Lecturer: Nir Shavit

Handout 12-s

The following are the answers to the second homework assignment.

Question 1

full/empty bits have an infinite consensus number.

Consider the following construction. The variable *decision* is initialized to $\perp$, which is no process' input value, and its *full/empty* bit is *empty*.

```
DECIDE(input) returns(value);
    ConditionalWrite(decision, input);
    return Read(decision);
    end DECIDE
```

Each process does a *ConditionalWrite* on *decision*. The semantics of *ConditionalWrite* specify that only the first *ConditionalWrite* will succeed, and set the *full/emtpy* bit to *full*. All subsequent *ConditionalWrites* will see the *full* bit and fail. Since standard *Reads* do not clear the *full/empty* bits, they can be used to determine the winner of the consensus algorithm.

Question 2

3-valued: The solution wil be given next week after you guys attempt to solve the problems in the end of this weeks homework...

As for boolean: The consensus number for *Compare&Swap* on a Boolean register is two. Consider the following construction. The variable r is a Boolean register, initialized to 0.

```
DECIDE(input ∈ {0,1}) returns(value);
  prefer[i] = input;
  first := Compare&Swap(r, 0, 1);
  if (first = 0)
    then return input;
    else  return prefer[i];
    end if
  end DECIDE
```

This construction shows that *Compare&Swap* on a Boolean register has consensus number at least 2.

To see that it has a consensus number no greater than two, let us prove that the operations commute and overwrite, that is, the 2-valued *Compare&Swap* is interfering.

Assume without loss of generality that a register v contains value 0. (1 symmetric). There are 4 operations:

f_1: *Compare&Swap*(v, 0, 0)
f_2: *Compare&Swap*(v, 0, 1)
f_3: *Compare&Swap*(v, 1, 0)
f_4: *Compare&Swap*(v, 1, 1)

	Result
f_1, f_1 : commute	$v = 0$
f_1, f_2 : commute	$v = 1$
f_1, f_3 : commute	$v = 0$
f_1, f_4 : commute	$v = 0$
f_2, f_2 : commute	$v = 1$
f_2, f_3 : overwrite : *i.e.* $f_2(v) = f_2(f_3(v))$	$v = 1$
f_2, f_4 : commute	$v = 1$
f_3, f_3 : commute	$v = 0$
f_3, f_4 : commute	$v = 0$
f_4, f_4 : commute	$v = 0$

Therefore, F is interfering since $\forall v$, i, j, the f either commute or overwrite.

Question 3

The consensus number of the gs object is 1.

Proof: As usual, we manipulate the state of the system to be bivalent, with the next step of P moving the system into an x-valent state, and the next step of Q moving the system into a y-valent state. By the standard arguments (operations on atomic registers has been proven to be insufficient for providing consensus and *label* operations on different *gs* objects commute), P and Q must be trying to *label* the same *gs* object. If Q goes first, its label puts the *gs* object into state s_2. This is system state s and it is y-valent. If P goes first, its *label* puts the *gs* object into state s_1. Then Q's *label* moves it into s_2. This new systen state, s' differs from s, assuming that P is suspended in both cases (which can not prevent Q's eventual termination due to the wait-free requirement), only by the internal state of P. However, s is y-valent and s' is x-valent. This is a contradiction. ■

Question 4

The consensus number for $k > 2$ is 1. Take for example a 3 process 2-set-consensus object, and assume by way of contradiction that there is a wait-free 2 process consensus protocol using as many copies of this object as you wish. Clearly it is valid that in every execution the 2-set-consensus objects return in each of the three possible accesses to each the values of the first two processes to access. Since there are only two processes, the third access must be completely after one of the other two. We can thus replace each 2-process set-consensus by two registers in which each of the processes first writes its input value during its first access and reads it during its second. We thus have an implementation of 2 process consensus using read/write registers, a contradiction.

Question 5

The consensus number is 1 based on the following implementation algorithm of an approximate agreement algorithm from two atomic registers (see Figure 1). For simplicity assume that the inputs to approximate agreement are non-negative.

Let us begin by outlining a simple variant of the algorthm for the case of two processes. Each of the processes p_i, $i \in \{0, 1\}$ has a register which it can write and the other can read. Here and elsewhere, we let $\bar{i}$ denote the index of the other process, i.e., $\bar{i} = 1 - i$. Due to the asynchrony in the system, it is impossible to have processes agree on one of the input values. Thus, our algorithm has them gradually converge from the input values x_0 and x_1 to value that are only ϵ apart. A process p_i repeatedly does the following: it writes its value v_i (initially the input value x_i) into its register, and then reads $p_{\bar{i}}$'s register. If p_i reads $\perp$ from $v_{\bar{i}}$, it must decide on its own value, since it can never know when $p_{\bar{i}}$ will write its input value (if at all, because $p_{\bar{i}}$ could have failed before writing. If p_i reads a non-$\perp$ value from $v_{\bar{i}}$, it! checks whether or not $|v_{\bar{i}} - v_i| \leq \epsilon$. If it is, p_i decides on its own value. If not, p_I sets v_i to be $\frac{v_i + v_{\bar{i}}}{2}$ and repeats.

Due to asynchrony, processes do not necessarily converge "directly" to a value. Rather, the following type of scenario is possible: $p_{\bar{i}}$, having previously written $v_{\bar{i}}$, reads p_i's current value v_i, and is delayed just before writing $\frac{v_i + v_{\bar{i}}}{2}$ to its register; then p_i repeatedly reads and writes, cutting the interval in half till its value is very close to $v_{\bar{i}}$; finally, $p_{\bar{i}}$ completes the write of $\frac{v_i + v_{\bar{i}}}{2}$ to its register, so that in fact, p_i has moved "too far" towards $p_{\bar{i}}$'s old value. This can repeat itself again and again. However, it can easily be seen that in every "time slice" in which both p_i and $p_{\bar{i}}$ each perform a read and a write, the diameter of the proposed values, $|v_i - v_{\bar{i}}|$, is cut by at least a half, and so the values converge in $O(\log(\frac{x_i - x_{\bar{i}}}{\epsilon}))$ such "slices". The algorithm is wait-free, ! since each process can ultimately reach a decision independently of the other taking steps.

I leave it up to you to prove formally that the algorithm does converge within a finite number of steps.

```
WAIT-FREE-APPROX(x, ε) returns(real);
    r_i := x;
    if r_i = ⊥ then return r_i;
    repeat r_i := (max(r_i, r_ī) − min(r_i, r_ī))/2;
            until |r_i − r_ī| ≤ ε;
    end repeat
    return r_i;
    end WAIT-FREE-APPROX
```

Figure 1: wait-free 2-process approximate agreement— Code for process i