Unit: Blocking Synchronization
Clocks, v0.3

Vijay Saraswat

This lecture discusses X10 clocks. For reference material please look at the chapter on Clocks in [].

1 Motivation

The central idea underlying clocks is that of *phased computations*.

1. Consider red/black iterations, e.g. to implement stencil computations. Each iteration may be considered a (determinate) phase. In each phase a well-defined set of locations is read and written (determinately) by concurrent activities. The end of a phase is detected by termination of these activities. In the next phase, a new set of activities are created.

   Typically in a phase information produced in the previous phase is read.

2. Consider streaming computations. A stream is a sequence of values. A stream is produced by a *source* and consumed by a *sink*. The values produced by the source are consumed by the sink in FIFO (First-In First-Out) order. A computing node typically has one or more incoming streams (that it reads from), and one or more outgoing streams (that it writes into).

   A stream may have finite capacity, e.g. it may hold only two values (e.g. in a circular buffer). In each phase, the producer can write a value, and the consumer can read (the previous) value. Thus computation can progress indefinitely in (determinate) phases.

2 Definition

Clocks in X10 are instances of the class `x10.lang.Clock`. While clocks may be created explicitly and managed by user code, there are several restrictions on their usage. Hence X10 supports some constructs that implicitly use clocks, there are few restrictions on these constructs.

A `finish S` statement may optionally be prefixed by `clock`: such a statement is called (naturally enough!) a *clocked finish* statement. It is associated with an (anonymous) clock. Execution of a `clocked async S` construct in the dynamic scope of such a finish *registers* the generated activity with the implicit clock. A `clocked async` A must occur (dynamically) nested in a `clocked async` or `clocked finish`. If it is enclosed in an unclocked `async` (which does not itself enclose the controlling `clocked finish` for A), a runtime exception is thrown.

A `clocked async` may invoke a `Clock.advanceAll()` method. This method call returns only when all activities registered on the implicit clock have themselves invoked `Clock.advanceAll()`. At thi point, we say that the clock can *advance its phase*: on advancing, it releases all asyncs waiting in their `Clock.advanceAll()` call, and these calls return. Thus clocks can be used to get *barrier* functionality: the barrier is raised only when all activities registered on it have arrived at it.

An important detail: The activity executing `clocked finish S` is automatically registered on the implicit clock created when `S` is executed. The activity is implicitly deregistered from this clock when `S` locally terminates. (Note: termination of `S`, as usual, does not require that all activities spawned by `S` have terminated.)

Example 2.1 *Conside the following simple code:*

```
var x:Int=0, y:Int=0;
clocked finish {
  clocked async {
    x=1;
    Clock.advanceAll();
    Console.OUT.println("y=" + y); // must print 1
  }
  clocked async {
    y=1;
    Clock.advanceAll();
    Console.OUT.println("x=" + x); // must print 1
  }
}
```

Exercise 2.1 *Work out how to translate* `finish S` *into a piece of code that does not use* `finish` *but uses clocks instead. Note that* `S` *may recursively contain* `finish S1` *statements.*

For simplicity, assume that `S` *may not itself contain clocked operations.*

Whenever you write code in which a `finish ... async...` pattern is used within a loop you should consider rewriting the code to use clocks. The basic idea is to spawn activities only once, outside the loop, and use a `next` operation in the body of the activity to signal completion of the phase. That is, translate:

```
for (p in R)
  finish
    for (q in S)
      async
          S // p and q may occur here
```

into

```
clocked finish
  for (q in S)
    clocked async
      for (p in R) {
        S; // p and q may occur here
        Clock.advanceAll();
      }
```

2.1 Determinacy

The clock operations do not introduce any indeterminacy – that is, there is no race condition involved in their operation.

The operation of a clock can be modeled by considering a clock to be associated with two counters `N` (the number of activities registered with the clock in the current phase), and `Q` (the number of activities that have entired a `advanceAll` call in the current phase). (One may optionally keep track of the current `phase`, an `Int` or `Long`, this is incremented each time the barrier is lifted.)

When a `clocked finish` is spawend, a new clock is created, and associated with the count `(1,0)`, since the current activity is automatically registered with the clock, and no activity has quiesced yet.

