Lower Bounds in Distributed Computing

by

Rui Fan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author ....
Department of Electrical Engineering and Computer Science
February 1, 2008

Certified by ....
Nancy A. Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Chapter 4

Mutual Exclusion

4.1 Introduction

In the mutual exclusion (*mutex*) problem, a set of processes communicating via shared memory access a shared resource, with the requirement that at most one process can access the resource at any time. Mutual exclusion is a fundamental primitive in many distributed algorithms, and is also a foundational problem in the theory of distributed computing. Numerous algorithms for solving the problem in a variety of cost models and hardware architectures have been proposed over the past four decades. In addition, a number of recent works have focused on proving lower bounds for the cost of mutual exclusion. The *cost* of a mutex algorithm may be measured in terms of the number of memory accesses the algorithm performs, the number of shared variables it accesses, or other measures reflective of the performance of the algorithm in a multicomputing environment.

In this chapter, we introduce a new *state change* cost model, based on a simplification of the standard *cache coherent* model [4], in which an algorithm is charged for performing operations that change the system state. Let a *canonical execution* be any execution in which n different processes each enter the critical section (*i.e.*, accesses the shared resource) exactly once. We prove that any deterministic mutex algorithm using registers must incur a state change cost of $\Omega(n \log n)$ in some canonical execution. This lower bound is tight, as the algorithm of Yang and Anderson [40] has $O(n \log n)$ cost in all canonical executions with our cost measure. To prove the result, we introduce a novel technique which is *information theoretic* in nature. We first argue that in each canonical execution, processes need to cumulatively acquire a certain amount of information. We then relate the amount of information processes can obtain by accessing shared memory to the cost of those accesses, to obtain a lower bound on the cost of the mutex algorithm.

We conjecture that this informational proof technique can be adapted to prove $\Omega(n \log n)$ cost lower bounds for mutual exclusion in the cache coherent and *distributed shared memory* [4] cost models[1], and in shared memory systems in which processes have access to shared objects more powerful than registers. Furthermore, the informational viewpoint may be useful in studying lower bounds for other distributed computing problems.

We now give a brief description of our proof technique. Intuitively, in order for n processes to all enter the critical section without colliding, the "visibility graph" of the processes, consisting of directed edges going from each process to all the other processes that it "sees", must contain a directed chain on all n processes. Indeed, if there exist two processes, neither of which has an edge to (sees) the other, then both processes could enter the critical section at the same time. To build up this directed n-chain during an execution, the processes must all together acquire $\Omega(n \log n)$ bits of information, enough to specify the permutation on the n process indices corresponding to the n-chain. We show that in some canonical executions, each time the processes perform some memory accesses with cost C, they gain only $O(C)$ bits of information. This implies that in some canonical executions, the processes must incur $\Omega(n \log n)$ cost. To formalize this intuition, we construct, for any permutation $\pi \in S_n$, an equivalence class (*i.e.*, a set) of executions A_π, such that for any $\alpha \in A_\pi$, a process ordered lower in π does not see any processes ordered higher in π[2][3]. In any $\alpha \in A_\pi$, we can show that the processes must enter their critical sections in the order specified by π. This implies that for permutations $\pi_1 \neq \pi_2$, we have $A_{\pi_1} \cap A_{\pi_2} = \emptyset$. We can show that all executions in A_π have the same cost, say C_π. We then show that we can *encode* A_π using $O(C_\pi)$ bits. Since $A_{\pi_1} \cap A_{\pi_2} = \emptyset$ for $\pi_1 \neq \pi_2$, and since it takes $\Omega(n \log n)$ bits to identify some $\pi \in S_n$, then there must exist some π for which $C_\pi = \Omega(n \log n)$.

The remainder of this chapter is organized as follows. In Section 4.2, we describe related work on mutual exclusion and other lower bounds. In Section 4.3, we formally define the mutual exclusion problem and the state change cost model. We give a detailed overview of our proof in Section 4.4. In Section 4.5, we present an adversary that, for every $\pi \in S_n$, constructs a set of executions A_π, such that all executions in the set have cost C_π. We prove correctness properties about this construction algorithm in Section 4.6, and show some additional properties about the construction in Section 4.7. We then show in Section 4.8 how to encode A_π as a string E_π of length $O(C_\pi)$, and prove this encoding is correct in Section 4.9. In Section 4.10, we show E_π uniquely identifies some $\alpha_\pi \in A_\pi$, by presenting a decoding algorithm that recovers α_π from E_π. The decoding algorithm is proved correct in Section 4.11. Our main lower bound result, which follows from this unique decoding, is presented in Section 4.12.

