PPLS: Examples of co Notation

Here are some more examples of the `co` notation in action, to help you understand some of the tricky issues which can arise. As in lectures and the example sheet, we assume atomic reads and writes and a sequentially consistent model of shared memory. Solutions are over the page.

1. Discuss the full range of possible outcomes of this program.

```
int a = 0, b = 4;
co [i = 0 to 4] {
  if (a<b) {
    a++;
  } else {
    b++;
  }
}
```

2. Discuss the full range of possible outcomes of this program (which is the same as question 1, but now with 6 concurrent threads).

```
int a = 0, b = 4;
co [i = 0 to 5] {
  if (a<b) {
    a++;
  } else {
    b++;
  }
}
```

3. Discuss the full range of possible outcomes of this program (which is the same as question 1, i.e. back to 5 concurrent threads, but now with atomic increments).

```
int a = 0, b = 4;
co [i = 0 to 4] {
  if (a<b) {
    <a++;>
  } else {
    <b++;>
  }
}
```

4. Discuss the full range of possible outcomes of this program.

```
int a = 0;
co [i = 0 to 1] {
  if (a<a) {
    a=10;
  } else {
    <a++;>
  }
}
```

5. Discuss the full range of possible outcomes of this program.

```
a=0; flag=0;
co
  {a=25; flag=1;}
//
  <await (flag==1) x=a;>
//
  flag=0;
oc
```

6. Discuss the full range of possible outcomes of this program.

```
x=0;
co
  <await (x>2) x=1;>
//
  x=4;
//
  <await (x>3) x=10;>
oc
```

Solutions

1. The "obvious" outcome is that four threads will see `a<b` and increment a, and the other thread will find `a==b` and increment b, leaving `a==4, b==5`. But what else might happen? Firstly we note that since increments are not atomic, any or all of them could find their reads and writes interleaved with respect to other increments. As we have seen, this can affect `a`. At one extreme, if all the reads of `a` happen before any of the writes, we can finish with `a==1, b==4`. At the other extreme, the intended increments could be "nicely" interleaved, leaving `a==5, b==4`. All values of `a` in between are also possible. In summary, the possible outcomes are `a==1/2/3/4/5, b==4`, or `a==4, b==5`.

2. This program is similar of course, but we could now have up to two threads seeing `a<b` as false. Remarks about interleaving of steps in the increments of `a` (and now possibly `b`) still apply. We can organize the possibilities into three cases, depending upon how many threads see `a<b` as true.

   - All six threads see `a<b` as true. The outcomes then simply depend upon the interleavings of the increments to `a`, so possible outcomes are `a==1/2/3/4/5/6, b==4`.
   - Exactly five threads see `a<b` as true. The sixth thread sees `a>=b` and definitely increments `b` to 5. The increments from the five `a<b` threads can be still be interleaved in many ways - at least four of them must happen "correctly" (otherwise `a` can't have reached 4), but the other one could have read `a` at any time during that phase, incremented the value it read and then written it back at some later point. This means that the possible outcomes are `a==1/2/3/4/5, b==5`.
   - Exactly four threads see `a<b` as true. For this to happen, the other two threads can't have read `a` in the test until all four increments of `a` have taken effect. The increments of `b` may be interleaved, so possible outcomes are `a==4, b==5/6`.

3. The atomicity of the increments simplifies things quite a bit! No increments will be lost, so since there are five increments altogether and the final sum of `a` and `b` will definitely be 9. There are only two scenarios: either all threads see `a<b` as true, so we end up with `a==5, b==4`, or one of the threads sees `a==b` (i.e. it reads `a` after the four increments), so we end up with `a==4, b==5`.

4. There are just two threads here and the increment of `a` is atomic. However, we should beware of the "stupid" `a<a` test. It feels as though this should always be false. It might by optimized by a compiler which wasn't aware of potential concurrency, either by making only one read of `a` or even by removing the condition altogether and just executing the `else` branch. But it is also perfectly valid for this to be implemented with two reads of `a`. In fact, there are three cases here, but only two final outcomes.

   - Both threads see `a==a==0`, and execute the increment, leaving `a==2`
   - One thread sees `a==a==0`, and increments `a`. The other thread sees `a` twice after the increment, i.e. `a==a==1`, and so also executes the increment, also leaving `a==2`.
   - One thread sees `a==a==0`, and increments `a`. The other thread reads `a` before and after this increment (and so sees `a<a` true), then sets `a` to 10, leaving leaving `a==10`.

   Note that there is no possibility of the outcome `a==1`, because the atomic increments can't be badly interleaved, and because the assignment of 10 can only occur if the other thread's increment of `a` has completed.

5. They key issue here is that the `await` thread may or may not see the state when flag is 1. If it does, then this will happen after the setting of `a` to 25. So `x==25,a==25,flag==0/1`

are both possible (depending on when `flag=0` takes effect). It is also possible that the flag is reset (to 0) after the setting of `a`, but before the `await` "sees" it. This would cause the program not to terminate, since the `await` will never complete (it is left waiting to see `flag==1`). At this time, we would have `a==25, flag==0` and `x` will have whatever value it had before this code fragment.

6. Clearly neither `await` can act immediately, so the setting of `x` to 4 happens first. There is then a race. If the `<await (x>2) x=1;>` acts first, then the other one never executes, and hence the program fails to terminate, with `x==1`. If the `<await (x>3) x=10;>` executes first, then the program does terminate, with `x==1`.