When an activity enters `advanceAll()`, the count makes a transition from `(N,Q)` to `(N,Q+1)`. Note that for this to happen it must be the case that `N > Q` – this call can only be made legally (i.e. without an exception being thrown) by a `clocked async` that is not already in the middle of a `advanceAll()` call.

$$\frac{N > Q, N > 0}{(N,Q) \longrightarrow (N,Q+1)}$$

When a `clocked async` is spawned the counts associated with the `finish` transition from `(N,Q)` to `(N+1,Q)`. GIven the restriction above (a `clocked async` can be spawned only by another `clocked async` or the activity that created the `clocked finish`), it is clear that this step can only happen if the condition `N > 0` is true.

$$\frac{N > Q, N > 0}{(N,Q) \longrightarrow (N+1,Q)}$$

When a `clocked async` terminates, the counts transition from `(N,Q)` to `(N-1,Q)`.

$$\frac{N > Q, N > 0}{(N,Q) \longrightarrow (N-1,Q)}$$

When the activity that launched the `clocked finish S` finishes executing `S`, the counts transition from `(N,Q)` to `(N-1,Q)`.

$$\frac{N > Q, N > 0}{(N,Q) \longrightarrow (N-1,Q)}$$

Given these operations on the clock, the condition `N==Q` is *stable* – none of these transitions can occur. Note that in this state no assertion can be made about the value of `N`, e.g. it may be zero.

Exercise 2.2 *Write a small snippet of code that shows how the state `(0,0)` can be reached.*

Exercise 2.3 *For every `N > 0` sketch out a snippet of code that could cause the clock to reach the state `(N,N)`.*

Once this state is reached, the clock detects the barrier is reached, increments the phase counter, and changes the count from (N,Q) to $(N,0)$, since it has released all the clocked asyncs from the barrier.

Because the condition $N == Q$ is stable for the clock, clock operations are determinate. There is no race condition between an activity trying to register itself on the clock, and the clock wanting to make a phase transition.

Note that multiple activities registered on the clock *are* racing in their operations: they are simultaneously mutating the state of the clock in ways that are not locally commutable. However, the key point is that no activity is able to actually *read* the values of `N` and `Q` – they can only detect when $N == Q$.

2.1.1 Application level determinacy

Can applications can make use of clocks to be determinate?

Yes! Suppose a location needs to be written and read by multiple concurrent asyncs. Now arrange for it to be the case that the readers and writers are registered on the same clock. Ensure that in a given phase at most one async will write into the location. Hence there will be no write-write conflicts. Further, ensure that in a given phase activities only read that location or write the location. This ensures that there are no read/write conflicts. Thus clocks help the programmer to segregate operations on controlled locations into non-overlapping phases.

If you combine this idea with the red/black idea, you can get a notion of clocked data structures that can be safely (determinately) read and written in each cycle. The read will return the value of the location that was last written into this location – before the current phase. The value (if any) written in the current phase will be visible only in subsequent phases. Hence there are no read/write conflicts.

2.2 Deadlock-freedom

Code using the implicit clock cannot introduce deadlocks. Why?

First let us see that just clock operations cannot introduce a deadlock. Note that starting any state in which $N \neq Q$ there are a sequence of possible transitions that take the clock to $N == Q$. So the system can never be "stuck" in an $N \neq Q$ state.

Finally we have ensured that there are no bad interactions between the `finish` construct and clocks by ensuring that when the activity that is executing `clocked finish S` finishes executing `S` it drops the clock.

3 Examples

3.1 Red black computations using clocks

Here is a version of `AllReduce` that uses a clock. `P` activites are spawned once and for all, and the activities use `next` to move from one phase of the computation to the next.

```
public class ClockedAllReduce {
  static def even(p:int):Boolean = p % 2 == 0;
  public static def allReduce(red:DistArray[int](1),
                          black:DistArray[int](1)) {
    val P = Place.MAX_PLACES;
    val phases = Utils.log2(P);
    clocked finish {
        for ([p] in red) at (red.dist(p)) clocked async {
            var shift_:Int=1;
            for (phase in 0..(phases-1)) {
                val ev = even(phase);
                val destId = (p+shift_)% P;
                val source = here;
                val elem = ev ? black(p) : red(p);
                at(Place(destId)) {
                    if (ev)
                        red(destId) = elem + black(destId);
                    else
                        black(destId) = elem + red(destId);
                }
                shift_ *=2;
                Clock.advanceAll();
            }
        }
    }
    return (even(phases-1)) ? red(0) : black(0);
  }
  public static def main(Rail[String]) {
    assert Utils.powerOf2(Place.MAX_PLACES)
        : " Must run on power of 2 places.";
    val D = Dist.makeUnique();
    val black = DistArray.make[int](D, (p:Point)=> p(0));
    val red = DistArray.make[int](D, (Point)=> 0);
    val result = allReduce(red, black);
    Console.OUT.println("allReduce = " + result);
  }
}
```