[1]At a high level, the cache coherent and distributed shared memory cost models assign costs to executions of a mutex algorithm based on the *locality* of the shared objects accessed: it is cheaper for a process to access a nearby object than a faraway one. The state change cost model can be interpreted as assigning costs in a similar way. All three cost models are discussed and compared in Section 4.3.3.

[2]A process is *ordered lower* in π if it appears earlier in π. For example, if $\pi = (4213)$, so that 1 maps to 4, 2 maps to 2, etc., then 4 is ordered lower in π than 1.

[3]The reason that we construct an equivalence class A_π, instead of constructing one particular $\alpha \in A_\pi$, is explained in Section 4.5.

The results described in this chapter appeared earlier in [15].

4.2 Related Work

Mutual exclusion is a seminal problem in distributed computing. Starting with Dijkstra's work in the 1960's, research in mutual exclusion has progressed in response to, and has sometimes driven, changes in computer hardware and the theory of distributed computing. For interesting accounts of the history of this problem, we refer the reader to the excellent book by Raynal [34] and survey by Anderson, Kim and Herman [4].

The performance of a mutual exclusion algorithm depends on a variety of factors. An especially relevant factor for modern computer architectures is *memory contention.* In [1], Alur and Taubenfeld prove that for any nontrivial mutual exclusion algorithm, some process must perform an unbounded number of memory accesses to enter its critical section. This comes from the need for some processes to busywait until the process currently in the critical section exits. Therefore, in order for a mutex algorithm to scale, it must ensure that its busywaiting steps do not congest the shared memory. *Local-spin* algorithms were proposed in [19] and [29], in which processes busywait only on *local* or *cached* variables, thereby relieving the gridlock on main memory. Local-spin mutex algorithms include [40], [23] and [3], among many others. In particular, the algorithm of Yang and Anderson [40] performs $O(n \log n)$ remote memory accesses in a canonical execution in which n processes each complete their critical section once. A remote memory access is the unit of cost in local spin algorithms. The cost of the YA algorithm is computed by "discounting" busywaiting steps on local variables. That is, several local busywaiting steps may be charged only once.

A number of lower bounds exist on the number of shared memory objects an algorithm needs to solve mutual exclusion [8]. Recently, considerable research has focused on proving time complexity (number of remote memory accesses) lower bounds for the problem. Cypher [9] first proved that any mutual exclusion algorithm must perform $\Omega(n \frac{\log \log n}{\log \log \log n})$ total remote memory accesses in some canonical execution. An improved lower bound by Anderson and Kim [2] showed that there exists an execution in which *some* process must perform at least $\Omega(\frac{\log n}{\log \log n})$ remote memory accesses. However, this result does not give a nontrivial lower bound for the *total* number of remote memory accesses performed by all the processes in a canonical execution. The techniques in these papers involve keeping the set of processes contending for the critical section "invisible" from each other, and eliminating certain processes when they become visible. Our technique is fundamentally different, because we do not require all processes to be invisible to each other. Instead, in the executions we construct, there is a permutation of the n processes such that processes indexed higher in the permutation can see processes indexed lower, but not vice versa. Instead of eliminating visible processes, we keep track of the amount of information that the processes have acquired. Additionally, in the adversarial execution constructed in [2], processes execute mostly in lock step, where as in our construction, the execution of processes adapts adversarially to the mutex algorithm against which we prove our lower bound, reminiscent of diagonalization arguments. Information-based arguments of a different nature than ours have been used by Jayanti [21] and Attiya and Hendler [6], among others, to prove lower bounds for other problems.

4.3 Model

In this section, we define the formal model for proving our lower bound. We first describe the general computational model, then define the mutual exclusion problem, and the state change cost model for computing the cost of an algorithm.

