RULE: If a class C overrides a method of a class or interface B, the guard of the method in B must entail the guard of the method in C.

A class C inherits from its direct superclass and superinterfaces all their methods visible according to the access modifiers of the superclass/superinterfaces that are not hidden or overridden. A method $\mathtt{M}_1$ in a class C overrides a method $\mathtt{M}_2$ in a superclass D if $\mathtt{M}_1$ and $\mathtt{M}_2$ have the same signature. Methods are overriden on a signature-by-signature basis.

A method invocation $\texttt{o.m(e}_1\texttt{, ..., e}_n\texttt{)}$ is said to have the *static signature* $\texttt{<T, T}_1\texttt{, ..., T}_n\texttt{>}$ where T is the static type of o, and $\texttt{T}_1, \ldots, \texttt{T}_n$ are the static types of $\texttt{e}_1, \ldots, \texttt{e}_n$, respectively. As in Java, it must be the case that the compiler can determine a single method defined on T with argument type $\texttt{T}_1, \ldots \texttt{T}_n$; 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.

8.4.4 Method qualifiers

There are a number of qualifiers which may be applied to X10 methods.

MethodModifier	::=	`atomic`
MethodModifier	::=	`global`
MethodModifier	::=	`pinned`
MethodModifier	::=	`nonblocking`
MethodModifier	::=	`sequential`
MethodModifier	::=	`safe`

`atomic` qualifier

A method may be declared `atomic`, indicating that it will be executed atomically—as if its body were wrapped in an `atomic` statement.

`global` qualifier

A method may be declared `global`, indicating that it can be invoked in any place. Unlike non-global methods, the body of a global method is type-checked without assuming that `here==this.home`. This permits `global` fields of `o` to be accessed, but not local fields. The programmer must insert an explicit `at(this)...` to get to the place where the object lives to access local fields.

`global` methods can be overridden only by methods also marked `global`.

`pinned` qualifier

A method may be declared `pinned`, indicating that the evaluation of the method takes place entirely `here`, without any communication necessary. A `pinned` method may not contain any `at` statement or expression whose place argument is not statically equivalent to `here`. It must call only `pinned` methods.

`pinned` methods can be overridden only by methods marked `pinned`.

`nonblocking` qualifier

A method may be declared `nonblocking`, indicating that it does not block. A `nonblocking` method may not contain any `when` statement whose condition is not statically equivalent to `true`. It must call only `nonblocking` methods.

`nonblocking` methods can be overridden only by methods marked `nonblocking`.

`sequential` qualifier

A method may be declared `sequential`, indicating that it does not spawn any other activities. A `sequential` method may not contain any `async` statement. It must call only `sequential` methods.

`sequential` methods can be overridden only by methods marked `sequential`.

`safe` qualifier

A method may be declared `safe`, indicating that it is `pinned`, `nonblocking`, and `sequential`. Safe methods are the only methods which can be called inside of `atomic` and `when` statements.

8.5 Instance Initialization

8.6 Constructors

Constructors allow the initialization of objects by the execution of almost-arbitrary code. Like methods, constructors can have formal parameters, a constraint, a return type, a `throws` clause, and a body. The formals, constraint, and `throws` clauses are identical to those for a method. A constructor is declared by `def this()...`; that is, as if it were a method whose name were the reserved word `this`.

8.6.1 Constructor Return Types

The return type of a constructor describes the values that constructor can create. While all constructors for class `C` create objects of base class `C`, some individual constructors may construct objects with more specific constraints. For example, in

```
class Crate(n:Int) {
  def this() : Crate{self.n==0} = { property(0); }
  def this(b:Boolean) : Crate{self.n==1} = { property(1); }
}
```

the nullary constructor call `new Crate()` will return a value of type `Crate{self.n == 0}`— the `n` field is zero and the compiler knows it. The unary Boolean constructor will return an object of type `Crate{self.n==1}`. A less trivial example might be a specialized constructor for a square matrix, which returned type `Matrix{self.rows==self.cols}`.

If the constructor type is omitted, the constructor returns the type of its class, constrained by the actual parameters to the `property` call in the constructor. That is, the first constructor call above could be abbreviated:

```
def this() { property(0); }
// And to prove that the nullary constructor knows n==0:
static def confirm() {
  val v : Crate{self.n == 0} = new Crate();
}
```

8.6.2 Constructor Bodies

Constructors have many restrictions, designed to ensure that objects behave sanely while being constructed.

8.6.3 `proto` qualifier on types

X10 ensures that every variable must have a value consistent with its type before it is read. This is to prevent the unfortunate situations possible in other languages, such as:

```
class Evil {
   val f : Int; // uninitialized
   def getf() = this.f;
   def this() {
       val f0 = this.getf();
       println("initially, f0=" + f0);
       this.f = f0 + 1;
       println("now, this.f =" + this.f);
   }
}
```

`f` is a `val` field and should never change. However, in this example, it starts out uninitialized. Its uninitialized value is read by the call `this.getf()`, giving a meaningless value—no value could be meaningful, since the field is not initialized. The constructor prints `f`'s initial value, increments it, initializes `f` to the incremented value, and prints the new value. This program prints two different numbers. That is, a `val` field—supposedly immutable—has been observed to change.

This approach can be extended to give almost arbitrarily bad behavior for partially-initialized objects. The issue is particularly pernicious with concurrency, where one activity initializes an object while another one uses it.

Many languages with constructors simply admit (or avoid admitting) that method calls on partially-initialized objects are dangerous and should be avoided. X10 takes a more careful approach: some method calls in constructors are allowed, but ones which could cause this problem are not.

The approach is based on `proto` types. If `C` is a class, `proto C` is a type that describes partially-initialized values of `C`. The rules in this section are designed to allow methods to *write* into fields of a `proto C` object, thus initializing it, but not *read* them and thus not stumble into the error above.

For local variables, this is ensured by using a pre-specified static analysis to ensure that every local variable is written into before it is read. Type-checking of assignment ensures the value written is consistent with the static type of the variable.

For fields, this is ensured by introducing a form of ownership types called *incomplete types* to address the *escaping-this* problem. To permit flexibility in writing constructors, X10 v1.7 permits `this` to be used in a constructor as a reference to the object currently being constructed. Unfortunately there are no restrictions on the usage of `this`. In particular, this reference can be permitted to escape: it may be stored in variables on the heap (thereby permitting concurrently executing activities to read the value of fields that may not yet have been initialized), passed as an argument to method invocations, or used as the target for a method invocation. Indeed, the method may be invoked in a super constructor, and may have been overridden at a subclass, guaranteeing that accesses to fields defined in the subclass are accesses to uninitialized variables. For instance an immutable field may be observed containing a value (the value the field was initialized with) which may be different from the value it will contain once the constructor has returned.

Incomplete types are designed with the following goals:

- Guarantee that fields are not read before they are initialized.
- Allow the creation of immutable cyclic object graphs.[2] This requires that it be possible to pass an object under construction into a constructor invocation.
- Allow appropriate user-defined methods can be called during object creation (so that the transformation between the values supplied as parameters to a constructor and the values actually placed in fields is determined by arbitrary user-defined code).
- Keep the design minimally invasive. Most programmers should not have to be concerned about this problem.
- Ensure that there is no runtime overhead.

These goals are met by introducing incomplete types through the type qualifier `proto`. Types of the form `proto T` are said to be *incomplete types*; types that do not have the qualifier are said to be *complete*. Say that an object `o` is *confined* to

[2](Mutable graphs can be created without escaping `this` by initializing the backpointer to `null` and then changing it later.

a given activity A if it can be reached only from stack frames of A or from objects which are, recursively, confined to A. Thus confined objects cannot be accessed by activities other than A.

Incomplete types ensure that objects whose construtors have not exited are confined. Further, all references to such objects on the stack are contained in variables of incomplete types. The compiler does not permit the fields of variables of incomplete types to be read. Thus incomplete types permit the construction of graphs of objects while ensuring that these objects are confined and their fields are not read during construction.

The return value of a constructor for class C that takes no incomplete arguments is (a subtype of) C, that is, a complete type. It will point only to completed objects. It can now be assigned to any (type-consistent) field of any object, that is, it is now allowed to escape.

8.7 Static initialization

The X10 runtime implements the following procedure to ensure reliable initialization of the static state of classes.

Execution commences with a single thread executing the *initialization* phase of an X10 computation at place 0. This phase must complete successfully before the body of the main method is executed.

The initialization phase must be thought of as if it is implemented in the following fashion: (The implementation may do something more efficient as long as it is faithful to this semantics.)

```
Within the scope of a new finish
for every static field f of every class C
   (with type T and initializer e):
async {
  val l = e;
  ateach (Dist.makeUnique()) {
     assign l to the static f field of
          the local C class object;
     mark the f field of the local C
          class object as initialized;
  }
```

```
}
```

During this phase, any read of a static field `C.f` (where `f` is of type `T`) is replaced by a call to the method `C.read_f():T` defined on class `C` as follows

```
def read_f():T {
   await (initialized(C.f));
   return C.f;
}
```

If all these activities terminate normally, all static field have a legal value (per their type), and the finish terminates normally. If any activity throws an exception, the finish throws an exception. Since no user code is executing which can catch exceptions thrown by the finish, the exceptions are printed on the console, and computation aborts.

If the activities deadlock, the implementation deadlocks.

In all cases, the main method is executed only once all static fields have been initialized correctly.

Since static state is immutable and is replicated to all places via the initialization phase as described above, it can be accessed from any place.

8.8 User-Defined Operators

It is often convenient to have methods named by symbols rather than words. For example, suppose that we wish to define a `Poly` class of polynomials – for the sake of illustration, single-variable polynomials with `Int` coefficients. It would be very nice to be able to manipulate these polynomials by the usual operations: + to add, `*` to multiply, `-` to subtract, and `p(x)` to compute the value of the polynomial at argument `x`. We would like to write code thus:

```
public static def main(Rail[String]) {
   val X = new Poly([0,1]);
   val t <: Poly = 7 * X + 6 * X * X * X;
   val u <: Poly = 3 + 5*X - 7*X*X;
   val v <: Poly = t * u - 1;
   for ( (i) in -3 .. 3) {
     x10.io.Console.OUT.println(
```

```
          "" + i + " X:" + X(i) + "  t:" + t(i)
          + "    u:" + u(i) + "  v:" + v(i)
          );
      }
   }
```

Writing the same code with method calls, while possible, is far less elegant:

```
   public static def uglymain() {
      val X = new UglyPoly([0,1]);
      val t <: UglyPoly = X.mult(7).plus(X.mult(X).mult(X).mult(6));
      val u <: UglyPoly = const(3).plus(X.mult(5)).minus(X.mult(X).mult(7));
      val v <: UglyPoly = t.mult(u).minus(1);
      for ( (i) in -3 .. 3) {
        x10.io.Console.OUT.println(
          "" + i + " X:" + X.apply(i) + "    t:" + t.apply(i)
           + "   u:" + u.apply(i) + "    v:" + v.apply(i)
          );
      }
   }
```

The operator-using code can be written in X10, though a few variations are necessary to handle such exotic cases as `1+X`.

8.8.1 Binary Operators

Defining the sum `P+Q` of two polynomials looks much like a method definition. It uses the `operator` keyword instead of `def`, and `this` appears in the definition in the place that a `Poly` would appear in a use of the operator. So, `operator this + (p:Poly!)` explains how to add `this` to a `Poly!` value.

```
 class Poly {
   public global val coeff : ValRail[Int];
   public def this(coeff: ValRail[Int]) { this.coeff = coeff;}
   public global def degree() = coeff.length()-1;
   public global def  a(i:Int) = (i<0 || i>this.degree()) ? 0 : coeff(i);

   public operator this + (p:Poly!) =  new Poly(
      ValRail.make[Int](
```

```
            Math.max(this.coeff.length(), p.coeff.length()),
            (i:Int) => this.a(i) + p.a(i)
         ));
      // ...
```

The sum of a polynomial and an integer, P+3, looks like an overloaded method definition.

```
public operator (n : Int) + this = new Poly([n]) + this;
```

However, we want to allow the sum of an integer and a polynomial as well: 3+P. It would be quite inconvenient to have to define this as a method on Int; changing Int is far outside of normal coding. So, we allow it as a method on Poly as well.

```
public operator this + (n : Int) = new Poly([n]) + this;
```

Furthermore, it is sometimes convenient to express a binary operation as a static method on a class. The definition for the sum of two Polys could have been written:

```
public static operator (p:Poly!) + (q:Poly!) =  new Poly(
   ValRail.make[Int](
      Math.max(q.coeff.length(), p.coeff.length()),
      (i:Int) => q.a(i) + p.a(i)
   ));
```

This requires the following syntax:

MethodHeader	::=	operator *TypeParameterList*? this *BinOp* (*FormalParameter*) *Guard*? *ReturnType*? *Throws*?
MethodHeader	::=	operator *TypeParameterList*? (*FormalParameter*) *BinOp* this *Guard*? *ReturnType*? *Throws*?
MethodHeader	::=	operator *TypeParameterList*? (*FormalParameter*) *BinOp* (*FormalParameter* *Guard*? *ReturnType*? *Throws*?

When X10 attempts to typecheck a binary operator expression like P+Q, it first typechecks P and Q. Then, it looks for operator declarations for + in the types of P and Q. If there are none, it is a static error. If there is precisely one, that one will be used. If there are several, X10 looks for a *best-matching* operation, *viz.* one which does not require the operands to be converted to another type. For example, operator this + (n:Long) and operator this + (n:Int) both apply to p+1, because 1 can be converted from an Int to a Long. However, the Int version will be chosen because it does not require a conversion. If even the best-matching operation is not uniquely determined, the compiler will report a static error.

The main difference between expressing a binary operation as an instance method (with a `this` in the definition) and a static one (no `this`) is that instance methods don't apply any conversions, while static methods attempt to convert both arguments.

8.8.2 Unary Operators

Unary operators are defined in a similar way, with `this` appearing in the `operator` definition where an actual value would occur in a unary expression. The operator to negate a polynomial is:

```
public operator - this = new Poly(
  ValRail.make[Int](coeff.length(), (i:Int) => -coeff(i))
  );
```

The syntax for unary operators is:

MethodHeader ::= `operator` *PrefixOp* `this` *Guard*$^?$ *ReturnType*$^?$ *Throws*$^?$

The rules for typechecking a unary operation are the same as for methods; the complexities of binary operations are not needed.

8.8.3 Type Conversions

Explicit type conversions, `e as T{c}`, can be defined as operators on class `T`.

```
class Poly {
  public global val coeff : ValRail[Int];
  public def this(coeff: ValRail[Int]) { this.coeff = coeff;}
  public static operator (a:Int) as Poly! = new Poly([a]);
  public static def main(Rail[String]) {
     val three : Poly! = 3 as Poly!;
  }
}
```

You may define a type conversion to a constrained type, like `Poly!` in the previous example. If you convert to a more specific constraint, X10 will use the conversion, but insert a dynamic check to make sure that you have satisfied the more specific constraint. For example:

```
class Uni(n:Int) {
  public def this(n:Int) : Uni{self.n==n} = {property(n);}
  static operator (String) as Uni{self.n != 9} = new Uni(3);
  public static def main(Rail[String]) {
    val u = "" as Uni{self.n != 9 && self.n != 3};
  }
}
```

The string `""` is converted to `Uni{self.n != 9}` via the defined conversion operator, and that value is checked against the remaining constraints `{self.n != 3}` at runtime. (In this case it will fail.)

There may be many conversions from different types to `T`, but there may be at most one conversion from any given type to `T`.

8.8.4 Implicit Type Coercions

You may also define *implicit* type coercions to `T{c}` as static operators in class `T`. The syntax for this is `static operator (x:U) : T{c} = e`. Implicit coercions are used automatically by the compiler.

