1 Changes

This document summarizes the main changes between X10 2.0.6 and X10 2.1. The descriptions are intended to be suggestive rather than definitive; see the language specification – when it is finished – for full details.

1.1 Object Model

1. Objects are now local rather than global.

   (a) The `home` property is gone.

   (b) `at(P)S` produces deep copies of all objects located `here` when it executes `S`. (**Warning:** They are copied even in `at(here)S`.)

2. The `GlobalRef[T]` struct is the only way to produce or manipulate cross-place references.

   (a) `GlobalRef`'s have a `home` property.

   (b) Use `GlobalRef[Foo](foo)` to make a new global reference.

   (c) Use `myGlobalRef()` to access the object referenced; this requires `here == myGlobalRef.home`.

3. All those cursed `!`s in types are gone.

4. `global` modifiers are now gone:

   (a) `global` methods in *interfaces* are now the default.

(b) `global` *fields* are gone. In some cases object copying will produce the same effect as global fields. In other cases code must be rewritten. It may be desirable to mark nonglobal fields `transient` in many cases.

(c) `global` *methods* are now marked `@Global` instead. Methods intended to be non-global may be marked `@Pinned`.

1.2 Constructors

1. `proto` types are gone.

2. Constructors and the methods they call must satisfy a number of static checks.

   (a) Constructors can only invoke `private` or `final` methods, or methods annotated `@NonEscaping("v1,v2")`.

   (b) Methods invoked by constructors cannot read fields before they are written.

   (c) The compiler ensures this with a detailed protocol.

3. It is still impossible for X10 constructors to leak references to `this` or observe uninitialized fields of an object. Now, however, the mechanisms enforcing this are less obtrusive than in 2.0.6; the burden is largely on the compiler, not the programmer.

1.3 Call by Reference

A very limited form of call-by-reference is now available.

1. Formal parameters to functions and methods may be `ref` rather than `var` or `val`.

2. Assignment to a `ref` parameter `x` changes the original location that the `ref` refers to. *e.g.*, `def inc(ref x:Int) { x ++; }` allows a call `inc(n)` to increment a local `var n`.

3. Only local variables or `ref` parameters can be passed as actual `ref` parameters. Fields, array elements, and other variable-like items cannot be.

4. External `ref` variables cannot be captured in closures. However, closures may have `ref` parameters.

5. `ref`s are *not* first-class objects in X10. They cannot be returned from functions, stored in data structures, etc.

6. These restrictions limit the possibilities of aliasing and the need for boxing of `ref` parameters. `ref`s to stack locations cannot, with these restrictions, live past the death of the location's containing stack frame.

7. This allows the implementation of many core constructs as syntactic sugar on library calls. Programmers may use it, but mutability should generally be encapsulated inside objects rather than `ref` parameters.

1.4 Accumulator Variables

Accumulator variables generalize and make explicit collecting `finish` in X10 2.0.6. An `acc` variable is declared:

```
acc(r) A;
```

where `r` is a *reducer* (much as in 2.0.6):

```
struct Reducer[T](zero:T, apply:global (T,T)=>T){}
```

Usage of `A` is restricted in ways that make it determinate in the intended case of a pure, associative, commutative `apply` with unit element `zero`.

1. `A` is initialized to `r.zero`.

2. Multiple activities can *write* into `A`. In particular, the "assignment" `A = v` is approximately interpreted as `atomic{A = r.apply(A, v)}` — that is, it accumulates `v` into `A` using `r.apply`.

3. *Reading* of `A` is restricted to situations where it makes sense. Specifically, only the activity in which `A` is declared can read from it, and it can only do so when all asyncs which it has spawned have terminated – *e.g.*, outside of the scope of all `async`s and `finish`es.

4. Formal parameters of functions may be marked `acc x:T`. The reducer `r` must not be specified; it is passed as an implicit parameter going with the actual `acc` variable.

5. X10 provides protocols for indexed collections of `acc` variables, presented as objects.

1.5 Implicit clocks for each finish

Clocks are no longer explicit objects. Many clock operations are available using implicit clocks.

1. A `finish` may be qualified with `clocked`, which gives it a clock.

2. An `async` in a `clocked finish` may be marked `clocked`. This registers it on the same clock as the enclosing `finish`.

3. `clocked async S` and `clocked finish S` may use `next` in the body of `S` to advance the clock.

4. When the body of a `clocked finish` completes, the `clocked finish` is dropped form the clock. It will still wait for spawned asyncs to terminate, but such asyncs need to wait for it.

1.6 Clocked local variables

Local `val` and `acc` variables may be `clocked`. They are associated with the clock of the surrounding `clocked finish`. Clocked variables have a *current* value and an *upcoming* value. The current value may be read at suitable times; the upcoming value may be updated. The `next` phase makes the upcoming value current.

1.7 Asynchronous initialization of val

`val`s can be initialized asynchronously. As always with `val`s, they can only be read after is guaranteed that they have been initialized. For example, both of the

prints below are good. However, the commented-out print in the async is bad, since it is possible that it will be executed before the initialization of a.

```
val a:Int;
finish {
  async {
     a = 1;
     print("a=" + a);
  }
  async {
     // WRONG: print("a=" + a);
  }
}
print("a=" + a);
```

1.8 Main Method

The signature for the main method is now:

```
def main(Array[String](1)) {..}
```

or, if the arguments are actually used,

```
def main(argv: Array[String](1)) {..}
```

1.9 Removed Topics

The following are gone:

1. x10.lang.Clock.
2. clocked clause on async, ateach, and foreach.
3. foreach is gone.
4. All vars are effectively shared, so shared is gone.
5. collecting finish, offer, and offers are gone. Use acc variables instead.

6. The place clause on `async` is gone. `async (P) S` should be written `at(P) async S`.

1.10 Assorted Additions

1. structs can have `var` fields. This goes with call-by-reference.

2. Statements can be annotated `@det`; the compiler will check them for determinacy.