4.3.1 The Shared Memory Framework

In the remainder of this chapter, fix an integer $n \geq 1$. For any positive natural number t, we use $[t]$ to denote the set $\{1, \ldots, t\}$. A *system* consists of a set of *processes* $p_1, \ldots, p_n$, and a collection L of *shared variables*. Where it is unambiguous, we sometimes write i to denote process p_i. A shared variable consists of a *type* and an *initial value*. In this chapter, we restrict the types of all shared variables to be multi-reader multi-writer registers. Let V be the set of values that the registers can take, and assume that all registers start with some initial value $v_0 \in V$. For each $i \in [n]$, we define a set S_i representing the set of states that process p_i can be in. We assume that p_i is initially in a state $\hat{s}_0^i \in S_i$. A *system state* is a tuple consisting of the states of all the processes and the values of all the registers. Let S denote the set of all system states. A system starts out in the initial system state $\hat{s}_0 \in S$, defined by the initial states of all the processes and the initial values of all the registers. Given a system state s, let $st(s, i)$ denote the state of process p_i in s, and let $st(s, \ell)$ denote the value of register ℓ in s.

Let $i \in [n]$, and let E_i denote the set of actions that p_i can perform to interact with the shared memory and the external environment. We call each $e \in E_i$ a *step* by p_i. e can be one of two types, either a *shared memory access* step, or a *critical* step. Critical steps are specific to the mutual exclusion problem, and will be described in Section 4.3.2. Here, we describe the shared memory access steps. Let $\ell \in L$ and $v \in V$. Then there exists a step $\mathsf{read}_i(\ell) \in E_i$, representing a read by p_i of register ℓ. We write $proc(\mathsf{read}_i(\ell)) = i$, indicating that p_i performs this step, and we write $reg(\mathsf{read}_i(\ell)) = \ell$, indicating that the step accesses ℓ. There also exists a step $\mathsf{write}_i(\ell, v) \in E_i$, representing a write by p_i of value v to register ℓ. We write $proc(\mathsf{write}_i(\ell, v)) = i$, $reg(\mathsf{write}_i(\ell, v)) = \ell$, and $val(\mathsf{write}_i(\ell, v)) = v$, to indicate that this step writes value v. Given a step e of the form $\mathsf{read}_i(\cdot)$, and a step e' of the form $\mathsf{write}_i(\cdot, \cdot)$, we say that e is a *read step* by p_i, and e' is a *write step* by p_i.

Let $\overline{E} = \bigcup_{i \in [n]} E_i$, and $\overline{S} = \bigcup_{i \in [n]} S_i$. A *state transition* function is a (deterministic, partial) function $\Delta : S \times \overline{E} \times [n] \to \overline{S}$, describing how any process changes its state after performing a step. More precisely, let $s \in S$, $i \in [n]$ and $e \in E_i$. Then if p_i performs e in system state s, its resulting state is $\Delta(s, e, i) \in S_i$. For example, if e is a read step by i on register ℓ, then $\Delta(s, e, i)$ is p_i's state after it reads value $st(s, \ell)$ while in state $st(s, i)$. A *step transition* function is a (deterministic, partial) function $\delta : \overline{S} \times [n] \to \overline{E}$. Let $i \in [n]$ and $s \in S_i$. Then $\delta(s, i) \in E_i$ is the next step that p_i will take if it is currently in state s.

An *execution* of a system consists of a (possibly infinite) alternating sequence of system states and process steps, beginning with the initial system state. That is, an execution is of the form $\hat{s}_0 e_1 s_1 e_2 s_2 \ldots$, where each s_i is a system state, and each e_i is a step by some process. The state changes and steps taken are consistent with Δ and δ. That is, for any $i \geq 1$, if e_i is a step by process p_j, then we have

$$e_i = \delta(st(s_{i-1}, j), j), \qquad st(s_i, j) = \Delta(s_{i-1}, e_i, j), \qquad \forall k \neq j : st(s_i, k) = st(s_{i-1}, k). \tag{4.1}$$

Here, we define $s_0 = \hat{s}_0$. Also, if e_i has the form $\mathsf{write}.(\ell, v)$, for some $\ell \in L$ and $v \in V$, then we have

$$st(s_i, \ell) = v, \qquad \forall \ell' \neq \ell : st(s_i, \ell') = st(s_{i-1}, \ell'). \tag{4.2}$$

