For the containment property, let us consider two processes p_i and p_j that stop at stairs k_i, and k_j, respectively. Without loss of generality, let $k_i \leq k_j$. Due to Lemma 20, there are exactly k_i processes on the stairs 1 to k_i, and k_j processes on stairs 1 to $k_j \leq k_i$. As no process backtracks on the stairway (a process proceeds downwards or stops), the set of k_j processes returned by p_j includes the set of k_1 processes returned by p_i.

Let us finally consider the immediacy property. Let us first observe that a process deposits its value before starting its descent of the stairway (line 1), from which it follows that, if $j \in set_i$, $REG[j]$ contains the value v_j deposited by p_j. Moreover, it follows from lines 4 and 5 that, if a a process p_j stops at a stair k_j and then $i \in set_j$, then p_i stopped at a stair $k_i \leq k_j$. It then follows from Lemma 20 that the set set_j returned by p_j includes the set set_i returned by p_i, from which follows the immediacy property. □

8.5.4 A Recursive Implementation of a One-Shot Immediate Snapshot Object

This section describes a recursive implementation of a one-shot immediate snapshot object due to E. Gafni and S. Rajsbaum (2010). This construction can be seen as a recursive formulation of the previous iterative algorithm.

Underlying data structure As we are about to see, when considering distributed computing, an important point that distinguishes distributed recursion from sequential recursion on data structures lies in the fact that the recursion parameter is usually the number n of processes involved in the computation. The recursion parameter is used by a process to compute a view of the concurrency degree among the participating processes.

The underlying data structure representing the immediate snapshot object consists of a shared array $REG[1..n]$ such that each $REG[x]$ is an array of n SWMR atomic registers. The aim of $REG[x]$, which is initialized to $[\perp, \ldots, \perp]$, is to contain the view obtained by the processes that see exactly x other processes in the system. For any x, $REG[x]$ is accessed only by the processes that attain recursion level x and the atomic register $REG[x][i]$ can be read by all these processes but can be written only by p_i.

The recursive algorithm implementing the operation `update_snapshot()` The algorithm is described in Fig. 8.16. Its main call is an invocation of `rec_update_snapshot`(n, v_i), where n is the initial value of the recursion parameter and v_i the value that p_i wants to deposit into the immediate snapshot object (line 1). This call is said to occur at recursion level n. More generally, an invocation `rec_update_snapshot`$(x, -)$ is said to occur at recursion level x. Hence, the recursion levels are decreasing from level n to level $n-1$, then to level $n-2$, etc. (Actually, a recursion level corresponds to what was called a "level" in Sect. 8.5.3.)

```
operation update_snapshot(v_i) is
(1)   my_view_i ← rec_update_snapshot(n, v_i)
(2)   return(my_view_i)
end operation.

operation rec_update_snapshot(x, v) is
      % x is the recursion parameter (n ≥ x ≥ 1) %
(3)   REG[x][i] ← v;
(4)   for each j ∈ {1, . . . , n} do reg_i[j] ← REG[x][j] end for;
(5)   view_i ← { (j, reg_i[j]) | reg_i[j] ≠ ⊥ };
(6)   if (|view_i| = x) then res_i ← view_i
(7)                     else res_i ← rec_update_snapshot(x − 1, v)
(8)   end if;
(9)   return(res_i)
end operation.
```

Fig. 8.16 Recursive construction of a one-shot immediate snapshot object (code for process p_i)

When it invokes rec_update_snapshot(x, v), p_i first writes v into $REG[x][i]$ and reads asynchronously the content of $REG[x][1..n]$ (lines 3–4, let us notice that these lines implement a store-collect). Hence, the array $REG[x][1..n]$ is devoted to the xth recursion level.

Then, p_i computes the view $view_i$ obtained from $REG[x][1..n]$ (line 5). Let us remark that, as the recursion levels are decreasing and there are at most n participating processes, the set $view_i$ contains the values deposited by $n' = |view_i|$ processes, where n' is the number of processes that, from p_i's point of view, have attained recursion level x.

