MODULE *AfekSimplified*

This module specifies the simplified *Afek* et al. snapshot algorithm algorithm described in Section 6.3 of the paper "Auxiliary Variables in TLA+". This is a simplified version of an algorithm in the 1993 paper "Atomic snapshots of Shared Memory" by *Afek*, *Attiya*, *Dolev*, *Gafni*, Merritt, and *Shavit*. It will be shown to satisfy the safety specification of a linearizable snapshot object in module *NewLinearSnapshot*. (The actual algorithm by *Afek* et al. also satisfies the specification's liveness property, but our simplified version does not.)

```
EXTENDS Integers
```

We begin by declaring and defining the same constants as in module *NewLinearSnapshot*.

```
CONSTANTS Readers, Writers, RegVals, InitRegVal

ASSUME  ∧ Readers ∩ Writers = {}
        ∧ InitRegVal ∈ RegVals

MemVals    ≜ [Writers → RegVals]
InitMem    ≜ [i ∈ Writers ↦ InitRegVal]
NotMemVal  ≜ CHOOSE v : v ∉ MemVals
NotRegVal  ≜ CHOOSE v : v ∉ RegVals
```

Instead of the internal variable *mem* of the specification, the algorithm maintains an internal variable *imem* such that for each writer i, the value of $imem[i]$ is a pair $\langle v, k\rangle$, where v is the last register value written by i, and k is the number of times the register has been written by i. The purpose of the second component of $imem[i]$ is to ensure that values written to $imem[i]$ by writer i in different write operations are different.

We now define some constants, including the set *IMemVals* of all possible values of *imem*.

```
IRegVals   ≜ RegVals × Nat
IMemVals   ≜ [Writers → IRegVals]
InitIMem   ≜ [i ∈ Writers ↦ ⟨InitRegVal, 0⟩]
```

In addition to *imem*, the algorithm has three internal variables: *wrNum*, *rdVal*1, and *rdVal*2. Each writer i records in $wrNum[i]$ the number of times it has written $imem[i]$. Writer i acts pretty much like the writer in the specification, except that $DoWr(i)$ writes a pair of values in $imem[i]$ and increments $wrNum[i]$. The writer needs no other internal information because it knows that it has performed a $BeginWr(i, cmd)$ step but not the subsequent $DoWr(i)$ step if $wrNum[i]$ is different from $imem[i][2]$; and it doesn't have to remember the command *cmd* because that's in $interface[i]$.

Reader i keeps performing the following scan procedure until the procedure succeeds in computing an output, whereupon the read operation terminates by producing that output. The scan procedure reads *imem* by reading the elements $imem[j]$ one at a time in any order, and it then reads *imem* again by reading its elements in any order. The scan procedure succeeds if both reads obtain the same value of *imem*, in which case it produces the output consisting of the register values of that value of *imem*. (This procedure produces a correct output only because a writer j cannot write the same value twice in $imem[j]$.) It's possible for the scan procedure never to succeed, in which case the read operation never terminates. Afek et al. have a method for terminating after a finite number of unsuccessful scans, but it complicates the algorithm without significantly changing the structure of its correctness proof.

Reader i keeps in $rdVal1[i][j]$ and $rdVal2[i][j]$ the values of $imem[j]$ that it has read so far during the current scan procedure's reads of *imem*. The values of $rdVal1[i]$ and $rdVal2[i]$ are each a function that maps a subset of the writers to the values it has read for those writer's registers. They both equal the function $\langle\rangle$ with empty domain when the writer is not performing a scan.

With this explanation of how the algorithm works, it should be easy to understand its TLA+ specification.

```
VARIABLES imem, interface, wrNum, rdVal1, rdVal2
vars ≜ ⟨imem, interface, wrNum, rdVal1, rdVal2⟩
```