If e_i has the form $\mathsf{read}.(\cdot)$, then we have

$$\forall \ell \in L : st(s_i, \ell) = st(s_{i-1}, \ell). \tag{4.3}$$

We say an execution β is an *extension* of α if β contains α as a prefix. If α is finite, we define the *length* of α, written $len(\alpha)$, to be the number of steps in α, and we define $st(\alpha)$ to be the final system state in α. For $i \in [n]$, let $st(\alpha, i)$ be the state of process p_i in $st(\alpha)$, and for $\ell \in L$, let $st(\alpha, \ell)$ be the value of register ℓ in $st(\alpha)$. For $i \in [n]$ and a step e by p_i, we write $\Delta(\alpha, e, i) = \Delta(st(\alpha), e, i)$ for the state of p_i after taking step e in the final state of α. Also, we write $\delta(\alpha, i) = \delta(st(\alpha, i), i)$ for the step p_i takes after the final state in α. Given any algorithm $\mathcal{A}$, we write $execs(\mathcal{A})$ for the set of executions of $\mathcal{A}$.

So far, we have described an execution as an alternating sequence of system states and process steps. Since the state and step transition functions that we consider are deterministic, there is an equivalent and sometimes more convenient representation of an execution as simply its sequence of process steps. We call an execution represented in this form a *run*. More precisely, let $\alpha = \hat{s}_0 e_1 s_1 e_2 s_2 \ldots \in execs(\mathcal{A})$. Then we define $run(\alpha) = e_1 e_2 \ldots$. We define the set of all runs of $\mathcal{A}$ as $runs(\mathcal{A}) = \{run(\alpha) \,|\, \alpha \in execs(\mathcal{A})\}$. Given $\alpha' = e'_1 e'_2 \ldots \in runs(\mathcal{A})$, we write $exec(\alpha') = \hat{s}_0 e'_1 s'_1 e'_2 s'_2 \ldots$ for the execution corresponding to α'. Here, the states s'_i, $i \geq 1$, are defined using Equations 4.1, 4.2 and 4.3. For any of the terminology we defined earlier that refer to executions, we can define the same terminology with respect to a run α, by first converting α to the execution $exec(\alpha)$. For example, if $\alpha \in runs(\mathcal{A})$ and $i \in [n]$, then we define $\delta(\alpha, i) = \delta(exec(\alpha), i)$ for the step p_i takes after the final state in $exec(\alpha)$. Sometimes we write a run $e_1 e_2 \ldots$ as $e_1 \circ e_2 \circ \ldots$, for visual clarity.

We define a *step sequence* $\alpha = e_1 e_2 \ldots$ to be an arbitrary sequence of steps. We write $spseq(\mathcal{A})$ for the set of all step sequences, where any step in any step sequence is a step by some process $p_i, i \in [n]$. A step sequence is not necessarily a run, since the steps in the sequence may not appear in any execution of $\mathcal{A}$. Therefore, a step sequence, unlike a run, is not meant to represent an execution. For any $\ell \in L$, we say that a step sequence α *accesses* ℓ if some step in α either reads or writes to ℓ. We write $acc(\alpha)$ for the set of registers accessed by α. We say that a process $i \in [n]$ *takes steps* in α if at least one of the steps of α is a step by i. We write $procs(\alpha)$ to be the set of processes that take steps in α.

Let $\alpha = e_1 e_2 \ldots$ be a run, and let $t \geq 0$ be a natural number. Then we write $\alpha(t) = e_1 \ldots e_t$ for the length t prefix of α. If $t > len(\alpha)$, then we define $\alpha(t) = \alpha$. Let $\alpha' = e_1 e_2 \ldots e_t$ and $\beta = e'_1 e'_2 \ldots$ be two step sequences, where α' is finite. We write the *concatenation* of α' and β as $\alpha \circ \beta = e_1 e_2 \ldots e_t e'_1 e'_2 \ldots$. Note that $\alpha' \circ \beta$ may or may not be a run. Sometimes we write $\alpha' \beta$ instead of $\alpha' \circ \beta$, for conciseness.

In a shared memory system, each process is aware only of its own state, and the values each register took in all the past times it had read the register. The process may not be aware of the current states of the other processes or the current values of the registers. This sometimes allows us to infer the existence of certain runs of a shared memory algorithm, given the existence of some other runs. In particular, we have the following.