3.2 Streaming through single memory locations

Below, we use the `@shared` annotation on mutable variables. A shared variable is permitted to be accessed from within spawned asyncs. It may be used for communication between spawned asyncs, or with the parent activity.

What does the following program do?

```
public class ClockTest {
  static def run() {
    val x=new Cell[Int](1), y= new Cell[Int](1);
      clocked finish  {
        clocked async
          while (true) {
            val r = x();
              Clock.advanceAll();
              y()=r;
              Console.OUT.println("y="+ y);
              Clock.advanceAll();
          }
          while(true) {
            val s = x()+y();
            Clock.advanceAll();
            x() = s;
            Console.OUT.println("x="+ x);
            Clock.advanceAll();
            }
        }
    }
    public static def main(Rail[String]) {
      run();
    }
}
```

It uses two shared mutable variables `x` and `y` (each initialized to `1`). It launches two clocked activities. The first reads `x` in one phase and passes the read value to `y` in the next phase. That is it implements the recurrence

$$y_{i+1} = x_i$$

The second implements the recurrence

$$x_{i+1} = x_i + y_i$$

Hence the sequence of writes to the two locations are:

$$x : 1, 2, 3, 5, 8, 13, 21, ...$$
$$y : 1, 1, 2, 3, 5, 8, 13, ...$$

That is, this program is computing the Fibonacci sequence!

Note that the timing of the writes is crucial for the correctness of this program. Computation progresses indefinitely in a succession of phases. In odd phases the current values of `x` and `y` are read by the two activities. In even phases the first activity writes a new value of `y` and the second the new value of `x`. Thus there is no read/write or write/write conflict.

Exercise 3.1 *Consider the program above, with all `Clock.advanceAll()` calls removed. What can you say about the results that could possibly be printed out?*

Exercise 3.2 *Write a program for computing the following recurrence*

$$A(i,j) = avg(A(i-1,j), A(i-1,j-1), A(i,j-1))$$

assume given boundary conditions (leftmost column, topmost row is independently determined).

3.3 Using clocks explicitly

Clocks can be created explicitly by user code, by invoking `Clock.make()`. The activity creating a clock is said to be *registered* to the clock. An activity may be registered on multiple clocks.

An activity registered on a clock `c` may choose to register activities that it is spawning on `c`. It does that using a `clocked(c)` clause. This clause may be specified for other `async` creating constructs such as `foreach` and `ateach` as well. An `async` created using a `clocked(c1,..., cn)` clause is registered on all these clocks.

Once an activity clocked on a clock `c` has finished performing the actions associated with the current clock phase, it executes a `c.advance()` operation. This operation does two things:

1. It *signals* to the clock that this activity has reached the end of the current phase.
2. It *waits* for the clock to progress.

A clock progresses only when all activities registered on the clock have signalled that they have reached the end of the current phase.

At any time an activity can declare that it does not wish to participate in the phases associated with the clock by *dropping* the clock `c.drop()`. The activity is said to have deregistered from the clock. On termination, an activity automatically deregisters from all the clocks it is registered on.

X10 also supports *split-phase* operations on clocks. An activity may execute a `c.resume()` operation. This signals to the clock that the activity has reached the end of the current phase; but the activity does not wait for the clock to progress. Thus the activity may continue executing other code that is not to be considered part of this phase. We say that the activity has become *quiescent* on `c`. While it is quiescent, it may not drop `c` and may not transmit it to new activities (the *Live Clocks Condition, LCC*). It becomes live again by executing `c.advance()`. This `advance` operation does not return until the clock has progressed to the next phase. Note that the clock may already have moved to the next phase, hence the `advance` operation may return immediately.