For example, we can define an implicit coercion from `Int` to `Poly!`, and avoid having to define the sum of an integer and a polynomial as many special cases. In the following example, we only define + on two polynomials (using a `static` operator, so that implicit coercions will be used – they would not be for an instance method operator). The calculation `1+x` coerces 1 to a polynomial and uses polynomial addition to add it to `x`.

```
public static safe operator (c : Int) : Poly! = new Poly([c]);

public static operator (p:Poly!) + (q:Poly!) = new Poly(
    ValRail.make[Int](
      Math.max(p.coeff.length(), q.coeff.length()),
      (i:Int) => p.a(i) + q.a(i)
   ));

public static def main(Rail[String]) {
```

```
    val x = new Poly([0,1]);
    x10.io.Console.OUT.println("1+x=" + (1+x));
  }
```

8.8.5 set and apply

X10 allows types to implement the subscripting / function application operator, and indexed assignment. The Array-like classes take advantage of both of these in a(i) = a(i) + 1. Unlike unary and binary operators, subscripting and indexed assignment are done by methods, apply and set respectively.

a(b,c,d) is short for the method call a.apply(b,c,d). Since it is possible to overload methods, the application syntax can be overloaded. For example, an ordered dictionary structure could allow subscripting by numbers with def apply(i:Int), and by string-valued keys with def apply(s:String).

a(i)=b is short for the method call a.set(b,i), with one or more indices i. (This has a possibly surprising consequence for the order of evaluation: in a(i)=b, as in a.set(b,i), a is evaluated first, then b, and finally i.) Again, it is possible to overload set to provide a variety of subscripting operations.

The Oddvec class of somewhat peculiar vectors illustrates this. a() returns a string representation of the oddvec, which probably should be done by toString() instead. a(i) picks out one of the three coordinates of a, which is sensible. a(i)=b assigns to one of the coordinates. a(i,j)=b assigns different values to a(i) and a(j), purely for the sake of the example.

```
class Oddvec {
  var v : Rail[Int]! = Rail.make[Int](3, (Int)=>0);
  public def apply() = "(" + v(0) + "," + v(1) + "," + v(2) + ")";
  public def apply(i:Int) = v(i);
  public def set(newval:Int, i:Int) = {v(i) = newval;}
  public def set(newval:Int, i:Int, j:Int) = {
       v(i) = newval; v(j) = newval+1;}
  // ...
```

proto Rules

For every type T (where T is not a type variable), we introduce the type proto T.

There is no relationship between types `T` and `proto T` – neither is a subtype of the other.[3]

Incomplete types are permitted to occur only as types of method parameters or local variables or as return types for methods and constructors. They may not occur in (the source or target of) cast statements, `extends` or `implements` clauses, `catch` clauses, or as types of class fields.

Within the body of a class `C` the type of `this` in constructors, instance initializers and instance variable initializers is `proto C`.

Let `v` be a value of type `proto C`, for some class `C`.

No fields of `v` can be read. (This is the defining property of `proto` types.) However, `v`'s (accessible) instance fields can be assigned.

`v` can be assigned to an instance field `o.f` only if `f` is of some type `S` such that `T <: S` and `o` has an incomplete type.

`v` can be assigned to local variables only if they are of some type `proto S` (such that `T <: S`).

Instance methods of class `C` may be qualified with `proto` (these methods are called *incomplete methods*). The type of `this` in incomplete methods is `proto C`. Incomplete methods can be overridden only by incomplete methods. Only incomplete methods can be invoked on `v`. Incomplete methods which do not take an argument of incomplete type can be invoked on completed values.

`v` can be passed as argument into a constructor or method call, or returned from a method. The return type of a method taking an argument at an incomplete type must be `void` or incomplete. The return type of a constructor taking an argument at a `proto` type must be incomplete.

A generic type parameter `T` can be instantiated with the type `proto S` (where `S` is not a type parameter itself), provided that the body of the entity being instantiated satisfies the conditions above for `proto S`.

During code generation, the type `proto T` is treated as if it were `T`. That is, there is no run-time cost to `proto` types.

[3]Clearly, a value of type `proto T` cannot be used anywhere that a `T` is needed, since its fields cannot be read. As discussed below, an incomplete value `v` can be assigned to a field `f` of an object `o` only if `o` is incomplete. This ensures that `v` cannot escape through this assignment. A completed value `p` cannot be substituted for `o` – it may permit `v` to escape through an assignment to its field. Therefore `T` cannot be a subtype of `proto T`.

The invariants maintained by the design are as follows. Say that an object field or stack variable (local variable) contains an incomplete value if a value of type `proto T` (for some `T`) was written into it.

- If an object `o` has a field containing an incomplete value `v`, then either `v`'s constructor has exited or `o` is confined. Further, every reference to `o` on the stack is held at an incomplete type.
- If a stack variable contains an incomplete value, then the variable's type is incomplete.

Say that a constructor invocation for a class `C` on the call stack is a *root* if it takes no incomplete arguments. Such a constructor invocation will return an object of type `C` whose fields may point to an arbitrary graph of newly created objects (objects created by the activity after the constructor invocation). Since the object returned is at type `C` – and not `proto C` – It may be assigned to any field of any object on the heap of type `D` such that `C <: D`. It is no longer confined. Thus the "magic moment" when an incomplete value becomes complete is when the last constructor for any incomplete value it references (including itself) returns.

Proto Example: Circular Buffer

Example 8.8.1 This example shows how to create a fixed-size circular buffer. (Its pointer structure is immutable, though the contents of each field are mutable.) □

8.9 Type invariants

There is a general recipe for constructing a list of parameters or properties x_1: $T_1\{c_1\}$, ..., x_k: $T_k\{c_k\}$ that must satisfy a given (satisfiable) constraint `c`.

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

The first type $\mathtt{x}_1\mathtt{:}\ \mathtt{T}_1\{\mathtt{x}_2\mathtt{:}\ \mathtt{T}_2\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{x}_k\mathtt{:}\ \mathtt{T}_k\mathtt{;}\ \mathtt{c}\}$ is consistent iff $\exists \mathtt{x}_1\mathtt{:}\ \mathtt{T}_1\mathtt{,}\ \mathtt{x}_2\mathtt{:}\ \mathtt{T}_2\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_k\mathtt{:}\ \mathtt{T}_k\mathtt{.}\ \mathtt{c}$ is consistent. The second is consistent iff

$$\forall \mathtt{x}_1\mathtt{:}\ \mathtt{T}_1\{\mathtt{x}_2\mathtt{:}\ \mathtt{T}_2\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{x}_k\mathtt{:}\ \mathtt{T}_k\mathtt{;}\ \mathtt{c}\}$$
$$\exists \mathtt{x}_2\mathtt{:}\ \mathtt{T}_2\mathtt{.}\ \exists \mathtt{x}_3\mathtt{:}\ \mathtt{T}_3\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_k\mathtt{:}\ \mathtt{T}_k\mathtt{.}\ \mathtt{c}$$

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

Thus logically every satisfiable constraint c on a list of parameters $\mathtt{x}_1, \ldots, \mathtt{x}_k$ can be expressed using the dependent types of $\mathtt{x}_i$, 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(i: Int, j: Int) { ... }
class Line(start: Point, end: Point){end != start} { ... }
class Triangle (a: Line, b: Line, c: Line)
   {a.end == b.start,  b.end == c.start,
    c.end == a.start} { ... }
```

Consider the definition of the class `Line`. This may be thought of as saying: the class `Line` has two fields, `start: Point` and `end: Point`. 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 *Guard*.

NormalClassDeclaration	::=	*ClassModifiers*$^?$ `class` *Identifier* *TypeParameterList*$^?$ *PropertyList*$^?$ *Guard*$^?$ *Extends*$^?$ *Interfaces*$^?$ *ClassBody*
NormalInterfaceDeclaration	::=	*InterfaceModifiers*$^?$ `interface` *Identifier* *TypeParameterList*$^?$ *PropertyList*$^?$ *Guard*$^?$ *ExtendsInterfaces*$^?$ *InterfaceBody*

All the properties in the list, together with inherited properties, may appear in the *Guard*. A guard c with property list $\mathtt{x}_1\mathtt{:}\ \mathtt{T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_n\mathtt{:}\ \mathtt{T}_n$ for a class C is said to be consistent if each of the $\mathtt{T}_i$ are consistent and the constraint

$$\exists \mathtt{x}_1\mathtt{:}\ \mathtt{T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_n\mathtt{:}\ \mathtt{T}_n\mathtt{,}\ \mathtt{self:}\ \mathtt{C.}\ \mathtt{c}$$

is valid (always true).

The guard is an invariant on all instances of the class or interface.

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

The type invariant associated with any interface `I` that extends interfaces $\mathtt{I}_1$, ..., $\mathtt{I}_k$ and defines properties $\mathtt{x}_1$: $\mathtt{P}_1$, ..., $\mathtt{x}_n$: $\mathtt{P}_n$ and specifies a guard `c` is given by:

$$inv(\mathtt{I}_1),\ \ldots,\ inv(\mathtt{I}_k),$$
$$\mathtt{self.x}_1\mathtt{:}\ \mathtt{P}_1,\ \ldots,\ \mathtt{self.x}_n\mathtt{:}\ \mathtt{P}_n,\ \mathtt{c}$$

Similarly the type invariant associated with any class `C` that implements interfaces $\mathtt{I}_1$, ..., $\mathtt{I}_k$, extends class `D` and defines properties $\mathtt{x}_1$: $\mathtt{P}_1$, ..., $\mathtt{x}_n$: $\mathtt{P}_n$ and specifies a guard `c` is given by:

$$inv(\mathtt{D}),\ inv(\mathtt{I}_1),\ \ldots,\ inv(\mathtt{I}_k),$$
$$\mathtt{self.x}_1\mathtt{:}\ \mathtt{P}_1,\ \ldots,\ \mathtt{self.x}_n\mathtt{:}\ \mathtt{P}_n,\ \mathtt{c}$$

It is required that the type invariant associated with a class entail the type invariants of each interface that it implements.

It is guaranteed 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 $inv(\mathtt{T}[\mathtt{o}/\mathtt{this}]) \wedge \mathtt{c}[\mathtt{o}/\mathtt{self}]$.

8.9.1 Invariants for `implements` and `extends` clauses

Consider a class definition

$$\mathit{ClassModifiers}^?$$
$$\mathtt{class\ C(x}_1\mathtt{:\ P}_1\mathtt{,}\ \ldots,\ \mathtt{x}_n\mathtt{:\ P}_n\mathtt{)\ extends\ D\{d\}}$$
$$\mathtt{implements\ I}_1\mathtt{\{c}_1\mathtt{\},}\ \ldots,\ \mathtt{I}_k\mathtt{\{c}_k\mathtt{\}}$$
$$\mathit{ClassBody}$$

Each of the following static semantics rules must be satisfied:

STATIC SEMANTICS RULE (Int-implements): The type invariant inv(`C`) of `C` must entail $\mathtt{c}_i$`[this/self]` for each i in $\{1, \ldots, k\}$

STATIC SEMANTICS RULE (Super-extends): The return type `c` of each constructor in *ClassBody* must entail `d`.

8.9.2 Invariants and constructor definitions

A constructor for a class `C` is guaranteed to return an object of the class on successful termination. This object must satisfy inv(`C`), the class invariant associated with `C` (§8.9). 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 constrained type (the "return type" of the constructor):

ConstructorDeclarator	::=	`def` `this` *TypeParameterList*$^?$ `(` *FormalParameterList*$^?$ `)` *ReturnType*$^?$ *Guard*$^?$ *Throws*$^?$
ReturnType	::=	`:` *Type*
Guard	::=	"{" *DepExpression* "}"
Throws	::=	`throws` *ExceptionType* (`,` *ExceptionType*)*
ExceptionType	::=	*ClassBaseType Annotation**

The parameter list for the constructor may specify a *guard* that is to be satisfied by the parameters to the list.

Example 8.9.1 Here is another example, constructed as a simplified version of `x10.lang.Region`. The `mockUnion` method has the type that a true `union` method would have.

```
class MyRegion(rank:Int) {
  static type MyRegion(n:Int)=MyRegion{self.rank==n};
  def this(r:Int):MyRegion(r) {
    property(r);
  }
  def this(diag:ValRail[Int]):MyRegion(diag.length){
    property(diag.length);
  }
  def mockUnion(r:MyRegion(rank)):MyRegion(rank) = this;
  def example() {
    val R1 : MyRegion(3)! = new MyRegion([4,4,4]);
    val R2 : MyRegion(3)! = new MyRegion([5,4,1]);
    val R3 = R1.mockUnion(R2); // inferred type MyRegion(3)
  }
}
```

The first constructor returns the empty region of rank `r`. The second constructor takes a `ValRail[Int]` of arbitrary length `n` and returns a `MyRegion(n)` (intended to represent the set of points in the rectangular parallelopiped between the origin and the `diag`.)

The code in `example` typechecks, and `R3`'s type is inferred as `MyRegion(3)`.

□

STATIC SEMANTICS RULE (Super-invoke): Let `C` be a class with properties $\mathtt{p}_1\mathtt{:}\ \mathtt{P}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{p}_n\mathtt{:}\ \mathtt{P}_n$, invariant `c` extending the constrained type `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 $\mathtt{super(e}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{e}_k\mathtt{)}$ and the static type of each expression $\mathtt{e}_i$ is $\mathtt{S}_i$, and the invocation is statically resolved to a constructor $\mathtt{def\ this(x}_1\mathtt{:}\ \mathtt{T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_k\mathtt{:}\ \mathtt{T}_k\mathtt{)\{c\}:}\ \mathtt{D\{d}_1\mathtt{\}}$ then it must be the case that

$$\mathtt{x}_1\mathtt{:}\ \mathtt{S}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_i\mathtt{:}\ \mathtt{S}_i \vdash \mathtt{x}_i\mathtt{:}\ \mathtt{T}_i \quad \mathtt{(for}\ i \in \{1,\ldots,k\}\mathtt{)}$$
$$\mathtt{x}_1\mathtt{:}\ \mathtt{S}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_k\mathtt{:}\ \mathtt{S}_k \vdash \mathtt{c}$$
$$\mathtt{d}_1\mathtt{[a/self],}\ \mathtt{x}_1\mathtt{:}\ \mathtt{S}_1\mathtt{,}\ \mathtt{...,}\ \mathtt{x}_k\mathtt{:}\ \mathtt{S}_k \vdash \mathtt{d[a/self]}$$

where `a` is a constant that does not appear in $\mathtt{x}_1\mathtt{:}\ \mathtt{S}_1 \wedge \mathtt{...} \wedge \mathtt{x}_k\mathtt{:}\ \mathtt{S}_k$.

STATIC SEMANTICS RULE (Constructor return): The compiler checks that every constructor for `C` ensures that the properties $\mathtt{p}_1\mathtt{,...,}\ \mathtt{p}_n$ 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 $\mathtt{T}_i$ of the expressions $\mathtt{e}_i$ assigned to $\mathtt{p}_i$ are such that the following is true:

$$\mathtt{p}_1\mathtt{:}\ \mathtt{T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{p}_n\mathtt{:}\ \mathtt{T}_n \vdash \mathtt{t(C)} \wedge \mathtt{c'}$$

(Note that for the assignment of $\mathtt{e}_i$ to $\mathtt{p}_i$ to be type-correct it must be the case that $\mathtt{p}_i\mathtt{:}\ \mathtt{T}_i \wedge \mathtt{p}_i\mathtt{:}\ \mathtt{P}_i$.)

STATIC SEMANTICS RULE (Constructor invocation): The compiler must check that every invocation $\mathtt{C(e}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{e}_n\mathtt{)}$ to a constructor is type correct: each argument $\mathtt{e}_i$ must have a static type that is a subtype of the declared type $\mathtt{T}_i$ for the ith argument of the constructor, and the conjunction of static types of the argument must entail the *Guard* in the parameter list of the constructor.

9 Structs

X10 objects are a powerful general-purpose programming tool. However, the power must be paid for in space and time. In space, a typical object implementation requires some extra memory for run-time class information, as well as a pointer for each reference to the object. In time, a typical object requires an extra indirection to read or write data, and some computation to figure out which method body to call.

For high-performance computing, this overhead may not be acceptable for all objects. X10 provides structs, which are stripped-down objects. They are less powerful than objects; in particular they lack inheritance and mutable fields. Without inheritance, method calls do not need to do any lookup; they can be implemented directly. Accordingly, structs can be implemented and used more cheaply than objects, potentially avoiding the space and time overhead. (Currently, the C++ back end avoids the overhead, but the Java back end implements structs as Java objects and does not avoid it.)

Structs and classes are interoperable. Both can implement interfaces (in particular, like all X10 values they implement `Any`), and subprocedures whose arguments are defined by interfaces can take both structs and classes. (Some caution is necessary here: referring to a struct through an interface requires overhead similar to that required for an object.)

They are also interconvertable, within the constraints of structs. If you start off defining a struct and decide you need a class instead, the code change required is simply changing the keyword `struct` to `class`. If you have a class that does not use inheritance or mutable fields, it can be converted to a struct by changing its keyword.

9.1 Struct declaration

A struct declaration has the structure:

```
StructModifiers?
struct C[X₁, ..., Xₙ](p₁:T₁, ..., pₙ:Tₙ){c}
   implements I₁, ..., Iₖ {
StructBody
}
```

Each field and method in a struct is implicitly `global`, and each field is `val`.

A struct `S` cannot contain a field of type `S`, or a field of struct type `T` which, recursively, contains a field of type `S`. This restriction is necessary to permit `S` to be implemented as a contiguous block of memory of size equal to the sum of the sizes of its fields.

Values of a struct `C` type can be created by invoking a constructor defined in `C`, but without prefixing it with `new`:

```
struct Polar(r:Double, theta:Double){
  def this(r:Double, theta:Double) {property(r,theta);}
  static val Origin = Polar(0,0);
  static val x0y1 = Polar(1, 3.14159/2);
}
```

Structs support the same notions of generics, properties, and constrained types that classes do. For example, the `Pair` type below provides pairs of values; the `diag()` method applies only when the two elements of the pair are equal, and returns that common value:

```
struct Pair[T,U](t:T, u:U) {
  def this(t:T, u:U) { property(t,u); }
  def diag(){T==U && t==u} = t;
}
```

9.2 Boxing of structs

If a struct `S` implements an interface `I` (*e.g.*, `Any`), a value `v` of type `S` can be assigned to a variable of type `I`. The implementation creates an object `o` that is an instance of an anonymous class implementing I and containing v. The result of invoking a method of I on o is the same as invoking it on v. This operation is termed *auto-boxing*. It allows full interoperability of structs and objects—at the cost of losing the extra efficiency of the structs when they are boxed.

In a generic class or struct obtained by instantiating a type parameter T with a struct S, variables declared at type T in the body of the class are not boxed. They are implemented as if they were declared at type S.

9.3 Optional Implementation of Any methods

Unlike objects, structs do not have global identity. Instead, two structs are equal (==) if and only if their corresponding fields are equal (==).

All structs implement x10.lang.Any. All structs have the following methods implicitly defined on them:

```
property def home()=here;
property def at(Place)=true;
property def at(Object)=true;
```

It is an error for a programmer to attempt to define them. Note that the home of a struct evaluated in place P is equal to P—a struct, unlike an object, can have different values of home.

Structs are required to implement the following methods from Any. Programmers need not provide them; X10 will produce them automatically if the program does not include them.

```
public global safe def equals(Any):Boolean;
public global safe def hashCode():Int;
public global safe def typeName():String;
public global safe def toString():String;
```

A programmer who provides an explicit implementation of equals(Any) for a struct S should also consider supplying a definition for equals(S):Boolean. This will often yield better performance since the cost of an upcast to Any and then a downcast to S can be avoided.

9.4 Primitive Types

Certain types that might be built in to other languages are in fact implemented as structs in package `x10.lang` in X10. Their methods and operations are often provided with `@Native` (§18) rather than X10 code, however. These types are:

```
boolean, char, byte, short, int, long
float, double, ubyte, ushort, uint, ulong
```

9.5 Generic programming with structs

An unconstrained type variable `X` can be instantiated with `Object` or its subclasses or structs or functions.

Within a generic struct, all the operations of `Any` are available on a variable of type `X`. Additionally, variables of type `X` may be used with `==`, `!=`, in `instanceof`, and casts.

The programmer must be aware of the different interpretations of equality for structs and classes and ensure that the code is correctly written for both cases. If necessary the programmer can write code that distinguishes between the two cases (a type parameter `X` is instantiated to a struct or not) as follows:

```
val x:X = ...;
if (x instanceof Object) { // x is a real object
   val x2 = x as Object; // this cast will always succeed.
   ...
} else { // x is a struct
   ...
}
```

9.6 Example structs

`x10.lang.Complex` provides a detailed example of a practical struct, suitable for use in a library. For a shorter example, we define the `Pair` struct—available in `x10.util.Pair`. A `Pair` packages two values of possibly unrelated type together in a single value, *e.g.*, to return two values from a function.

```
struct Pair[T,U] {
    public val first:T;
    public val second:U;
    public def this(first:T, second:U):Pair[T,U] {
        this.first = first;
        this.second = second;
    }
    public global safe def toString():String {
        return "(" + first + ", " + second + ")";
    }
}
```

10 Functions

10.1 Overview

Functions, the last of the three kinds of values in X10, encapsulate pieces of code which can be applied to a vector of arguments to produce a value. Functions, when applied, can do nearly anything that any other code could do: fail to terminate, throw an exception, modify variables, spawn activities, execute in several places, and so on. X10 functions are not mathematical functions: the `f(1)` may return `true` on one call and `false` on an immediately following call.

It is a limitation of X10 v2.0 that functions do not support type arguments. This limitation may be removed in future versions of the language.

A *function literal* `(x1:T1,..,xn:Tn){c}:T=>e` creates a function of type `(x1:T1,...,xn:Tn){c}=>T` (§4.6). For example, `(x:Int) => x*x` is a function literal describing the squaring function on integers. `null` is also a function value.

Function application is written `f(a,b,c)`, following common mathematical usage. Function invocation may throw unchecked exceptions.

The function body may be a block. To compute integer squares by repeated addition (inefficiently), one may write:

```
val sq: (Int) => Int
      = (n:Int) => {
           var s : Int = 0;
           val abs_n = n < 0 ? -n : n;
           for ((i) in 1..abs_n) s += abs_n;
           s
        };
```

A function literal evaluates to a function entity ϕ. When ϕ is applied to a suitable list of actual parameters `a1-an`, it evaluates `e` with the formal parameters bound to the actual parameters. So, the following are equivalent, where `e` is an expression involving `x1` and `x2`[1]

```
{
  val f = (x1:T1,x2:T2){true}:T => e;
  val a1 : T1 = arg1();
  val a2 : T2 = arg2();
  result = f(a1,a2);
}
```

and

```
{
  val a1 : T1 = arg1();
  val a2 : T2 = arg2();
  {
    val x1 : T1 = a1;
    val x2 : T2 = a2;
    result = e;
  }
}
```

This doesn't quite work if the body is a statement rather than an expression. A few language features are forbidden (`break` or `continue` of a loop that surrounds the function literal) or mean something different (`return` inside a function returns from the function).

The *method selector expression* `e.m.(x1:T1,...,xn:Tn)` (§10.3) permits the specification of the function underlying the method `m`, which takes arguments of type `(x1:T1,..., xn:Tn)`. Within this function, `this` is bound to the result of evaluating `e`.

Function types may be used in `implements` clauses of class definitions. Instances of such classes may be used as functions of the given type. Indeed, an object may behave like any (fixed) number of functions, since the class it is an instance of may implement any (fixed) number of function types.

[1] Strictly, there are a few other requirements; *e.g.*, `result` must be a `var` of type `T` defined outside the outer block, the variables `a1` and `a2` had better not appear in `e`, and everything in sight had better typecheck properly.

10.2 Function Literals

X10 provides first-class, typed functions, including *closures*, *operator functions*, and *method selectors*.

ClosureExpression	::=	(*Formals*?)
		Guard? *ReturnType*? *Throws*? => *ClosureBody*
ClosureBody	::=	*Expression*
	|	{ *Statement** }
	|	{ *Statement** *Expression* }

Functions have zero or more formal parameters, an optional return type and optional set of exceptions throws by the body. The body has the same syntax as a method body; it may be either an expression, a block of statements, or a block terminated by an expression to return. In particular, a value may be returned from the body of the function using a return statement (§12.15).

The type of a function is a function type (§4.6). In some cases the return type `T` is also optional and defaults to the type of the body. If a formal `xi` does not occur in any `Tj`, `c`, `T` or `e`, the declaration `xi:Ti` may be replaced by just `Ti`: `(Int)=>7` is the integer function returning 7 for all inputs.

As with methods, a function may declare a guard to constrain the actual parameters with which it may be invoked. The guard may refer to the type parameters, formal parameters, and any `val`s in scope at the function expression.

The body of the function is evaluated when the function is invoked by a call expression (§11.6), not at the function's place in the program text.

As with methods, a function with return type `Void` cannot have a terminating expression. If the return type is omitted, it is inferred, as described in §4.11. It is a static error if the return type cannot be inferred. *E.g.*, `(Int)=>null` is not well-defined; X10 does not know which type of `null` is intended. But `(Int):Rail[Double] => null` is legal.

Example 10.2.1 The following method takes a function parameter and uses it to test each element of the list, returning the first matching element. It returns `otherwise` if no element matches.

```
def find[T](f: (T) => Boolean, xs: List[T], absent:T): T = {
```

```
for (x: T in xs)
  if (f(x)) return x;
absent
}
```

The method may be invoked thus:

```
xs: List[Int] = new ArrayList[Int]();
x: Int = find((x: Int) => x>0, xs, 0);
```

□

As with a normal method, the function may have a `throws` clause. It is a static error if the body of the function throws a checked exception that is not declared in the function's `throws` clause.

10.2.1 Outer variable access

In a function $(\mathtt{x}_1\mathtt{:}\ \mathtt{T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_n\mathtt{:}\ \mathtt{T}_n)\{\mathtt{c}\}$ `=> { s }` the types $\mathtt{T}_i$, the guard `c` and the body `s` may access many, though not all, sorts of variables from outer scopes. Specifically, they can access:

- All fields of the enclosing object and class;
- All type parameters;
- All `val` variables;
- `var` variables with the `shared` annotation.

Limitation `shared` *is not currently supported.*

The function body may refer to instances of enclosing classes using the syntax `C.this`, where `C` is the name of the enclosing class. `this` refers to the instance of the immediately enclosing class, as usual.

For example, the following is legal. However, it would not be legal to add `e` or `h` to the sum; they are non-`shared` `var`s from the surrounding scope.

```
class Lambda {
   var a : Int = 0;
   val b = 0;
   def m(var c : Int, shared var d : Int,  val e : Int) {
      var f : Int = 0;
      shared var g : Int = 0;
      val h : Int = 0;
      val closure = (var i: Int, val j: Int) => {
          return a + b + d + e + g + h + i + j + this.a + Lambda.this.a;
      };
      return closure;
   }
}
```

Rationale: Non-`shared` `var`s like `e` and `h` are excluded in X10, as in many other languages, for practical implementation reasons. They are allocated on the stack, which is desirable for efficiency. However, the closure may exist for long after the stack frame containing `e` and `h` has been freed, so those storage locations are no longer valid for those variables. `shared vars` are heap-allocated, which is less efficient but allows them to exist after `m` returns.

`shared` does not guarantee **atomic** access to the shared variable. As with any code that might mutate shared data concurrently, be sure to protect references to mutable shared state with `atomic`. For example, the following code returns a pair of closures which operate on the same shared variable `a`, which are concurrency-safe—even if invoked many times simultaneously. Without `atomic`, it would no longer be concurrency-safe.

```
 def counters() {
     shared var a : Int = 0;
      return [
         () => {atomic a ++;},
         () => {atomic return a;}
         ];
  }
```

10.3 Method selectors

A method selector expression allows a method to be used as a first-class function, without writing a function expression for it. For example, consider a class `Span` defining ranges of integers.

```
class Span(low:Int, high:Int) {
  def this(low:Int, high:Int) {property(low,high);}
  def between(n:Int) = low <= n && n <= high;
  def example() {
    val digit = new Span(0,9);
    val isDigit : (Int) => Boolean = digit.between.(Int);
    if (isDigit(8)) x10.io.Console.OUT.println("8 is!");
  }
}
```

In `example()`, `digit.between.(Int)` is a unary function testing whether its argument is between zero and nine. It could also be written `(n:Int) => digit.between(n)`.

This is formalized thus:

MethodSelector	::=	*Primary* . *MethodName* . *TypeParameters*? (*Formals*?)
	|	*TypeName* . *MethodName* . *TypeParameters*? (*Formals*?)

The *method selector expression* `e.m.(T1,...,Tn)` is type correct only if the static type of `e` is a class or struct or interface `V` with a method `m(x1:T1,...xn:Tn){c}:T` defined on it (for some `x1,...,xn,c,T`). At runtime the evaluation of this expression evaluates `e` to a value `v` and creates a function `f` which, when applied to an argument list `(a1,...,an)` (of the right type) yields the value obtained by evaluating `v.m(a1,...,an)`.

Thus, the method selector

$\mathtt{e.m.[X}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{X}_m\mathtt{](T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{T}_n\mathtt{)}$

behaves as if it were the function

$\mathtt{((v:V)=>}$
$\quad\mathtt{[X}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{X}_m\mathtt{](x}_1\mathtt{:\ T}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_n\mathtt{:\ T}_n\mathtt{)\{c\}}$
$\quad\mathtt{=>\ v.m[X}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{X}_m\mathtt{](x}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{x}_n\mathtt{))}$
$\mathtt{(e)}$

Limitation *X10 functions, including method selectors, do not currently accept generic arguments.*

Because of overloading, a method name is not sufficient to uniquely identify a function for a given class (in Java-like languages). One needs the argument type information as well. The selector syntax (dot) is used to distinguish `e.m()` (a method invocation on `e` of method named `m` with no arguments) from `e.m.()` (the function bound to the method).

A static method provides a binding from a name to a function that is independent of any instance of a class; rather it is associated with the class itself. The static function selector `T.m.(`T_1`, ..., `T_n`)` denotes the function bound to the static method named `m`, with argument types `(`T_1`, ..., `T_n`)` for the type `T`. The return type of the function is specified by the declaration of `T.m`.

Users of a function type do not care whether a function was defined directly (using the function syntax), or obtained via (static or instance) function selectors.

10.4 Operator functions

Every operator (e.g., +, -, *, /, ...) has a family of functions, one for each type on which the operator is defined. The function can be selected using the "." syntax:

OperatorFunction	::=	*TypeName* `.` *Operator* `(` *Formals*$^?$ `)`
	|	*TypeName* `.` *Operator*

If an operator has more than one arity (*e.g.*, unary and binary `-`), the unary version may be selected by giving the formal parameter types. The binary version is selected by default, or the types may be specified for clarity. For example, the following equivalences hold:

```
String.+              ≡ (x: String, y: String): String => x + y
Long.-                ≡ (x: Long, y: Long): Long => x - y
Float.-(Float,Float)  ≡ (x: Float, y: Float): Float => x - y
Int.-(Int)            ≡ (x: Int): Int => -x
Boolean.&             ≡ (x: Boolean, y: Boolean): Boolean => x & y
Boolean.!             ≡ (x: Boolean): Boolean => !x
Int.<(Int,Int)        ≡ (x: Int, y: Int): Boolean => x < y
Dist.|(Place)         ≡ (d: Dist, p: Place): Dist => d | p
```

Unary and binary promotion (§11.9) is not performed when invoking these operations; instead, the operands are coerced individually via implicit coercions (§11.27), as appropriate.

Planned 10.4.1 The following is not implemented in version 2.0.3:

Additionally, for every expression `e` of a type `T` at which a binary operator `OP` is defined, the expression `e.OP` or `e.OP(T)` represents the function defined by:

```
(x: T): T => { e OP x }
```

Primary	::=	*Expr* `.` *Operator* `(` *Formals*$^?$ `)`
	|	*Expr* `.` *Operator*

For example, one may write an expression that adds one to each member of a list `xs` by:

```
xs.map(1.+);
```

□

10.5 Functions as objects of type `Any`

Two functions `f` and `g` are equal ("==") if both are instances of classes and the same object, or if both were obtained by the same evaluation of a function literal.[2] Further, it is guaranteed that if two functions are equal then they refer to the same locations in the environment and represent the same code, so their executions in an identical situation are indistinguishable. (Specifically, if `f == g`, then `f(1)` can be substituted for `g(1)` and the result will be identical. However, there is no guarantee that `f(1)==g(1)` will evaluate to true. Indeed, there is no guarantee that `f(1)==f(1)` will evaluate to true either, as `f` might be a function which returns n on its n^{th} invocation. However, `f(1)==f(1)` and `f(1)==g(1)` are interchangeable.)

Every function type implements all the methods of `Any`. `bf.equals(g)` is equivalent to `f==g`. `f.hashCode()`, `f.toString()`, and `f.typeName()` are implementation-dependent, but respect `equals` and the basic contracts of `Any`. `f.home()` returns `here` and `f.at(x)` always returns true, as for structs.

[2] A literal may occur in program text within a loop, and hence may be evaluated multiple times.

11 Expressions

X10 has a rich expression language. Evaluating an expression produces a value, or, in a few cases, no value. Expression evaluation may have side effects, such as change of the value of a `var` variable or a data structure, allocation of new values, or throwing an exception.

Evaluation is performed left to right, wherever possible. For example, in `f()(a(),b())`, `f()` is evaluated, then `a()`, then `b()`, andd then the application.

11.1 Literals

Literals denote fixed values of built-in types. The syntax for literals is given in §3.

The type that X10 gives a literal often includes its value. *E.g.*, `1` is of type `Int{self==1}`, and `true` is of type `Boolean{self==true}`.

11.2 `this`

ThisExpression	::=	`this`
	|	*ClassName* `.` `this`

The expression `this` is a local `val` containing a reference to an instance of the lexically enclosing class. It may be used only within the body of an instance method, a constructor, or in the initializer of a instance field – that is, the places where there is an instance of the class under consideration.

Within an inner class, `this` may be qualified with the name of a lexically enclosing class. In this case, it represents an instance of that enclosing class. For example, `Outer` is a class containing `Inner`. Each instance of `Inner` has a reference `outer` to the `Outer` involved in its creation, which is acquired by use of `Outer.this`.

```
class Outer {
  val inner : Inner! = new Inner();
  class Inner {
    val outer : Outer = Outer.this;
  }
  def alwaysTrue() = (this == inner.outer);
}
```

The type of a `this` expression is the innermost enclosing class, or the qualifying class, constrained by the class invariant and the method guard, if any.

The `this` expression may also be used within constraints in a class or interface header (the class invariant and `extends` and `implements` clauses). Here, the type of `this` is restricted so that only properties declared in the class header itself, and specifically not any members declared in the class body or in supertypes, are accessible through `this`.

11.3 Local variables

LocalExpression ::= *Identifier*

A local variable expression consists simply of the name of the local variable, field of the current object, formal parameter in scope, etc. It evaluates to the value of the local variable. `n` in the second line below is a local variable expression.

```
val n = 22;
val m = n + 56;
```

11.4 Field access

FieldExpression	::=	*Expression* . *Identifier*
	|	`super` . *Identifier*
	|	*ClassName* . *Identifier*
	|	*ClassName* . `super` . *Identifier*

A field of an object instance may be accessed with a field access expression.

The type of the access is the declared type of the field with the actual target substituted for `this` in the type.

The field accessed is selected from the fields and value properties of the static type of the target and its superclasses.

If the field target is given by the keyword `super`, the target's type is the superclass of the enclosing class. This form is used to access fields of the parent class shadowed by same-named fields of the current class.

If the field target is `Cls.super`, then the target's type is `Cls`, which must be an ancestor class of the enclosing class. This (admittedly obscure) form is used to access fields of an ancestor class which are shadowed by same-named fields of some more recent ancestor. The following example illustrates all four cases:

```
class Uncle {
  public static global val f = 1;
}
class Parent {
  public global val f = 2;
}
class Ego extends Parent {
  public global val f = 3;
  class Child extends Ego {
     public global val f = 4;
     def expDotId() = this.f; // 4
     def superDotId() = super.f; // 3
     def classNameDotId() = Uncle.f; // 1;
     def cnDotSuperDotId() = Ego.super.f; // 2
  }
}
```

If the field target is `null`, a `NullPointerException` is thrown.

If the field target is a class name, a static field is selected.

It is illegal to access a field that is not visible from the current context. It is illegal to access a non-static field through a static field access expression.

11.5 Function Literals

Function literals are described in §10.

11.6 Calls

MethodCall	::=	*TypeName* `.` *Identifier TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
	|	`super` `.` *Identifier TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
	|	*ClassName* `.` `super` `.` *Identifier TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
Call	::=	*Primary TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)`
TypeArguments	::=	`[` *Type (* `,` *Type)** `]`

A *MethodCall* may be to either a `static` or an instance method. A *Call* may invoke either a method or a closure.

The syntax is ambiguous; the target must be type-checked to determine if it is the name of a method or if it refers to a field containing a closure. It is a static error if a call may resolve to both a closure call or to a method call.

```
class Callsome {
  static val closure = () => 1;
  static def method () = 2;
  static val closureEvaluated = Callsome.closure();
  static val methodEvaluated = Callsome.method();
}
```

However, adding a static method called `closure` makes `Callsome.closure()` ambiguous: it could be a call to the closure, or to the static method:

```
  static def closure () = 3;
  // ERROR: static errory = Callsome.closure();
```

A closure call `e(...)` is shorthand for a method call `e.apply(...)`.

Method selection rules are similar to that of Java. For a call with no explicit type arguments, a method with no parameters is considered more specific than a method with one or more type parameters that would have to be inferred.

It is a static error if a method's *Guard* is not satisfied by the caller.

11.7 Assignment

Expression	::=	*Assignment*
Assignment	::=	*SimpleAssignment*
	|	*OpAssignment*
SimpleAssignment	::=	*LeftHandSide* = *Expression*
OpAssignment	::=	*LeftHandSide* += *Expression*
	::=	*LeftHandSide* -= *Expression*
	::=	*LeftHandSide* *= *Expression*
	::=	*LeftHandSide* /= *Expression*
	::=	*LeftHandSide* %= *Expression*
	::=	*LeftHandSide* &= *Expression*
	::=	*LeftHandSide* |= *Expression*
	::=	*LeftHandSide* ^= *Expression*
	::=	*LeftHandSide* <<= *Expression*
	::=	*LeftHandSide* >>= *Expression*
	::=	*LeftHandSide* >>>= *Expression*
LeftHandSide	::=	*Identifier*
	|	*Primary* . *Identifier*
	|	*Primary* (*Expression*)

The assignment expression `x = e` assigns a value given by expression `e` to a variable `x`. Most often, `x` is a mutable (`var` variable). The same syntax is used for delayed initialization of a `val`, but `val`s can only be initialized once.

```
var x : Int;
val y : Int;
x = 1;
y = 2; // Correct; initializes y
x = 3;
```

```
// Incorrect: y = 4;
```

There are three syntactic forms of assignment:

1. `x = e;`, assigning to a local variable, formal parameter, field of `this`, etc.
2. `x.f = e;`, assigning to a field of an object.
3. `a(`$\mathtt{i}_1$`,...,`$\mathtt{i}_n$`) = v;`, where $n \geq 0$, assigning to an element of an array or some other such structure. This is syntactic sugar for a method call: `a.set(v,`$\mathtt{i}_1$`,...,`$\mathtt{i}_n$`)`. Naturally, it is a static error if no suitable `set` method exists for `a`.

For a binary operator $\diamond$, the $\diamond$-assignment expression `x` $\diamond$`= e` combines the current value of `x` with the value of `e` by $\diamond$, and stores the result back into `x`. `i += 2`, for example, adds 2 to `i`. For variables and fields, `x` $\diamond$`= e` behaves just like `x = x` $\diamond$ `e`.

The subscripting forms of `a(i)` $\diamond$`= b` are slightly subtle. Subexpressions of `a` and `i` are only evaluated once. However, `a(i)` and `a(i)=c` are each executed once—in particular, there is one call to `a.apply(i)` and one to `a.set(i,c)`, the desugared forms of `a(i)` and `a(i)=c`. If subscripting is implemented strangely for the class of `a`, the behavior is *not* necessarily updating a single storage location. Specifically, `A()(I()) += B()` is tantamount to:

```
{
  val aa = A();  // Evaluate A() once
  val ii = I();  // Evaluate I() once
  val bb = B();  // Evaluate B() once
  val tmp = aa(ii) + bb; // read aa(ii)
  aa(ii) = tmp;  // write sum back to aa(ii)
}
```

Limitation += *does not currently meet this specification.*

11.8 Increment and decrement

The operators ++ and `--` increment and decrement a variable, respectively. `x++` and `++x` both increment `x`, just as the statement `x += 1` would, and similarly for `--`.

The difference between the two is the return value. `++x` returns the *new* value of `x`, after incrementing. `x++` returns the *old* value of `, before incrementing.`

Limitation *This currently only works for numeric types.*

11.9 Numeric Operations

Numeric types (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, and unsigned variants of fixed-point types) are normal X10 structs, though most of their methods are implemented via native code. They obey the same general rules as other X10 structs. For example, numeric operations are defined by `operator` definitions, the same way you could for any struct.

11.9.1 Conversions and coercions

Specifically, each numeric type can be converted or coerced into each other numeric type, perhaps with loss of accuracy.

```
val f : (Int)=>Boolean = (Int) => true;
val n : Byte = 123 as Byte; // explicit
val ok = f(n); // implicit
```

11.9.2 Unary plus and unary minus

The unary + operation on numbers is an identity function. The unary `-` operation on numbers is a negation function. On unsigned numbers, these are two's-complement. For example, `-(0x0F as UByte)` is `(0xF1 as UByte)`.

11.10 Bitwise complement

The unary ˜ operator, only defined on integral types, complements each bit in its operand.

11.11 Binary arithmetic operations

The binary arithmetic operators perform the familiar binary arithmetic operations: `+` adds, `-` subtracts, `*` multiplies, `/` divides, and `%` computes remainder.

On integers, the operands are coerced to the longer of their two types, and then operated upon. Floating point operations are determined by the IEEE 754 standard. The integer `/` and `%` throw a `DivideByZeroException` if the right operand is zero.

11.12 Binary shift operations

The operands of the binary shift operations must be of integral type. The type of the result is the type of the left operand.

If the promoted type of the left operand is `Int`, the right operand is masked with `0x1f` using the bitwise AND (&) operator, giving a number $\leq$ the number of bits in an `Int`. If the promoted type of the left operand is `Long`, the right operand is masked with `0x3f` using the bitwise AND (&) operator, giving a number $\leq$ the number of bits in a `Long`.

The << operator left-shifts the left operand by the number of bits given by the right operand. The >> operator right-shifts the left operand by the number of bits given by the right operand. The result is sign extended; that is, if the right operand is k, the most significant k bits of the result are set to the most significant bit of the operand.

The >>> operator right-shifts the left operand by the number of bits given by the right operand. The result is not sign extended; that is, if the right operand is k, the most significant k bits of the result are set to `0`. This operation is deprecated, and may be removed in a later version of the language.

11.13 Binary bitwise operations

The binary bitwise operations operate on integral types, which are promoted to the longer of the two types. The `&` operator performs the bitwise AND of the promoted operands. The `|` operator performs the bitwise inclusive OR of the promoted operands. The ˆ operator performs the bitwise exclusive OR of the promoted operands.

11.14 String concatenation

The + operator is used for string concatenation as well as addition. If either operand is of static type `x10.lang.String`, the other operand is converted to a `String` , if needed, and the two strings are concatenated. String conversion of a non-`null` value is performed by invoking the `toString()` method of the value. If the value is `null`, the value is converted to `"null"`.

The type of the result is `String`.

For example, `"one " + 2 + here` evaluates to something like `one 2(Place 0)`.

11.15 Logical negation

The operand of the unary `!` operator must be of type `x10.lang.Boolean`. The type of the result is `Boolean`. If the value of the operand is `true`, the result is `false`; if if the value of the operand is `false`, the result is `true`.

11.16 Boolean logical operations

Operands of the binary boolean logical operators must be of type `Boolean`. The type of the result is `Boolean`

The & operator evaluates to `true` if both of its operands evaluate to `true`; otherwise, the operator evaluates to `false`.

The `|` operator evaluates to `false` if both of its operands evaluate to `false`; otherwise, the operator evaluates to `true`.

11.17 Boolean conditional operations

Operands of the binary boolean conditional operators must be of type `Boolean`. The type of the result is `Boolean`

The && operator evaluates to `true` if both of its operands evaluate to `true`; otherwise, the operator evaluates to `false`. Unlike the logical operator &, if the first operand is `false`, the second operand is not evaluated.

The `||` operator evaluates to `false` if both of its operands evaluate to `false`; otherwise, the operator evaluates to `true`. Unlike the logical operator `||`, if the first operand is `true`, the second operand is not evaluated.

11.18 Relational operations

The relational operations compare numbers, producing `Boolean` results.

The < operator evaluates to `true` if the left operand is less than the right. The <= operator evaluates to `true` if the left operand is less than or equal to the right. The > operator evaluates to `true` if the left operand is greater than the right. The >= operator evaluates to `true` if the left operand is greater than or equal to the right.

Floating point comparison is determined by the IEEE 754 standard. Thus, if either operand is NaN, the result is `false`. Negative zero and positive zero are considered to be equal. All finite values are less than positive infinity and greater than negative infinity.

11.19 Conditional expressions

ConditionalExpression ::= *Expression* ? *Expression* : *Expression*

A conditional expression evaluates its first subexpression (the condition); if `true` the second subexpression (the consequent) is evaluated; otherwise, the third subexpression (the alternative) is evaluated.

The type of the condition must be `Boolean`. The type of the conditional expression is some common ancestor (as constrained by §4.9) of the types of the consequent and the alternative.

For example, `a == b ? 1 : 2` evaluates to `1` if `a` and `b` are the same, and `2` if they are different. As the type of `1` is `Int{self==1}` and of `2` is `Int{self==2}`, the type of the conditional expression has the form `Int{c}`, where `self==1` and `self==2` both imply `c`. For example, it might be `Int{true}` – or perhaps it might be `Int{self != 8}`. Note that this term has no most accurate type in the X10 type system.

11.20 Stable equality

EqualityExpression	::=	*Expression* == *Expression*
	|	*Expression* != *Expression*

The == and != operators provide a fundamental, though non-abstract, notion of equality. `a==b` is true if the values of `a` and `b` are extremely identical, in a sense defined shortly. `a != b` is true iff `a==b` is false.

- If `a` and `b` are values of object type, then `a==b` holds if `a` and `b` are the same object.
- If one operand is `null`, then `a==b` holds iff the other is also `null`.
- If the operands both have struct type, then they must be structurally equal; that is, they must be instances of the same struct and all their fields or components must be ==.
- The definition of equality for function types is specified in §10.5.
- If the operands have numeric types, they are coerced into the larger of the two types and then compared for numeric equality.

The predicates == and != may not be overridden by the programmer. Note that `a==b` is a form of *stable equality*; that is, the result of the equality operation is not affected by the mutable state of the program, after evaluation of `a` and `b`.

11.21 Allocation

NewExpression	::=	`new` *ClassName TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)` *ClassBody*$^?$
	|	`new` *InterfaceName TypeArguments*$^?$ `(` *ArgumentList*$^?$ `)` *ClassBody*

An allocation expression creates a new instance of a class and invokes a constructor of the class. The expression designates the class name and passes type and value arguments to the constructor.

The allocation expression may have an optional class body. In this case, an anonymous subclass of the given class is allocated. An anonymous class allocation may also specify a single super-interface rather than a superclass; the superclass of the anonymous class is `x10.lang.Object`.

If the class is anonymous—that is, if a class body is provided—then the constructor is selected from the superclass. The constructor to invoke is selected using the same rules as for method invocation (§11.6).

The type of an allocation expression is the return type of the constructor invoked, with appropriate substitutions of actual arguments for formal parameters, as specified in §11.6.

It is illegal to allocate an instance of an `abstract` class. It is illegal to allocate an instance of a class or to invoke a constructor that is not visible at the allocation expression.

Note that instantiating a struct type uses function application syntax, not `new`. As structs do not have subclassing, there is no need or possibility of a *ClassBody*.

11.22 Casts

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

UnaryExpression	::=	*CastExpression*
CastExpression	::=	*UnaryExpression* `as` *Type*

The result of this operation is a value of the given type if the cast is permissible at run time, and either a compile-time error or a runtime exception (`x10.lang.TypeCastException`) if it is not.

When evaluating `E as T{c}`, first the value of `E` is converted to type `T` (which may fail), and then the constraint `{c}` is checked.

- If `T` is a primitive type, then `E`'s value is converted to type `T` according to the rules of §11.27.1.
- If `T` is a class, then the first half of the cast succeeds if the run-time value of `E` is an instance of class `T`, or of a subclass
- If `T` is an interface, then the first half of the cast succeeds if the run-time value of `E` is an instance of a class implementing `T`.

- If `T` is a struct type, then the first half of the cast succeeds if the run-time value of `E` is an instance of `T`.
- If `T` is a function type, then the first half of the cast succeeds if the run-time value of `X` is a function of precisely that type.

If the first half of the cast succeeds, the second half – the constraint `{c}` – must be checked. In general this will be done at runtime, though in special cases it can be checked at compile time. For example, `n as Int{self != w}` succeeds if `n != w` — even if `w` is a value read from input, and thus not determined at compile time.

The compiler may forbid casts that it knows cannot possibly work. If there is no way for the value of `E` to be of type `T{c}`, then `E as T{c}` can result in a static error, rather than a runtime error. For example, `1 as Int{self==2}` may fail to compile, because the compiler knows that `1`, which has type `Int{self==1}`, cannot possibly be of type `Int{self==2}`.

11.23 `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. At run time, the result of this operator is `true` if the *RelationalExpression* can be coerced to *Type* without a `TypeCastException` being thrown or static error occurring. Otherwise the result is `false`. This determination may involve checking that the constraint, if any, associated with the type is true for the given expression.

For example, `3 instanceof Int{self==x}` is an overly-complicated way of saying `3==x`.

11.24 Subtyping expressions

SubtypingExpression	::=	*Expression* `<:` *Expression*
	|	*Expression* `:>` *Expression*
	|	*Expression* == *Expression*

The subtyping expression $\texttt{T}_1$ `<:` $\texttt{T}_2$ evaluates to `true` $\texttt{T}_1$ is a subtype of $\texttt{T}_2$.

The expression $\texttt{T}_1$ `:>` $\texttt{T}_2$ evaluates to `true` $\texttt{T}_2$ is a subtype of $\texttt{T}_1$.

The expression $\texttt{T}_1$ `==` $\texttt{T}_2$ evaluates to `true` $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ and if $\texttt{T}_2$ is a subtype of $\texttt{T}_1$.

Subtyping expressions are particularly useful in giving constraints on generic types. `x10.util.Ordered[T]` is an interface whose values can be compared with values of type `T`. In particular, `T <: x10.util.Ordered[T]` is true if values of type `T` can be compared to other values of type `T`. So, if we wish to define a generic class `OrderedList[T]`, of lists whose elements are kept in the right order, we need the elements to be ordered. This is phrased as a constraint on `T`:

```
class OrderedList[T]{T <: x10.util.Ordered[T]} {
  // ...
}
```

11.25 Contains expressions

ContainsExpression ::= *Expression* `in` *Expression*

The expression `p in r` tests if a value `p` is in a collection `r`; it evaluates to `r.contains(p)`. The collection `r` must be of type `Collection[T]` and the value `p` must be of type `T`.

11.26 Rail constructors

RailConstructor ::= `[` *Expressions* `]`
Expressions ::= *Expression* (`,` *Expression*)*

The rail constructor $[\texttt{a}_0, \ldots, \texttt{a}_{k-1}]$ creates an instance of `ValRail` with length k, whose ith element is $\texttt{a}_i$. The element type of the rail is a common ancestor of the types of the $\texttt{a}_i$'s, as per §4.9.

```
val a : ValRail[Int] = [1,3,5];
val b : ValRail[Any] = [1, a, "please"];
```

Since arrays are subtypes of `(Point) => T`, rail constructors can be passed into the `Array` and `ValArray` constructors as initializer functions.

Rail constructors of type `ValRail[Int]` and length `n` may be implicitly converted to type `Point{rank==n}`. Rail constructors of type `ValRail[Region]` and length `n` may be implicitly converted to type `Region{rank==n}`.

```
val a : Point{rank==4} = [1,2,3,4];
val b : Region{rank==2} = [ -1 .. 1, -2 .. 2];
```

11.27 Coercions and conversions

X10 v2.0 supports the following coercions and conversions.

11.27.1 Coercions

A *coercion* does not change object identity; a coerced object may be explicitly coerced back to its original type through a cast. A *conversion* may change object identity if the type being converted to is not the same as the type converted from. X10 permits user-defined conversions (§11.27.2).

Subsumption coercion. A subtype may be implicitly coerced to any supertype.

Explicit coercion (casting with `as`) An object of any class may be explicitly coerced to any other class type using the `as` operation. If `Child <: Person` and `rhys:Child`, then

```
rhys as Person
```

is an expression of type `Person`.

If the value coerced is not an instance of the target type, a `ClassCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

Limitation *It is currently a static error, rather than the specified* `ClassCastException`*, when the cast is statically determinable to be impossible.*

Effects of explicit numeric coercion Coercing a number of one type to another type gives the best approximation of the number in the result type, or a suitable disaster value if no approximation is good enough.

- Casting a number to a *wider* numeric type is safe and effective, and can be done by an implicit conversion as well as an explicit coercion. For example, `4 as Long` produces the `Long` value of 4.

- Casting a floating-point value to an integer value truncates the digits after the decimal point, thereby rounding the number towards zero. `54.321 as Int` is `54`, and `-54.321 as Int` is `-54`. If the floating-point value is too large to represent as that kind of integer, the coercion returns the largest or smallest value of that type instead: `1e110 as Int` is `Int.MAX_VALUE`, `2147483647`.

- Casting a `Double` to a `Float` normally truncates digits: `0.12345678901234567890 as Float` is `0.12345679f`. This can turn a nonzero `Double` into `0.0f`, the zero of type `Float`: `1e-100 as Float` is `0.0f`. Since `Double`s can be as large as about `1.79E308` and `Float`s can only be as large as about `3.4E38f`, a large `Double` will be converted to the special `Float` value of `Infinity`: `1e100 as Float` is `Infinity`.

- Integers are coerced to smaller integer types by truncating the high-order bits. If the value of the large integer fits into the smaller integer's range, this gives the same number in the smaller type: `12 as Byte` is the `Byte`-sized 12, `-12 as Byte` is -12. However, if the larger integer *doesn't* fit in the smaller type, the numeric value and even the sign can change: `254 as Byte` is `Byte`-sized `-2`.

11.27.2 Conversions

Widening numeric conversion. A numeric type may be implicitly converted to a wider numeric type. In particular, an implicit conversion may be performed between a numeric type and a type to its right, below:

```
Byte < Short < Int < Long < Float < Double
```

String conversion. Any object that is an operand of the binary + operator may be converted to `String` if the other operand is a `String`. A conversion to `String` is performed by invoking the `toString()` method of the object.

User defined conversions. The user may define conversion operators from type `A` *to* a container type `B` by specifying a method on `B` as follows:

```
public static operator (r: A): T = ...
```

The return type `T` should be a subtype of `B`. The return type need not be specified explicitly; it will be computed in the usual fashion if it is not. However, it is good practice for the programmer to specify the return type for such operators explicitly.

For instance, the code for `x10.lang.Point` contains:

```
public static global safe operator (r: Rail[int])
   : Point(r.length) = make(r);
```

The compiler looks for such operators on the container type `B` when it encounters an expression of the form `r as B` (where `r` is of type `A`). If it finds such a method, it sets the type of the expression `r as B` to be the return type of the method. Thus the type of `r as B` is guaranteed to be some subtype of `B`.

Example 11.27.1 Consider the following code:

```
val p  = [2, 2, 2, 2, 2] as Point;
val q = [1, 1, 1, 1, 1] as Point;
val a = p - q;
```

This code fragment compiles successfully, given the above operator definition. The type of `p` is inferred to be `Point(5)` (i.e. the type `Point{self.rank==5}`. Similarly for `q`. Hence the application of the operator “`-`” is legal (it requires both arguments to have the same rank). The type of `a` is computed as `Point(5)`. □

12 Statements

This chapter describes the statements in the sequential core of X10. Statements involving concurrency and distribution are described in §14.

12.1 Empty statement

Statement ::= ;

The empty statement ; does nothing. It is useful when a loop header is evaluated for its side effects. For example, the following code sums the elements of a rail.

```
var sum: Int = 0;
for (var i: Int = 0; i < a.length; i++, sum += a(i))
    ;
```

12.2 Local variable declaration

Short-lived variables are introduced by local variables declarations, as described in §5. Local variables may be declared only within a block statement (§12.3). The scope of a local variable declaration is the statement itself and the subsequent statements in the block.

```
if (a > 1) {
   val b = a/2;
   var c : Int = 0;
   // b and c are defined here
}
```

```
// b and c are not defined here.
```

12.3 Block statement

Statement	::=	*BlockStatement*
BlockStatement	::=	{ *Statement** }

A block statement consists of a sequence of statements delimited by "{" and "}". When a block is evaluated, the statements inside of it are evaluated in order. Blocks are useful for putting several statements in a place where X10 asks for a single one, such as the consequent of an `if`, and for limiting the scope of local variables.

```
if (b) {
  // This is a block
  val v = 1;
  S1(v);
  S2(v);
}
```

12.4 Expression statement

Statement	::=	*ExpressionStatement*
ExpressionStatement	::=	*StatementExpression* ;
StatementExpression	::=	*Assignment*
	|	*Allocation*
	|	*Call*

The expression statement evaluates an expression, ignoring the result. Side effects of the expression occur. The expression must be either an assignment, an allocation, or a call; *viz.*, a sort of expression which could possibly have side effects.

```
class StmtEx {
  def this() { x10.io.Console.OUT.println("New StmtEx made");  }
  static def call() { x10.io.Console.OUT.println("call!");  }
```

```
    def example() {
       var a : Int = 0;
       a = 1; // assignment
       new StmtEx(); // allocation
       call(); // call
    }
}
```

12.5 Labeled statement

Statement	::=	*LabeledStatement*
LabeledStatement	::=	*Identifier* : *LoopStatement*

Loop statements (`for`, `while`, `do`, `ateach`, `foreach`) may be labeled. The label may be used as the target of a break or continue statement. The scope of a label is the statement labeled.

```
lbl : for ((i) in 1..10) {
   for ((j) in i..10) {
      if (a(i,j) == 0) break lbl;
      if (a(i,j) == 1) continue lbl;
      do_things_to(a(i,j));
   }
}
```

12.6 Break statement

Statement	::=	*BreakStatement*
BreakStatement	::=	`break` *Identifier*$^?$

An unlabeled break statement exits the currently enclosing loop or switch statement. An labeled break statement exits the enclosing loop or switch statement with the given label. It is illegal to break out of a loop not defined in the current method, constructor, initializer, or closure.

The following code searches for an element of a two-dimensional array and breaks out of the loop when it is found:

```
var found: Boolean = false;
outer: for (var i: Int = 0; i < a.length; i++)
    for (var j: Int = 0; j < a(i).length; j++)
        if (a(i)(j) == v) {
            found = true;
            break outer;
        }
```

12.7 Continue statement

Statement	::=	*ContinueStatement*
ContinueStatement	::=	`continue` *Identifier*$^?$

An unlabeled `continue` skips the rest of the current iteration of the innermost enclosing loop, and proceeds on to the next. A labeled `continue` does the same to the enclosing loop with that label. It is illegal to continue a loop not defined in the current method, constructor, initializer, or closure.

12.8 If statement

Statement	::=	*IfThenStatement*
	|	*IfThenElseStatement*
IfThenStatement	::=	`if (` *Expression* `)` *Statement*
IfThenElseStatement	::=	`if (` *Expression* `)` *Statement* `else` *Statement*

An if statement comes in two forms: with and without an else clause.

The if-then statement evaluates a condition expression, which must be of type `Boolean`. If the condition is `true`, it evaluates the then-clause. If the condition is `false`, the if-then statement completes normally.

The if-then-else statement evaluates a condition expression and evaluates the then-clause if the condition is `true`; otherwise, the `else`-clause is evaluated.

As is traditional in languages derived from Algol, the if-statement is syntactically ambiguous. That is,

```
if (B1) if (B2) S1 else S2
```

could be intended to mean either

```
if (B1) { if (B2) S1 else S2 }
```

or

```
if (B1) {if (B2) S1} else S2
```

X10, as is traditional, attaches an `else` clause to the most recent `if` that doesn't have one. This example is interpreted as `if (B1) { if (B2) S1 else S2 }`.

12.9 Switch statement

Statement	::=	*SwitchStatement*
SwitchStatement	::=	`switch` (*Expression*) { *Case*$^+$ }
Case	::=	`case` *Expression* : *Statement**
	|	`default` : *Statement**

A switch statement evaluates an index expression and then branches to a case whose value equal to the value of the index expression. If no such case exists, the switch branches to the `default` case, if any.

Statements in each case branch evaluated in sequence. At the end of the branch, normal control-flow falls through to the next case, if any. To prevent fall-through, a case branch may be exited using a `break` statement.

The index expression must be of type `Int`. Case labels must be of type `Int`, `Byte`, `Short`, or `Char` and must be compile-time constants. Case labels cannot be duplicated within the `switch` statement.

In the following example, case `1` falls through to case3 `2`. The other cases are separated by `break`s.