Theorem 4.3.1 (Extension Theorem) *Let $\mathcal{A}$ be an algorithm in a shared memory system, and let $\alpha_1, \alpha_2 \in runs(\mathcal{A})$. Let $\beta \in spseq(\mathcal{A})$ be a step sequence such that $\alpha_1 \beta \in runs(\mathcal{A})$. Suppose that the following conditions hold:*

1. *$st(\alpha_1, i) = st(\alpha_2, i)$, for all $i \in procs(\beta)$.*
2. *$st(\alpha_1, \ell) = st(\alpha_2, \ell)$, for all $\ell \in acc(\beta)$.*

Then $\alpha_2 \beta \in runs(\mathcal{A})$.

The Extension Theorem says that if α_1, α_2 and $\alpha_1 \beta$ are all runs of $\mathcal{A}$, and if the final states in (the executions corresponding to) α_1 and α_2 are identical in the states of all the processes that take steps in β, and in the values of all registers accessed in β, then $\alpha_2 \beta$ is also a run of $\mathcal{A}$. Notice that the states of some processes or the values of some registers may indeed differ after α_1 and α_2. However, as long as those processes do not take steps in β and those registers are not accessed in β, then processes taking steps in β cannot tell the difference between α_1 and α_2. Based on this idea, we now prove the theorem.

Proof of Theorem 4.3.1. Let $\beta = e_1 e_2 \ldots$. For visual clarity, we write, for $k \geq 0$, β_k in place of $\beta(k)$, as the length k prefix of β. Note that $\beta_0 = \varepsilon$, the empty string. For any $k \geq 0$, we prove the following:

$$\alpha_2 \beta_k \in \mathit{runs}(\mathcal{A}) \tag{4.4}$$

$$\forall i \in \mathit{procs}(\beta) : \mathit{st}(\alpha_1\beta_k, i) = \mathit{st}(\alpha_2\beta_k, i), \qquad \forall \ell \in \mathit{acc}(\beta) : \mathit{st}(\alpha_1\beta_k, \ell) = \mathit{st}(\alpha_2\beta_k, \ell). \tag{4.5}$$

Equations 4.4 and 4.5 hold for $k = 0$, by the assumption of the theorem. We show that if it holds up to k, then it holds for $k + 1$.

Suppose e_{k+1} is a step by process p_{i^*}, accessing register ℓ^*. Since $\alpha_1\beta \in \mathit{runs}(\mathcal{A})$, we have $\delta(\alpha_1\beta_k, i^*) = e_{k+1}$. Then, since $\mathit{st}(\alpha_1\beta_k, i^*) = \mathit{st}(\alpha_2\beta_k, i^*)$ by the inductive hypothesis, we have $\delta(\alpha_2\beta_k, i^*) = e_{k+1}$. That is, since p_{i^*} has the same state after runs $\alpha_1\beta_k$ and $\alpha_2\beta_k$, then it performs the same step after $\alpha_1\beta_k$ and $\alpha_2\beta_k$. Thus, we have $\alpha_2\beta_k e_{k+1} = \alpha_2\beta_{k+1} \in \mathit{runs}(\mathcal{A})$, and so Equation 4.4 holds for $k + 1$.

Since $\alpha_2\beta_{k+1} \in \mathit{runs}(\mathcal{A})$, then $\mathit{st}(\alpha_2\beta_{k+1}, i)$ and $\mathit{st}(\alpha_2\beta_{k+1}, \ell)$ are defined, for any $i \in [n]$ and $\ell \in L$. Now, since we have $\mathit{st}(\alpha_1\beta_k, \ell^*) = \mathit{st}(\alpha_2\beta_k, \ell^*)$ by induction, we get that

$$\mathit{st}(\alpha_1\beta_{k+1}, i^*) = \mathit{st}(\alpha_1\beta_k e_{k+1}, i^*) = \mathit{st}(\alpha_2\beta_k e_{k+1}, i^*) = \mathit{st}(\alpha_2\beta_{k+1}, i^*),$$

$$\mathit{st}(\alpha_1\beta_{k+1}, \ell^*) = \mathit{st}(\alpha_1\beta_k e_{k+1}, \ell^*) = \mathit{st}(\alpha_2\beta_k e_{k+1}, \ell^*) = \mathit{st}(\alpha_2\beta_{k+1}, \ell^*).$$