We define $PartialFcns(U, V)$ to be the set of functions from a subset of U to V. It is used only in the type-correctness invariant.

```
PartialFcns(U, V) ≜ UNION {[D → V] : D ∈ SUBSET U}
TypeOK ≜ ∧ imem ∈ IMemVals
         ∧ ∧ DOMAIN interface = Readers ∪ Writers
           ∧ ∀ i ∈ Readers : interface[i] ∈ MemVals ∪ {NotMemVal}
           ∧ ∀ i ∈ Writers : interface[i] ∈ RegVals  ∪ {NotRegVal}
         ∧ wrNum ∈ [Writers → Nat]
         ∧ rdVal1 ∈ [Readers → PartialFcns(Writers, IRegVals)]
         ∧ rdVal2 ∈ [Readers → PartialFcns(Writers, IRegVals)]

Init ≜ ∧ imem = InitIMem
       ∧ interface = [i ∈ Readers ∪ Writers ↦
                          IF i ∈ Readers THEN InitMem ELSE  NotRegVal]
       ∧ wrNum = [i ∈ Writers ↦ 0]
       ∧ rdVal1 = [i ∈ Readers ↦ ⟨⟩]
       ∧ rdVal2 = [i ∈ Readers ↦ ⟨⟩]

BeginWr(i, cmd) ≜ ∧ interface[i] = NotRegVal
                  ∧ wrNum′ = [wrNum EXCEPT ![i] = wrNum[i] + 1]
                  ∧ interface′ = [interface EXCEPT ![i] = cmd]
                  ∧ UNCHANGED ⟨imem, rdVal1, rdVal2⟩

DoWr(i) ≜ ∧ interface[i] ∈ RegVals
          ∧ imem[i][2] ≠ wrNum[i]
          ∧ imem′ = [imem EXCEPT ![i] = ⟨interface[i], wrNum[i]⟩]
          ∧ UNCHANGED ⟨interface, wrNum, rdVal1, rdVal2⟩

EndWr(i) ≜ ∧ interface[i] ∈ RegVals
           ∧ imem[i][2] = wrNum[i]
           ∧ interface′  = [interface EXCEPT ![i] = NotRegVal]
           ∧ UNCHANGED ⟨imem, wrNum, rdVal1, rdVal2⟩

BeginRd(i) ≜ ∧ interface[i] ∈ MemVals
             ∧ interface′ = [interface EXCEPT ![i] = NotMemVal]
             ∧ UNCHANGED ⟨imem, wrNum, rdVal1, rdVal2⟩
```

If x is not in the domain of the function v, then $AddToFcn(f, x, v)$ is the function obtained from f by adding x to its domain and letting x be mapped to v. This could be written as as f @@ $(x :> v)$, where the operators :> and @@ are defined in the standard *TLC* module.

```
AddToFcn(f, x, v) ≜
  [y ∈ (DOMAIN f) ∪ {x} ↦ IF y = x THEN v ELSE f[y]]

Rd1(i) ≜ ∧ interface[i] = NotMemVal
         ∧ ∃ j ∈ Writers \ DOMAIN rdVal1[i] :
              rdVal1′ = [rdVal1 EXCEPT ![i] = AddToFcn(rdVal1[i], j, imem[j])]
         ∧ UNCHANGED ⟨interface, imem, wrNum, rdVal2⟩

Rd2(i) ≜ ∧ interface[i] = NotMemVal
         ∧ DOMAIN rdVal1[i] = Writers
         ∧ ∃ j ∈ Writers \ DOMAIN rdVal2[i] :
              rdVal2′ = [rdVal2 EXCEPT ![i] = AddToFcn(rdVal2[i], j, imem[j])]
         ∧ UNCHANGED ⟨interface, imem, wrNum, rdVal1⟩

TryEndRd(i) ≜ ∧ interface[i] = NotMemVal
              ∧ DOMAIN rdVal1[i] = Writers
              ∧ DOMAIN rdVal2[i] = Writers
              ∧ IF rdVal1[i] = rdVal2[i]
                  THEN ∧ interface′ =
                            [interface EXCEPT
                              ![i] = [j ∈ Writers ↦ rdVal1[i][j][1]]]
                  ELSE ∧ interface′ = interface
              ∧ rdVal1′ = [rdVal1 EXCEPT ![i] = ⟨⟩]
              ∧ rdVal2′ = [rdVal2 EXCEPT ![i] = ⟨⟩]
              ∧ UNCHANGED ⟨imem, wrNum⟩

Next ≜ ∨ ∃ i ∈ Readers : BeginRd(i) ∨ Rd1(i) ∨ Rd2(i) ∨ TryEndRd(i)
       ∨ ∃ i ∈ Writers : ∨ ∃ cmd ∈ RegVals : BeginWr(i, cmd)
                         ∨ DoWr(i) ∨ EndWr(i)
```

Since a read need never terminate, the algorithm doesn't satisfy the *NewLinearSnapshot* specification's liveness requirements, so we don't bother specifying any fairness of the actions.

```
Spec ≜ Init ∧ □[Next]_vars
```

\* Modification History

\* Last modified Sat *Oct* 22 01:58:50 *PDT* 2016 by *lamport*

\* Created *Wed Oct* 05 08:23:18 *PDT* 2016 by *lamport*