```
switch (i) {
  case 1: println("one, and ");
  case 2: println("two");
          break;
```

```
    case 3: println("three");
            break;
    default: println("Something else");
             break;
}
```

12.10 While statement

Statement	::=	*WhileStatement*
WhileStatement	::=	`while` `(` *Expression* `)` *Statement*

A while statement evaluates a `Boolean`-valued condition and executes a loop body if `true`. If the loop body completes normally (either by reaching the end or via a `continue` statement with the loop header as target), the condition is reevaluated and the loop repeats if `true`. If the condition is `false`, the loop exits.

```
while (n > 1) {
   n = (n % 2 == 1) ? 3*n+1 : n/2;
}
```

12.11 Do–while statement

Statement	::=	*DoWhileStatement*
DoWhileStatement	::=	`do` *Statement* `while` `(` *Expression* `)` `;`

A do-while statement executes a loop body, and then evaluates a `Boolean`-valued condition expression. If `true`, the loop repeats. Otherwise, the loop exits.

12.12 For statement

Statement	::=	*ForStatement*
	|	*EnhancedForStatement*
ForStatement	::=	`for` `(` *ForInit*$^?$ `;` *Expression*$^?$ `;` *ForUpdate*$^?$ `)` *Statement*
ForInit	::=	*StatementExpression* (`,` *StatementExpression*)*
	|	*LocalVariableDeclaration*
ForUpdate	::=	*StatementExpression* (`,` *StatementExpression*)*
EnhancedForStatement	::=	`for` `(` *Formal* `in` *Expression* `)` *Statement*

`for` statements provide bounded iteration, such as looping over a list. It has two forms: a basic form allowing near-arbitrary iteration, *a la* C, and an enhanced form designed to iterate over a collection.

A basic `for` statement provides for arbitrary iteration in a somewhat more organized fashion than a `while`. `for(init; test; step)body` is equivalent to:

```
{
   init;
   while(test) {
      body;
      step;
   }
}
```

`init` is performed before the loop, and is traditionally used to declare and/or initialize the loop variables. It may be a single variable binding statement, such as `var i:Int = 0` or `var i:Int=0, j:Int=100`. (Note that a single variable binding statement may bind multiple variables.) Variables introduced by `init` may appear anywhere in the `for` statement, but not outside of it. Or, it may be a sequence of expression statements, such as `i=0, j=100`, operating on already-defined variables. If omitted, `init` does nothing.

`test` is a Boolean-valued expression; an iteration of the loop will only proceed if `test` is true at the beginning of the loop, after `init` on the first iteration or or `step` on later ones. If omitted, `test` defaults to `true`, giving a loop that will run until stopped by some other means such as `break`, `return`, or `throw`.

`step` is performed after the loop body, between one iteration and the next. It traditionally updates the loop variables from one iteration to the next: *e.g.*, `i++` and `i++,j--`. If omitted, `step` does nothing.

`body` is a statement, often a code block, which is performed whenever `test` is true. If omitted, `body` does nothing.

An enhanced for statement is used to iterate over a collection, or other structure designed to support iteration by implementing the interface `Iterable[T]`. The loop variable must be of type `T`, or destructurable from a value of type `T` (§5; in practice, this means that `for ((i) in 1..10)` iterates over numbers from 1 to 10, while `for (i in 1..10` iterates over `Points` from 1 to 10). Each iteration of the loop binds the iteration variable to another element of the collection.

```
var sum : Int = 0;
for ((i) in 1..n) sum += i;
```

Certain common variant cases are accepted. If collection is of type `Region`, the iteration variable may be of type `Point`. If the iteratable `e` is of type `Dist` or `Array`, it is treated as if it were `e.region`.

12.13 Throw statement

Statement	::=	*ThrowStatement*
ThrowStatement	::=	`throw` *Expression* `;`

The `throw` statement throws an exception, which must be an instance of `x10.lang.Throwable`.

For example, the following code checks if an index is in range and throws an exception if not.

```
if (i < 0 || i >= x.length)
    throw new MyIndexOutOfBoundsException();
```

12.14 Try–catch statement

Statement	::=	*TryStatement*
TryStatement	::=	`try` *BlockStatement Catch*$^+$
	|	`try` *BlockStatement Catch** *Finally*
Catch	::=	`catch` `(` *Formal* `)` *BlockStatement*
Finally	::=	`finally` *BlockStatement*

Exceptions are handled with a `try` statement. A `try` statement consists of a `try` block, zero or more `catch` blocks, and an optional `finally` block.

First, the `try` block is evaluated. If the block throws an exception, control transfers to the first matching `catch` block, if any. A `catch` matches if the value of the exception thrown is a subclass of the `catch` block's formal parameter type.

The `finally` block, if present, is evaluated on all normal and exceptional control-flow paths from the `try` block. If the `try` block completes normally or via a `return`, a `break`, or a `continue` statement, the `finally` block is evaluated, and then control resumes at the statement following the `try` statement, at the branch target, or at the caller as appropriate. If the `try` block completes exceptionally, the `finally` block is evaluated after the matching `catch` block, if any, and then the exception is rethrown.

It is a static error to attempt to catch an exception type which is not throwable by the block.

12.15 Return statement

Statement	::=	*ReturnStatement*
ReturnStatement	::=	`return` *Expression* `;`
	|	`return ;`

Methods and closures may return values using a return statement. If the method's return type is explicity declared `Void`, the method must return without a value; otherwise, it must return a value of the appropriate type.

13 Places

An X10 place is a repository for data and activities, corresponding loosely to a process or a processor. Places induce a concept of "local". 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's system of places is designed to make this obvious. Programmers are aware of the places of their data, and know when they are incurring communication costs, but the actual operation to do so is easy. It's not hard to use non-local data; it's simply hard to to do so accidentally.

The set of places available to a computation is determined at the time that the program is started, and remains fixed through the run of the program. See the `README` documentation on how to set command line and configuration options to set the number of places.)

Places are first-class values in X10, as instances of the built-in struct, `x10.lang.Place`. `Place` provides a number of useful ways to query places, such as `Place.places`, a `ValRail` of all the places available to the current run of the program.

Every object `ob` created by the program has a *home place* `ob.home`. Accesses to `ob` from activities located at `ob.home` are privileged. Mutable fields can only be changed from `ob.home`, and normal fields and methods can only be accessed from there.

However, objects can be referred to from anywhere. Places other than `ob.home` may have *remote references* to `ob`. Remote references convey fewer privileges than local ones, but they are far from useless. Fields and methods can be defined to be `global`, usable from anywhere.

13.1 The Structure of Places

Places are numbered 0 through `Places.MAX_PLACES`, stored in the field `pl.id`. The `ValRail` `Place.places` contains the places of the program, in numeric order. The program starts by executing a `main` method at `Place.FIRST_PLACE`, which is `Place.places(0)`; see §14.5.

Operations on places include `pl.next()`, which gives the next entry (looping around) in `Place.places` and its opposite `pl.prev()`. In particular, `here.next()` means "a place other than `here`", except in single-place executions. There are also a number of tests, like `pl.isSPE()` and `pl.isCUDA()`, which test for particular kinds of processors.

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

Place expressions (*viz.*, expressions of type `Place`), such as `here` and `ob.home`, are used in `at` and `asynch` statements.

13.2 `here`

The variable `here` is always bound to the place at which the current computation is running, in the same way that `this` is always bound to the instance of the current class (for non-static code), or `self` is bound to the instance of the type currently being constrained. `here` may denote different places in the same method body, due to place-shifting operations. In the following code, `here` has one value at `h0`, and a different one at `h1`.

```
val h0 = here;
at (here.next()) {
  val h1 = here;
  assert (h0 != h1);
}
```

(Similar examples show that `self` and `this` have the same behavior: `self` can be modified by constrained types appearing inside of type constraints, and `this` by inner classes.)

Expression ::= `here`

Example 13.2.1 For example, the following method looks through a collection of `Thing`s for ones which belong in the current place `here`, and deals with the things which do. Note that every object `thing` has a property `thing.home` giving its home location.

```
public static def dealWithLocal(things: Rail[Thing]) {
   for(thing in things) {
       if (thing.home == here)
          dealWith(thing);
   }
}
```

□

`here` is frequently used in constraints, quite often of the form `ob.home == here`. Such constraints are necessary to check that a non-global method can be called on `ob`:

```
val ob : Thing{self.home == here} = new Thing();
ob.nonGlobalMethod();
```

This idiom is so common and useful that the constraint `T{self.home==here}` can be abbreviated as `T!`:

```
val ob : Thing! = new Thing();
ob.nonGlobalMethod();
```

LimitationIn the current implementation, sometimes `T{self.home==here}` *does not work, though* `T!` *does.*

14 Activities

An *activity* is a statement being executed, independently, with its own local variables; it may be thought of as a very light-weight thread. An X10 computation may have many concurrent *activities* executing at any give time. All X10 code runs as part of an activity; when an X10 program is started, the `main` method is invoked in an activity, called the *root activity*.

Activities coordinate their execution by various control and data structures. For example, ‘ `await (x==0);` blocks the current activity until some other activity sets `x` to zero. However, activities determine the places at which they may be blocked and resumed, by `await` and similar constructs. There are no means by which one activity can arbitrarily interrupt, block, or resume another, no method `activity.interrupt()`.

An activity may be *running*, *blocked* on some condition or *terminated*. If terminated, it is terminated in the same way that its statement is: in particular, if the statement terminates abruptly, the activity terminates abruptly for the same reason. (§14.1).

Activities can be long-running entities with a good deal of local state. In particular they can involve recursive method calls (and therefore have runtime stacks). However, activities can also be short-running light-weight entities, *e.g.*, it is reasonable to have an activity that simply increments a variable.

An activity may asynchronously and in parallel launch activities at other places. Every activity save the initial `main` activity is spawned by another. Thus, at any instant, the activities in a program form a tree.

X10 uses this tree in crucial ways. First is the distinction 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. (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 have, recursively, terminated globally. For example, consider:

```
async {s1();}
async {s2();}
```

The primary activity spawns two child activities and then terminates locally, very quickly. The child activities may take arbitrary amounts of time to terminate (and may spawn grandchildren). When `s1()`, `s2()`, and all their descendants terminate locally, then the primary activity terminates globally.

The program as a whole terminates when the root activity terminates globally. In particular, X10 does not permit the creation of daemon threads—threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§14.5).

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

14.1 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 and the concept of global termination permits X10 to adopt a more powerful exception 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 blocked at a statement (such as the `finish` statement §14.4) awaiting the termination of B (and possibly other activities). 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]

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

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] In Java, exceptions may be overlooked because there is no good place to put a `try-catch` block; this is never the case in X10.

14.2 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 `async` statement:

Statement	::=	*AsyncStatement*
AsyncStatement	::=	`async` *PlaceExpressionSingleList*$^?$ *Statement*
PlaceExpressionSingleList	::=	`(` *PlaceExpression* `)`
PlaceExpression	::=	*Expression*

The optional place expression `e` will often be of type `Place`; *e.g.*, `here` or `d(p)` for some distribution `d` and point `p` (§13). It may also be an object type; `e` is treated as `e.home` if `e` is of type `x10.lang.Object`. As other kinds of values have no usefully-defined place, it is a static error to use them for `e`.

Note that the array subscript expression `a(i)`, when used as a place expression evaluates to `a(i).home`, *viz.*. the home of the contents of `a(i)` In general, this is not the same place that the array cell itself is, *viz.*. `a.dist(i)`. Accesses to `a(i)` should typically be guarded by the place expression `a.dist(i)`:

```
async (a.dist(i)) {
  a(i) += 1;
}
```

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 occurring in the statement. (The place must be such that the statement can be executed safely,

[2] In X10 v2.0 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.

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 place `P` (or `P.home` if `P` is of an object type), to execute `S`. The statement terminates locally as soon as B is launched. The activation path for B is that of A augmented by the information that A is the parent of B. 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 §14.4), otherwise it is propagated through A to A's root activity. Note that while A is running, exceptions thrown by activities it has already spawned may propagate through it up to its root activity, without A noticing.

Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the set of activities at the target place and will be executed based on the local scheduler's decisions. If some particular sequencing of events is needed, `when`, `atomic`, `await`, `finish`, clocks, and other X10 constructs can be used. 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 RULE: 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 anonymous 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, §15) provided that such variables are (implicitly or explicitly) `val`.

14.3 Place changes

An activity may change place using the `at` statement or `at` expression:

Statement	::=	*AtStatement*
AtStatement	::=	`at` *PlaceExpressionSingleList Statement*
Expression	::=	*AtExpression*
AtExpression	::=	`at` *PlaceExpressionSingleList ClosureBody*

[3] X10 v2.0 does not specify a particular algorithm; this will be fixed in future versions.

The statement `at (p) S` executes the statement `S` synchronously at place `p`. The expression `at (p) E` executes the statement `E` synchronously at place `p`, returning the result to the originating place. *E.g.*, if `obj` is an object with a non-`global` method `meth()`, the general way to call the method from anywhere is

```
at(x.home) x.meth();
```

Or, if you want to capture the result of the method call, use the `at`-expression:

```
val res = at(x.home)x.meth();
```

`at(p)S` does *not* start a new activity. It should be thought of as transporting the current activity to `p`, running `S` there, and then transporting it back. If you want to start a new activity, use `async`; if you want to start a new activity at `p`, use `at(p) async S`.

As a consequence of this, `S` may contain constructs which only make sense within a single activity. For example,

```
for(x in things)
  if (at(x.home) x.nice())
      return x;
```

returns the first nice thing in a collection. If we had used `async at(x.home)`, this would not be allowed; you can't `return` from an `async`.

14.4 Finish

The statement `finish S` converts global termination to local termination and introduces a root activity. It executes `S`, and then waits for all activities spawned by `S`, directly or indirectly, to finish. It also collects exceptions thrown by those activities.

Statement	::=	*FinishStatement*
FinishStatement	::=	`finish` *Statement*

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`. `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, `x10.lang.MultipleExceptions` 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 little effect after the first `finish`: `finish finish S` is indistinguishable from `finish S` if `S` throws no exceptions. (If `S` throws exceptions, `finish S` wraps them in one layer of `MultipleExceptions` and `finish finish S` in two layers.)

Interaction with clocks. `finish S` interacts with clocks (§15). 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 v2.0 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.*

14.5 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 def main(a: Rail[String])` 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 Rail of strings created from the command line arguments. This single activity is the root activity for the entire computation. (See §13 for a discussion of places.)

14.6 Foreach statements

Statement ::= *ForEachStatement*
ForEachStatement ::= `foreach` `(` *Formal* `in` *Expression* `)` *Statement*

The `foreach` statement is a parallel version of the enhanced `for` statement (§12.12). `for(x in C)S` executes `S` *sequentially*, with everything happening `here`. `foreach(x in C)S` executes `S` for each iteration of the loop *in parallel*, located at `x.home`. It is thus equivalent to:

```
foreach (x in C)
  async at (x.home) S
```

As a common and useful special case, `C` may be a `Dist` or an `Array`. For both of these, `foreach(x in C)S` is treated just like `foreach(x in C.region)S`. `x` ranges over the `Points` of the region. Each activity that `foreach` starts is located at `here` – the same place that the `foreach` statement itself is executing. (If you want to start an activity at the place where the array element `C(p)` is located, use `ateach` (§14.7) instead of `foreach`.)

Exceptions thrown by `S`, like other exceptions in `asyncs`, are propagated to the root activity of the `foreach`.

14.7 Ateach statements

Statement ::= *AtEachStatement*
AtEachStatement ::= `ateach` `(` *Formal* `in` *Expression* `)` *Statement*

The `ateach` statement is similar to the `foreach` statement, but it spawns activites at each place of a distribution. In `ateach(p in D) S`, `D` must be either of type `Dist` or of type `DistArray[T]`, and `p` will be of type `Point`.

This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, if `D` is a `Dist`, `ateach(p in D) S` could be implemented as:

```
foreach (p in D.region)
  async (D(p)) S(p)
```

However, the compiler may implement it more efficiently to avoid extraneous communications. In particular, it is recommended that `ateach(p in D)S` be implemented as the following code, which coordinates with each place of `D` just once, rather than once per element of `D` at that place:

```
foreach (p in D.places()) at (p) {
    foreach (pt in D|here) {
        S(p);
    }
}
```

If `e` is an `DistArray[T]`, then `ateach (p in e)S` is identical to `ateach(p in e.dist)S`; the iteration is over the array's underlying distribution. The code below is a common and generally efficient way to work with the elements of a distributed array:

```
ateach(p in A)
  dealWith(A(p));
```

14.8 Futures

X10 provides syntactic support for *asynchronous expressions*, also known as futures. Futures let you start a subcomputation that will return a value, and let it proceed asynchronously. When you need its value, apply `force()` to the future, which will wait until that computation is done (if necessary). Then, calling the future as a nullary function retrieves the value of the computation.