Clocks may be passed as arguments to method invocations, returned from methods, stored in local variables or object fields – there are no restrictions. However, there is no operation provided in the language to permit an activity to register itself on a clock that it reads from a variable: the only way that an activity can register on a clock is by creating the clock or being created by an activity that is already registered on the clock. We say that *clocks are transmitted only through the spawn-tree.*

Thus clocks may be thought of as implementing the idea of *barriers* – an activity executing `c.advance()` is said to have reached the barrier, and cannot progress until all other activities registered on `c` have reached that barrier. Then they can *all* progress.

Semantically clocks are closely related to `finish`. An activity spawned during the execution of a `finish` is automatically "registered" with the `finish` (unless it is spawned within the context of a `finish` nested inside the original `finish`). With clocks the programmer has the choice of specifying whether a spawned activity should be registered on the clock or not. Finish detects the termination of a phase by detecting that all activities spawned during the execution of its body have *terminated.* A clock detects the termination of a phase by receiving explicit `advance` or `resume` signals from all activities registered on the clock.

3.4 Restrictions

Clock usage is subject to two restrictions:

Live Clock Condition, LCC An activity that is quiescent on a clock cannot use a clock (transmit it to a spawned `async`, drop it).

Old Clock Restriction When the statement `S` in a `finish S` terminates, the current activity must no be registered on a clock.

(Note that this condition can be established by ensuring that when an activity enters a `finish`, it is not registered on a clock and during the execution of a finish it does not create a clock.)

These restrictions are checked at runtime. If a violation is detected, a `ClockUseException` is thrown. There has been work by Olivier Tardieu, Nalini Vasudevan, and Julian Dolby to check these conditions statically [VTDE09].

3.5 Determinate advancement

LCC is necessary to ensure that clock advancement is determinate. That is, there is no race condition between an activity getting registered on the clock, and the clock advancing to the next phase.

To see this, notice that if at any time during execution a clock is ready to advance, it will continue to be ready to advance until it actually advances. We say that clock quiescence is stable [SJ05, Theorem 5]. [1] More precisely, once a clock is ready to advance, no activity can be registered on the clock until it has advanced. Conversely, an activity can be registered on a clock only when the clock is not ready to advance.

Clearly, if an activity is not quiescent on a clock, the clock is not ready to advance, and it is ok for the activity to register additional activities on the clock. We must now establish that if a clock is ready to advance then no activity can be registered on the clock (until the clock has advanced).

Hence all we need to do to ensure the stability property is to establish that there is no way by which an activity can "spontaneously" be registered on a clock. That is, there is no way by which (a) an activity that is not registered on the clock can cause an activity to be registered on the clock , and (b) an activity that is quiescent on the clock can cause an activity to be registered on the clock.

Possibility (a) is ruled out because clocks are transmitted only through the inheritance tree (see above). Possibility (b) is ruled out because of the Quiescence Restriction.

Thus clock advancement is determinate.

3.6 Deadlock-freedom

The Old Clock restriction is necessary for deadlock-freedom. `X10` has the property that any program written using `finish`, `async`, `atomic` and clocks (that uses only `Clock.advanceAll()` can never deadlock. This theorem was stated in [SJ05] and a detailed proof can be found in [LP10].

First, let us discuss how deadlock could possibly arise. To have deadlock there must be a cycle in the *wait-for* graph. The wait-for graph can be defined in many ways. For our current purposes let us define it as a graph whose nodes are activities and whose edges $a \to b$ indicate that activities `a` and `b` are stuck

[1] A property is said to be stable if once it holds it continues to hold as the system evolves.

and that for `a` to progress `b` must progress. [2] Clearly there is a deadlock in a computation if and only if there is a cycle in this graph.

To see how wait-for graphs can be used, let us establish that an X10 program with `finish`, `async` and `atomic` cannot deadlock. Clearly, an activity whose next statement is an `async` is not stuck, hence it cannot be the target of a wait-for edge. Similarly an activity whose next statement is an `atomic` cannot be stuck – an `atomic` statement can always be executed. [3]

So that brings us to a `finish S`. An activity `A2` that has finished executing the statement `S` in a `finish S` is indeed waiting for the activities spawned during `S` to terminate. However, note that `finish` has a hierarchical structure: it can wait only on the activities it has spawned to terminate. That is, if there is a wait-for edge `A1` $\rightarrow$ `A2` (because `A1` and `A2` are executing `finish` statements and waiting at the end of the statement) then it must be the case that `A1` has spawned `A2`. However the "`A` spawns `B`" relation is a tree – hence it can have no cycles.