If p_i sees that exactly x processes have attained the recursion level x (i.e., $n' = x$), it returns $view_i$ as the result of its invocation of the immediate snapshot object (lines 6 and 9). Otherwise, fewer than x processes have attained recursion level x and consequently p_i invokes recursively rec_update_snapshot$(x - 1, v)$ (line 7) in order to attain a recursion level $x' < x$ accessed by exactly x' processes. It will stop its recursive invocations when it attains such a recursion level (in the worst case, $x' = 1$).

Theorem 38 *The algorithm described in Fig. 8.16 is a wait-free construction of an immediate snapshot object. Its step complexity (number of shared memory accesses) is $O(n(n - |res| + 1))$, where res is the set returned by* update_snapshot(v).

Proof Claim C. If at most x processes invoke rec_update_snapshot$(x, -)$ then (a) at most $(x - 1)$ processes invoke rec_update_snapshot$(x - 1, -)$ and (b) at least one process stops at line 6 of its invocation rec_update_snapshot$(x, -)$.

Proof of claim C. Assuming that at most x processes invoke update_snapshot $(x, -)$, let p_k be the last process that writes into $REG[x][1..n]$. We necessarily have $|view_k| \leq x$.If p_k finds $|view_k| = x$, it stops at line 6. Otherwise, we have $|view_k| < x$ and p_k invokes rec_update_snapshot$(x - 1, -)$ at line 7. But in that case, as p_k is the last process that wrote into the array $REG[x][1..n]$, it follows from $|view_k| < x$ that fewer than x processes have written into $REG[x][1..n]$, and consequently, at most $(x-1)$ processes invoke rec_update_snapshot$(x-1, -)$. End of the proof of claim C.

To prove the termination property, let us consider a correct process p_i that invokes update_snapshot(v_i). Hence, it invokes rec_update_snapshot$(n, -)$. It follows from Claim C and the fact that at most n processes invoke rec_update_snapshot $(n, -)$ that either p_i stops at that invocation or belongs to the set of at most $n-1$ processes that invoke rec_update_snapshot$(n-1, -)$. It then follows by induction from the claim that if p_i has not stopped during a previous invocation, it is the only process that invokes rec_update_snapshot(1). It then follows from the text of the algorithm that it stops at that invocation.

The proof of the self-inclusion property is trivial. Before stopping at recursion level x (line 6), a process p_i has written v_i into $REG[x][i]$ (line 3), and consequently we have then $(i, v_i) \in view_i$, which concludes the proof of the self-inclusion property.

To prove the self-containment and immediacy properties, let us first consider the case of two processes that return at the same recursion level x. If a process p_i returns at line 6 of recursion level x, let $view_i[x]$ denote the corresponding value of $view_i$. Among the processes that stop at recursion level x, let p_i be the last process which writes into $REG[x][1..n]$. As p_i stops, this means that $REG[x][1..n]$ has exactly x entries different from $\perp$ and (due to Claim C) no more of its entries will be set to a non-$\perp$ value. It follows that, as any other process p_j that stops at recursion level x reads x non-$\perp$ entries from $REG[x][1..n]$, we have $view_i[x] = view_j[x]$ which proves the properties.

Let us now consider the case of two processes p_i and p_j that return at line 6 of recursion level x and y, respectively, with $x > y$; i.e., p_i returns $view_i[x]$ while p_j returns $view_j[y]$. The self-containment follows then from $x > y$ and the fact that p_j has written into all the arrays $REG[z][1..n]$ with $n \geq z \geq y$, from which we conclude that $view_j[y] \subseteq view_i[x]$. Moreover, as $x > y$, p_i has not written into $REG[y][1..n]$ while p_j has written into $REG[x][1..n]$, and consequently $(j, v_j) \in view_i[x]$ while $(i, v_i) \notin view_j[y]$, from which the containment and immediacy properties follow.

As far as the number of shared memory accesses is concerned we have the following. Let *res* be the set returned by an invocation of rec_update_snapshot$(n, -)$. Each recursive invocation costs $n+1$ shared memory accesses (lines 3–4). Moreover, the sequence of invocations, namely rec_update_snapshot$(n, -)$, rec_update_snapshot$(n-1, -)$, etc., until rec_update_snapshot$(|res|, -)$ (where $x = |res|$ is the recursion level at which the recursion stops) contains $n - |res| + 1$ invocations. It follows that the cost is $O(n(n - |res| + 1))$ shared memory accesses. □