```
val future_a = future "expression that takes a long time";
val b = "Some long computation happens here";
future_a.force();
val a = future_a(); // get its value
```

Primary	::=	*FutureExpression*
FutureExpression	::=	`future` *PlaceExpressionSingleList*$^?$ *ClosureBody*

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 `val`.

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:

```
package x10.lang;
public interface Future[T] implements () => T {
  public def apply(): T;
  global def forced(): Boolean;
  global def force(): T;
}
```

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.

`Future[T]` is a subtype of the function type `() => T`. Invoking—*forcing*—the future `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 the function (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 queries.

14.9 At expressions

Expression ::= `at` `(` *Expression* `)` *Expression*

An `at` expression evaluates an expression synchronously at the given place and returns its value. If `fld` is a non-global field of `ob`, then `ob.fld` can be read (from anywhere) by:

```
val f = at(ob) ob.fld;
```

The expression evaluation may spawn asynchronous activities. The `at` expression will return without waiting for those activities to terminate. That is, `at` does not have built-in `finish` semantics.

14.10 Shared variables

Limitation *Shared variables are not currently implemented.*

A *shared local variable* is declared with the annotation `shared`. It can be accessed within any control construct in its scope, including `async`, `at`, `future` and closures.

Note that the lifetime of some of these constructs may outlast the lifetime of the scope – requiring the implementation to allocate them outside the current stack frame.

14.11 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.

14.11.1 Unconditional atomic blocks

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

Statement	::=	*AtomicStatement*
AtomicStatement	::=	`atomic` *Statement*
MethodModifier	::=	`atomic`

For the sake of efficient implementation X10 v2.0 requires that the atomic block be *analyzable*, that is, the set of locations that are read and written by the *Block-*

Statement 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 blocked. 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 [6], and can be used to implement non-blocking concurrent algorithms.

STATIC SEMANTICS RULE: In `atomic S`, `S` may include method calls, conditionals, etc.

It may *not* include an `async` activity (such as creation of a `future`).

It may *not* include any statement that may potentially block at runtime (e.g., `when`, `force` operations, `next` operations on clocks, `finish`).

All locations accessed in an atomic block must statically satisfy the *locality condition*: they must belong to the place of the current activity.

The compiler checks for this condition by checking whether the statement could be the body of a `void` method annotated with `safe` at that point in the code (§8.4.4).

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

$$\texttt{atomic \{s1; atomic s2\}} \;=\; \texttt{atomic \{s1; s2\}} \qquad (14.1)$$

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

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

Example

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

```
var target:Object = null;
public atomic def CAS(old: Object, new: Object): Boolean {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

14.11.2 Conditional atomic blocks

Conditional atomic blocks are of the form:

Statement	::=	*WhenStatement*
WhenStatement	::=	`when` `(` *Expression* `)` *Statement*
	|	*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 intermmittently. 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 RULE: 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 (as for the `sequential` qualifier on methods) and to have a statically determinable upper bound on their execution (as for the `nonblocking` qualifier on methods). 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.

Example 14.11.1 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 {
  var datum: Object = null;
  var filled: Boolean = false;
```

```
    public def send(v: Object) {
      when (!filled) {
        this.datum = v;
        this.filled = true;
      }
    }
    public def receive(): Object {
      when (filled) {
        v: Object = datum;
        datum = null;
        filled = false;
        return v;
      }
    }
  }
```

□

15 Clocks

Many concurrent algorithms proceed in phases: in phase k, several activities work independently, but synchronize together before proceeding on to phase $k + 1$. X10 supports this communication structure (and many variations on it) with a generalization of barriers called *clocks*. Clocks are designed so that programs which follow a simple syntactic discipline will not have either deadlocks or race conditions.

The following minimalist example of clocked code has two worker activities A and B, and three phases. In the first phase, each worker activity says its name followed by 1; in the second phase, by a 2, and in the third, by a 3. So, if `say` prints its argument, `A-1 B-1 A-2 B-2 A-3 B-3` would be a legitimate run of the program, but `A-1 A-2 B-1 B-2 A-3 B-3` (with `A-2` before `B-1`) would not.

The program creates a clock `cl` to manage the phases. Each worker does the work of its first phase, and then executes `next;` to signal that it is finished with that work. `next;` is blocking, and causes the worker to wait until all workers have finished with the phase – as measured by the clock `cl` to which they are both registered. Then they do the second phase, and another `next;` to make sure that neither proceeds to the third phase until both are ready. This example uses `finish` to wait for both workers to finish. The parent thread is also registered on the clock just as the workers are, and executes `next;next;` to run through the phases.

```
val cl = Clock.make();
finish{
  async clocked(cl) {// Activity A
    say("A-1");
    next;
    say("A-2");
    next;
    say("A-3");
```

```
    }// Activity A

    async clocked(cl) {// Activity B
      say("B-1");
      next;
      say("B-2");
      next;
      say("B-3");
    }// Activity B
    next;next;
  }
  say("All done");
```

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.

The primary operations that an activity `a` may perform on a clock `c` that it is registered upon are:

- It may *register* a newly-created activity on `c`: `async clocked(c){S}`.
- It may *unregister* itself from `c`, with `c.drop()`. After doing so, it can no longer use most primary operations on `c`.
- It may *resume* the clock, with `c.resume()`, indicating that it has finished with the current phase associated with `c` and is ready to move on to the next one.
- It may *wait* on the clock, with `c.next()`. This first does `c.resume()`, and then blocks the current activity until the start of the next phase, *viz.*, until all other activities registered on that clock have called `c.resume()`.

- It may *block* on all the clocks it is registered with simultaneously, by the command `next;`. This calls `c.next()` on all clocks `c` that the current activity is registered with.

- Other miscellaneous operations are available as well; see the `Clock` API.

Though clocks introduce a blocking statement (`next`) an important property of X10 is that clocks – when used with the `next;` *statement* only, without the `c.next()` method call – 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 (§14.4) guarantees that this cannot cause a cycle in the wait-for graph. A more rigorous proof may be found in [9].

15.1 Clock operations

There are two language constructs for working with clocks. `async clocked(cl) S` starts a new activity registered on one or more clocks. `next;` blocks the current activity until all the activities sharing clocks with it are ready to proceed. Clocks are objects, and have a number of useful methods on them as well.

15.1.1 Creating new clocks

Clocks are created using a factory method on `x10.lang.Clock`:

```
val timeSynchronizer: Clock = Clock.make();
```

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).

15.1.2 Registering new activities on clocks

The statement

```
async clocked (c1, c2, c3) S
```

starts a new activity, initially registered with clocks `c1`, `c2`, and `c3`, and running `S`. The activity running this code must be registered on those clocks. Furthermore, it cannot be quiescent on any of them (see §15.1.3), because that would introduce a race condition. Violations of these conditions are punished by the throwing of a `ClockUseException`.

An activity may check that it is registered on a clock `c` by the predicate `c.registered()`

NOTE: X10 does not contain a "register" operation that would allow an activity to discover a clock in a datastructure and register itself on it. Therefore, while a clock `c` may be stored in a data structure by one activity `a` and read from it by another activity `b`, `b` cannot do much with `c` unless it is already registered with it. In particular, it cannot register itself on `c`, and, lacking that registration, cannot register a sub-activity on it with `clocked(c) async S`.

15.1.3 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 `c.resume()`.

An activity may invoke `resume()` only on a clock it is registered with, and has not yet dropped (§15.1.5). A `ClockUseException` is thrown if this condition is violated. Nothing happens if the activity has already invoked a `resume` on this clock in the current phase. Otherwise, `c.resume()` indicates that the activity will not transmit `c` to an `async` (through a `clocked` clause), until it terminates, drops `c` or executes a `next`.

STATIC SEMANTICS RULE: It is a static error if any activity has a potentially live execution path from a `resume` statement on a clock `c` to a async spawn statement (which registers the new activity on `c`) unless the path goes through a `next` statement. (A `c.drop()` following a `c.resume()` is legal, as is `c.resume()` following a `c.resume()`.)

15.1.4 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 quiescence 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.

15.1.5 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`.

15.2 Program equivalences

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

$$\texttt{c.resume(); next;} \;=\; \texttt{next;} \tag{15.1}$$

$$\texttt{c.resume(); d.resume();} \;=\; \texttt{d.resume(); c.resume();} \tag{15.2}$$

$$\texttt{c.resume(); c.resume();} \;=\; \texttt{c.resume();} \tag{15.3}$$

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

16 Arrays

Limitation *This chapter is obsolete and needs extensive revision.*

This chapter provides an overview of the class `Array`, and its supporting classes `Point`, `Region` and `Dist`. All these classes are in the `x10.lang` package. For more details, please consider the API documentation.

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.dist` which may be used to obtain the underlying distribution.

16.0.1 Array types

Arrays in X10 are instances of the class `x10.lang.Array`. Because of the importance of arrays in X10, the language supports more concise syntax for accessing array elements and performing operations on arrays.

The array type `Array[T]` is the type of all reference arrays of base type `T`. Such an array can take on any distribution, over any region.

Both array classes implement the function type `(Point) => T`; the element of array `A` at point `p` may be accessed using the syntax `A(p)`. The `Array` class also implements the `Settable[Point,T]` interface permitting assignment to an array element using the syntax `A(p) = v`.

X10 also supports dependent types for arrays, e.g., `Array[Double]{rank==3}` is the type of all arrays of `Double` of rank `3`. The `Array` class has distribution, region, and rank properties. X10 v2.0 defines type definitions that allows a distribution, region, or rank to be specified with on the array type.

```
package x10.lang;
type Array[T](n: Int) = Array[T]{rank==n};
type Array[T](d: Dist) = Array[T]{dist==d};
type Array[T](r: Region) = Array[T]{region==r};
```

16.0.2 Rails

A *rail* is a one-dimensional, zero-based, local array. It is more primitive than the `Array` class. Rails are indexed by integers rather than multi-dimensional points. Rails have a single `length` property of type `Int`. Rails can be mutable or immutable and are defined by the following class definitions:

```
package x10.lang;
public class ValRail[T](length: Int) implements (Int)=>T, Iterable[T] { }
public class Rail[T](length: Int) implements (Int)=>T, Settable[Int,T], Itera
```

X10 supports shorthand syntax for rail construction (§11.26).

16.1 Points

Arrays are indexed by points–n-dimensional tuples of integers, implemented by the class `x10.lang.Point`. X10 specifies a simple syntax for the construction of points. A rail constructor (§11.26) of type `ValRail[Int]` of length n can be implicitly coerced to a `Point` of rank n. For example, the following code initializes `p` to a point of rank two using a rail constructor:

CHECK: *Huh? Why do we say rail constructor above? As opposed to just any old valrail?*

```
p: Point(2) = [1,2];
```

The `Point` factory method `make` can take a rail constructor as argument. The assignment above can be written, without implicit coercion, as:

```
p: Point = Point.make([1,2]);
```

Points implement the function type `(Int) => Int`; thus, the `i`th element of a point `p` may be accessed as `p(i)`.

16.2 Regions

A region is a set of points. X10 provides a built-in class, `x10.lang.Region`, to allow the creation of new regions and to perform operations on regions.

Each region `R` has a constant integer rank, `R.rank`.

Here are several examples of region declarations:

```
val MAX_HEIGHT=20;
val Null = Region.makeUnit();  // Empty 0-dimensional region
val N = 10;
val K = 2;
val R1 = 1..100; // 1-dim region with extent 1..100
val R2 = [1..100] as Region(1); // same as R1
val R3 = (0..99) * (-1..MAX_HEIGHT);
val R4 = [0..99, -1..MAX_HEIGHT] as Region(2); // same as R3
val R5 = Region.makeUpperTriangular(N);
val R7 = R4 && R5; // intersection of two regions
val R8 = R4 || R5; // union of two regions
```

The expression $\mathtt{a}_1$`..`$\mathtt{a}_2$ is shorthand for the rectangular, rank-1 region consisting of the points $\{$`[`$\mathtt{a}_1$`]`, ..., `[`$\mathtt{a}_2$`]`$\}$. Each subexpression of $\mathtt{a}_i$ must be of type `Int`. If $\mathtt{a}_1$ is greater than $\mathtt{a}_2$, the region is empty.

A region may be constructed by converting from a rail of regions or a rail of points, typically using a rail constructor (§11.26) (e.g., `R4` above). The region constructed from a rail of points represents the region containing just those points. The region constructed from a rail of regions represents the Cartesian product of each of the arguments. X10 v2.0 does not (yet) support hierarchical regions.

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

- `Region.makeUpperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular `N x N` matrix.
- `Region.makeLowerTriangular(N)` returns a region corresponding to the non-zero indices in a lower-triangular `N x N` matrix.

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of a region `R=(1..2)*(1..2)` are ordered as

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

Sequential iteration statements such as `for` (§12.12) iterate over the points in a region in the canonical order.

A region is said to be *rectangular* if it is of the form $(\mathtt{T}_1 \ \mathtt{*} \ \cdots \ \mathtt{*} \ \mathtt{T}_k)$ for some set of regions $\mathtt{T}_i$. 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 (that is, it is *convex*). (Note that `||` may produce non-convex regions from convex regions, e.g., `[1,1] || [3,3]` is a non-convex region. The operation `R.boundingBox()` gives the smallest rectangular region containing `R`.)

16.2.1 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 $\mathtt{R}_1$ and $\mathtt{R}_2$ be two regions whose type establishes that they are of the same rank. Let `S` be a region of unrelated rank.

$\mathtt{R}_1$ && $\mathtt{R}_2$ is the intersection of $\mathtt{R}_1$ and $\mathtt{R}_2$.

$\mathtt{R}_1$ `||` $\mathtt{R}_2$ is the union of the $\mathtt{R}_1$ and $\mathtt{R}_2$.

$\mathtt{R}_1$ `-` $\mathtt{R}_2$ is the set difference of $\mathtt{R}_1$ and $\mathtt{R}_2$.

$\mathtt{R}_1$ `*` `S` is the Cartesian product of $\mathtt{R}_1$ and `S`, formed by pairing each point in $\mathtt{R}_1$ with every the point in `S`. Thus, `([1..2,3..4] as Region 2) * (5..6)` is the region of rank `3` containing the points `(x,y,z)` where `x` is `1` or `2`, `y` is `3` or `4`, and `z` is `5` or `6`.

For a region `R` and point `p` of the same rank `R+p` and `R-p` represent the translation of the region with `p`. That is, point `q` is in `R` if and only if point `q+p` is in `R+p`. (And similarly for `R-p`.)

For more details on the available methods on `Region`, please consult the API documentation.

16.3 Distributions

A *distribution* is a mapping from a region to a set of places. X10 provides a built-in class, `x10.lang.Dist`, to allow the creation of new distributions and to perform operations on distributions. This class is `final` in X10 v2.0; future versions of the language may permit user-definable distributions.

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

```
R: Region = 1..100;
D: Dist = Dist.makeBlock(R);
D: Dist = Dist.makeCyclic(R);
D: Dist = R -> here;
D: Dist = Dist.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`.