Therefore a wait-for graph cannot have a cycle involving only activities executing `finish` statements.

So a clock must be involved in order to create a deadlock.

Example 3.1 *Does this program deadlock?*

```
  finish async { // A1
    val c = Clock.make();
    val d = Clock.make();
    async clocked(c,d) { // A2
        c.advance();
        d.advance();
    }
    async clocked(c,d) { // A3
        d.advance();
        n.advance();
    }
}
```

Indeed it does! `A1` *spawns two activities, each registered on the clock* `c` *and* `d`*. It then terminates. Now* `A2` *will peform a* `advance` *on* `c` *and* `A3` *on* `d`*, leading to deadlock.* `A2` *will wait for* `c` *to advance – but to advance* `c` *requires* `A3` *to perform a* `c.advance()`*. However,* `A3` *cannot do so for a symmetric reason – it is stuck at* `d.advance()` *and needs* `A2` *to execute* `d.advance()` *in order for* `d` *to advance.*

In brief, once `A2` *and* `A3` *reach their first* `advance` *statement, the wait-for graph contains the edges* `A1` $\rightarrow$ `A2` *and* `A2` $\rightarrow$ `A1` *— a cycle!*

If only conjunctive clocks are permitted, then the above situation cannot arise. The programmer would be forced to write:

[2] We are deliberately being a bit informal about these concepts, for the sake of ease of exposition.

[3] Note that this is not true for a `when (c) S` statement. An activity executing `when (c) S` is indeed stuck as long as `c` is not true.

```
finish async { // A1
  val c = Clock.make();
  val d = Clock.make();
  async clocked(c,d) { // A2
      Clock.advanceAll();
  }
  async clocked(c,d) { // A3
    Clock.advanceAll();
  }
}
```

Now when `A2` reaches `advanceAll`, it signals both `c` and `d`, and waits for both of them to advance. `A3` does the same. Since all clocks registered on `c` have reached the barrier, `c` can advance. So can `d`. Hence both `A2` and `A3` can advance.

Thus by restricting ourselves to conjunctive clocks, we can ensure that deadlocks do not arise because some subset of activities registered on a clock `c` are stuck at an `advance` on a different clock `d`, and vice versa.

So to get a deadlock we must have the case that an activity is suspended on something other than an `advanceAll`. The only other construct in the language that introduces a suspension is `finish`. Can `finish` and `advanceAll` combine to produce deadlock?

Example 3.2 *Consider this example:*

```
// A1
val c = Clock.make();
finish {
 async clocked(c) { // A2
    Clock.advanceAll();
 }
}
```

The program deadlocks. When `A1` *has reached the end of the statement inside the finish, and* `A2` *is executing the* `advanceAll`*, the wait-for graph contains the cycle* `A1` $\rightarrow$ `A2`, `A2` $\rightarrow$ `A1`*.*

Here is another example illustrating deadlock:

```
// A1
finish {
  val c = Clock.make();
  async clocked(c) { // A2
    Clock.advanceAll();
  }
}
```

These programs are ruled out by the Old Clock Restriction. The restriction ensures that when an activity is waiting on a `finish` (for spawned activities to terminate) it is not registered on any clock, hence in the wait-for graph there can be no edge targeted at this activity. Thus this activity cannot be part of a cycle.

A detailed proof of deadlock-freedom may be found in [LP10].

References

[LP10] Jonathan Lee and Jens Palsberg. Featherweight x10: A core calculus for async-finish parallelism. In *Proceedings of PPoPP 2010*, 2010.

[SJ05] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of Concur 2005*, 2005.

[VTDE09] Nalini Vasudevan, Olivier Tardieu, Julian Dolby, and Stephen Edwards. Compile-time analysis and specialization of clocks in concurrent programs. In *Proceedings of Compile Construction*, pages 48–62, 2009.