16.3.1 Operations returning distributions

Let `R` be a region, `Q` a set of places {`p1`, ..., `pk`} (enumerated in canonical order), and `P` a place.

Unique distribution The distribution `Dist.makeUnique(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`, as does `Dist.makeConstant(R,P)`.

Block distributions. The distribution `Dist.makeBlock(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 `Dist.makeBlock(R)` is the same distribution as `Dist.makeBlock(R, Place.places)`.

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

The distribution `Dist.makeCyclic(R)` is the same distribution as `Dist.makeCyclic(R, Place.places)`.

Thus the distribution `Dist.makeCyclic(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 `Dist.makeCyclic(Place.places)`.

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

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

Domain Restriction. If `D` is a distribution and `R` is a sub-region of `D.region`, 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.region`.

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 region 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.

16.3.2 User-defined distributions

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

16.3.3 Operations on distributions

A *sub-distribution* of D is any distribution E defined on some subset of the region of D, which agrees with D on all points in its region. We also say that D is a *super-distribution* of E. A distribution D_1 *is larger than* D_2 if D_1 is a super-distribution of D_2.

Let D_1 and D_2 be two distributions.

Intersection of distributions. D_1 && D_2, the intersection of D_1 and D_2, is the largest common sub-distribution of D_1 and D_2.

Asymmetric union of distributions. D_1.overlay(D_2), the asymmetric union of D_1 and D_2, is the distribution whose region is the union of the regions of D_1 and D_2, and whose value at each point p in its region is D_2(p) if p lies in D_2.region otherwise it is D_1(p). (D_1 provides the defaults.)

Disjoint union of distributions. D_1 || D_2, the disjoint union of D_1 and D_2, is defined only if the regions of D_1 and D_2 are disjoint. Its value is D_1.overlay(D_2) (or equivalently D_2.overlay(D_1). (It is the least super-distribution of D_1 and D_2.)

Difference of distributions. D_1 - D_2 is the largest sub-distribution of D_1 whose region is disjoint from that of D_2.

16.3.4 Example

```
def dotProduct(a: Array[T](D), b: Array[T](D)): Array[Double](D) =
  (new Array[T]([1:D.places],
      (Point) => (new Array[T](D | here,
                     (i): Point) => 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.

16.4 Array initializer

Arrays are instantiated by invoking one of the make factory methods of the Array class.

An array creation must take either an Int as an argument or a Dist. 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 function. 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:

```
val data : Array[Int]
    = Array.make[Int](1..1000->here, ((i):Point) => i);
val data2 : Array[Int]
    = Array.make[Int]([1..1000,1..1000]->here, ((i,j):Point) => 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 a reference to a mutable 2-d array over [1..1000, 1..1000] initialized with i*j at point [i,j].

Other examples:

```
val D1:Dist(1) = ...; /* An expression that creates a Dist */
val D2:Dist(2) = ...; /* An expression that creates a Dist */

val data : Array[Int]
    = Array.make[Int](1000, ((i):Point) => i*i);

val data2 : Array[Float]
```

```
      = Array.make[Float](D1, ((i):Point) => i*i as Float);

   val result : Array[Float]
     = Array.make[Float](D2, ((i,j):Point) => i+j as Float);;
```

16.5 Operations on arrays

In the following let `a` be an array with distribution `D` and base type `T`.

16.5.1 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.

For array variables, the right-hand-side of an assignment must have the same distribution `D` as an array being assigned. This assignment 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`.

16.5.2 Constant promotion

For a distribution `D` and a val `v` of type `T` the expression `new Array[T](D, (p: Point) => v)` denotes the mutable array with distribution `D` and base type `T` initialized with `v` at every point.

16.5.3 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 (§16.3) to obtain sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

16.5.4 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)`.

16.5.5 Global operations

Pointwise operations The unary `lift` operation applies a function to each element of an array, returning a new array with the same distribution. The `lift` operation is implemented by the following method in `Array[T]`:

```
def lift[S](f: (T) => S): Array[S](dist);
```

The binary `lift` operation takes a binary function and another array over the same distribution and applies the function pointwise to corresponding elements of the two arrays, returning a new array with the same distribution. The `lift` operation is implemented by the following method in `Array[T]`:

```
def lift[S,R](f: (T,S) => R, Array[S](dist)): Array[R](dist);
```

Reductions Let `f` be a function of type `(T,T)=>T`. Let `a` be a value or reference array over base type `T`. Let `unit` be a value of type `T`. Then the operation `a.reduce(f, unit)` returns a value of type `T` obtained by performing `f` on all points in `a` in some order, and in parallel. The function `f` must be associative and commutative. The value `unit` should satisfy `f(unit,x)` == `x` == `f(x,unit)`.

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 `f` 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||f()` returns an array of base type `T` and distribution `D` whose ith element (in canonical order) is obtained by performing the reduction `f` 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.Array` and other related classes.

17 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.

17.1 Annotation syntax

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

Annotation ::= `@` *InterfaceBaseType Constraints*$^?$

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.

$$
\begin{array}{rcl}
\textit{ClassModifier} & ::= & \textit{Annotation} \\
\textit{InterfaceModifier} & ::= & \textit{Annotation} \\
\textit{FieldModifier} & ::= & \textit{Annotation} \\
\textit{MethodModifier} & ::= & \textit{Annotation} \\
\textit{VariableModifier} & ::= & \textit{Annotation} \\
\textit{ConstructorModifier} & ::= & \textit{Annotation} \\
\textit{AbstractMethodModifier} & ::= & \textit{Annotation} \\
\textit{ConstantModifier} & ::= & \textit{Annotation} \\
\textit{Type} & ::= & \textit{AnnotatedType} \\
\textit{AnnotatedType} & ::= & \textit{Annotation}^{+}\ \textit{Type} \\
\textit{Statement} & ::= & \textit{AnnotatedStatement} \\
\textit{AnnotatedStatement} & ::= & \textit{Annotation}^{+}\ \textit{Statement} \\
\textit{Expression} & ::= & \textit{AnnotatedExpression} \\
\textit{AnnotatedExpression} & ::= & \textit{Annotation}^{+}\ \textit{Expression}
\end{array}
$$

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 def get(i: Int): Object { ... }

// constructor annotation
@Where(x != null)
def this(x: T) { ... }

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

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

- Type annotations:

```
List@Nonempty

Int@Range(1,4)

Array[Array[Double]]@Size(n * n)
```

- Expression annotations:

```
m() : @RemoteCall
```

- Statement annotations:

```
@Atomic { ... }

@MinIterations(0)
@MaxIterations(n)
for (var i: Int = 0; i < n; i++) { ... }

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

17.2 Annotation declarations

Annotations are declared as interfaces. They must be subtypes of the interface `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.

17.3 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 [8] 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 `Node`s 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();
```

18 Native Code Integration

At times it becomes necessary to call non-X10 code from X10, perhaps to make use of specialized libraries in other languages or to write more precisely controlled code than X10 generally makes available. The `@Native(lang,code) Phrase` annotation from `x10.compiler.Native` in X10 can be used to tell the X10 compiler to generate `code` for certain kinds of `Phrase`, instead of what it would normally compile to, when compiling to the `lang` back end.

18.1 Native `static` Methods

`static` methods can be given native implementations. Note that these implementations are syntactically *expressions*, not statements, in C++ or Java. Also, it is possible (and common) to provide native implementations into both Java and C++ for the same method.

```
import x10.compiler.Native;
class Son {
  @Native("c++", "printf(\"Hi!\")")
  @Native("java", "System.out.println(\"Hi!\")")
  static def printNatively():Void = {};
}
```

If only some back-end languages are given, the X10 code will be used for the remaining back ends:

```
import x10.compiler.Native;
class Land {
  @Native("c++", "printf(\"Hi from C++!\")")
  static def example():Void = {
```

```
    x10.io.Console.OUT.println("Hi from X10!");
  };
}
```

The `native` modifier on methods indicates that the method must not have an X10 code body, and `@Native` implementations must be given for all back ends:

```
import x10.compiler.Native;
class Plants {
  @Native("c++", "printf(\"Hi!\")")
  @Native("java", "System.out.println(\"Hi!\")")
  static native def printNatively():Void;
}
```

Values may be returned from external code to X10. Scalar types in Java and C++ correspond directly to the analogous types in X10.

```
import x10.compiler.Native;
class Return {
  @Native("c++", "1")
  @Native("java", "1")
  static native def one():Int;
}
```

Parameters may be passed to external code. `(#1)` is the first parameter, `(#2)` the second, and so forth. (`(#0)` is the name of the enclosing class.) Be aware that this is macro substitution rather than normal parameter passing; *e.g.*, if the first actual parameter is `i++`, and `(#1)` appears twice in the external code, `i` will be incremented twice. For example, a (ridiculous) way to print the sum of two numbers is:

```
import x10.compiler.Native;
class Species {
  @Native("c++", "printf(\"Sum=%d\", ((#1)+(#2)) )")
  @Native("java", "System.out.println(\"Hi!\")")
  static native def printNatively(x:Int, y:Int):Void;
}
```

Static variables in the class are available in the external code.

18.2 Native Blocks

Any block may be annotated with `@Native(lang,stmt)`, indicating that, in the given back end, it should be implemented as `stmt`. All value variables from the surrounding context are available inside `stmt`. For example, the method call `born.example(10)`, if compiled to Java, changes the field `y` of a `Born` object to 10. If compiled to C++ (for which there is no `@Native`), it sets it to 3.

```
import x10.compiler.Native;
class Born {
  var y : Int = 1;
  public def example(x:Int):Int{
    @Native("java", "y=x;")
    {y = 3;}
    return y;
  }
}
```

Note that the code being replaced is a statement – the block `{y = 3;}` in this case – so the replacement should also be a statement.

Other X10 constructs may or may not be available in Java and/or C++ code. For example, type variables do not correspond exactly to type variables in either language, and may not be available there. The exact compilation scheme is *not* fully specified. You may inspect the generated Java or C++ code and see how to do specific things, but there is no guarantee that fancy extern coding will continue to work in later versions of X10.

The full facilities of C++ or Java are available in native code blocks. However, there is no guarantee that advanced features behave sensibly. You must follow the exact conventions that the code generator does, or you will get unpredictable results. Furthermore, the code generator's conventions may change without notice or documentation from version to version. In most cases the code should either be a very simple expression, or a method or function call to external code.

18.3 External Java Code

When X10 is compiled to Java, mentioning a Java class name in native code will cause the Java compiler to find it in the sourcepath or classpath, in the usual way.

This requires no particular extra work from the programmer.

18.4 External C++ Code

C++ code can be linked to X10 code, either by writing auxiliary C++ files and adding them with suitable annotations, or by linking libraries.

18.4.1 Auxiliary C++ Files

Auxiliary C++ code can be written in `.h` and `.cc` files, which should be put in the same directory as the the X10 file using them. Connecting with the library uses the `@NativeCPPInclude(dot_h_file_name)` annotation to include the header file, and the `@NativeCPPCompilationUnit(dot_cc_file_name)` annotation to include the C++ code proper. For example:

MyCppCode.h:

```
void foo();
```

MyCppCode.cc:

```
#include <cstdlib>
#include <cstdio>
void foo() {
    printf("Hello World!\n");
}
```

Test.x10:

```
import x10.compiler.Native;
import x10.compiler.NativeCPPInclude;
import x10.compiler.NativeCPPCompilationUnit;

@NativeCPPInclude("MyCPPCode.h")
@NativeCPPCompilationUnit("MyCPPCode.cc")
public class Test {
    public static def main (args:Rail[String]!) {
        { @Native("c++","foo();") {} }
    }
}
```

18.4.2 C++ System Libraries

If we want to additionally link to more libraries in `/usr/lib` for example, it is necessary to adjust the post-compilation directly. The post-compilation is the compilation of the C++ which the X10-to-C++ compiler `x10c++` produces.

The mechanism used for this is the `-post` command line parameter to `x10c++`. The following example shows how to compile `blas` into the executable via post compiler parameters.

```
x10c++ Test.x10 -post '# # -I /usr/local/blas # -L /usr/local/blas -lblas'
```

- The first `#` means to use the default compiler for the architecture (from x10rt properties file).
- The second `#` is substituted for the .cc files and CXXFLAGS that would ordinarily be used.
- The third `#` is substituted for the libraries and LDFLAGS that would ordinarily used.
- For the second and third, if a `%` is used instead of a `#` then the the substitution does not occur in that position. The `%` is erased. The desired parameter value should appear after the `%` on the line. This allows a complete override of the postcompiler behaviour.

19 Lost Bits

19.1 Visibility of Local Variables and Formals

In general, variables (*i.e.*, local variables, parameters, properties, fields) are visible at a point in the `c` 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). A formal parameter is visible in the types of all other formal parameters of the same method, constructor, or type definition, as well as in the method or constructor body itself. Properties are accessible via their containing object–`this` within the body of their class declaration. The special variable `this` is in scope at each property declaration, constructor signatures and bodies, instance method signatures and bodies, and instance field signatures and initializers, but not in scope at `static` method or field declarations or `static` initializers.

References

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

[2] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, 1997.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in LNCS, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[9] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367, London, UK, 2005. Springer-Verlag.

[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.

A Change Log

A.1 Changes from X10 v2.0

- `Any` is now the top of the type hierarchy (every object, struct and function has a type that is a subtype of `Any`). `Any` defines `home`, `at`, `toString`, `typeName`, `equals` and `hashCode`. `Any` also defines the methods of `Equals`, so `Equals` is not needed any more.
- Revised discussion of incomplete types (§8.8.5).
- The manual has been revised and brought into line with the current implementation.

A.2 Changes from X10 v1.7

The language has changed in the following way:

- **Type system changes**: There are now three kinds of entities in an X10 computation: objects, structs and functions. Their associated types are class types, struct types and function types.

  Class and struct types are called *container types* in that they specify a collection of fields and methods. Container types have a name and a signature (the collection of members accessible on that type). Collection types support primitive equality == and may support user-defined equality if they implement the `x10.lang.Equals` interface.

  Container types (and interface types) may be further qualified with constraints.

A function type specifies a set of arguments and their type, the result type, and (optionally) a guard. A function application type-checks if the arguments are of the given type and the guard is satisfied, and the return value is of the given type. A function type does not permit == checks. Closure literals create instances of the corresponding function type.

Container types may implement interfaces and zero or more function types.

All types support a basic set of operations that return a string representation, a type name, and specify the home place of the entity.

The type system is not unitary. However, any type may be used to instantiate a generic type.

There is no longer any notion of `value` classes. `value` classes must be re-written into structs or (reference) classes.

- **Global object model**: Objects are instances of classes. Each object is associated with a globally unique identifier. Two objects are considered identical == if their ids are identical. Classes may specify `global` fields and methods. These can be accessed at any place. (`global` fields must be immutable.)

- **Proto types.** For the decidability of dependent type checking it is necessary that the property graph is acyclic. This is ensured by enforcing rules on the leakage of `this` in constructors. The rules are flexible enough to permit cycles to be created with normal fields, but not with properties.

- Place types. Place types are now implemented. This means that non-global methods can be invoked on a variable, only if the variable's type is either a struct type or a function type, or a class type whose constraint specifies that the object is located in the current place.

  There is still no support for statically checking array access bounds, or performing place checks on array accesses.

The X10 language has been developed as part of the IBM PERCS Project, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.