The state of any process in $\mathit{procs}(\beta)$ other than p_{i^*} does not change, and the value of any register in $\mathit{acc}(\beta)$ other than ℓ^* does not change. Thus, we have $\forall i \in \mathit{procs}(\beta) : \mathit{st}(\alpha_1\beta_{k+1}, i) = \mathit{st}(\alpha_2\beta_{k+1}, i)$ and $\forall \ell \in \mathit{acc}(\beta) : \mathit{st}(\alpha_1\beta_{k+1}, \ell) = \mathit{st}(\alpha_2\beta_{k+1}, \ell)$. So, Equation 4.5 holds for $k + 1$, and the lemma holds by induction. $\square$

We define the following notation for a permutation $\pi \in S_n$. We will write a permutation π as $(\pi_1, \pi_2, \ldots, \pi_n)$, meaning that 1 maps to π_1 under π, 2 maps to π_2, etc. We write $\pi^{-1}(i)$ for the element that maps to i under π, for $i \in [n]$. We write $i \leq_\pi j$ if $\pi^{-1}(i) \leq \pi^{-1}(j)$; that is, i equals j, or i comes before j in π. If $S \subseteq [n]$, we write $\min_\pi S$ for the minimum element in S, where elements are ordered by $\leq_\pi$.

Let M be a set, and let $\preceq$ be a partial order on the elements of M. We can think of $\preceq$ equivalently as a relation or a set. That is, if $i, j \in M$, then $i \preceq j$ if and only if $(i, j) \in \preceq$. Depending on the context, one notation may be more convenient than the other. If $\leq$ is a total order on the elements of M, then we say that $\leq$ is *consistent* with $\preceq$ if, for any $i, j \in M$ such that $i \preceq j$, we have $i \leq j$. Let $N \subseteq M$. Then we say N is a *prefix* of $(M, \preceq)$ if whenever we have $m_1, m_2 \in M$, $m_2 \in N$ and $m_1 \preceq m_2$, we also have $m_1 \in N$. We define $\min(M, \preceq) = \{\mu \,|\, (\mu \in M) \wedge (\nexists \mu' \in M : \mu' \prec \mu)\}$ and $\max_\preceq M = \{\mu \,|\, (\mu \in M) \wedge (\nexists \mu' \in M : \mu \prec \mu')\}$ to be the set of minimal and maximal elements in M, with respect to $\preceq$. Note that we define $\min(\emptyset, \preceq) = \max_\preceq \emptyset = \emptyset$.

For any set M, we define $\diamond(M)$ to be M if $|M| \neq 1$, and we define it to be m, if $M = \{m\}$. That is, the diamond extracts the unique element in M, if M is a singleton set, and otherwise does nothing. We define $\min_{\preceq} M = \diamond(\min(M, \preceq))$. Thus, $\min_{\preceq} M$ is the set of minimal elements in M, if there is more than one minimal element, or no elements in M. If M contains a minimum element, then $\min_{\preceq} M$ is simply that element. Note that $\max_{\preceq} M$ and $\min_{\preceq} M$ are defined somewhat differently, in that $\max_{\preceq} M$ always returns a set, while $\min_{\preceq} M$ can return a set or an element. We adopt this convention because it leads to somewhat simpler notation later.

4.3.2 The Mutual Exclusion Problem

Given a shared memory algorithm $\mathcal{A}$, we say that $\mathcal{A}$ is a *mutual exclusion algorithm* if each process p_i can perform, in addition to its read and write steps, the following *critical steps*: $\mathsf{try}_i, \mathsf{enter}_i, \mathsf{exit}_i, \mathsf{rem}_i$. For any critical step $e \in \{\mathsf{try}_i, \mathsf{enter}_i, \mathsf{exit}_i, \mathsf{rem}_i\}_{i \in [n]}$, we define $type(e) = \mathsf{C}$. We define $reg(e) = \perp$. We will assume that the only steps that p_i can perform are its read, write and critical steps. That is, we assume that

$$E_i = \{\mathsf{try}_i, \mathsf{enter}_i, \mathsf{exit}_i, \mathsf{rem}_i\} \cup \bigcup_{\ell \in L, v \in V} \{\mathsf{read}_i(\ell), \mathsf{write}_i(\ell, v)\}.$$

Given a run $\alpha \in runs(\mathcal{A})$, we say a process p_i is in its *trying section* after α if its last critical step in α is try_i. We say it is in its *critical section* after α if the last critical step is enter_i. We say it is in its *exit section* after α if the last critical step is exit_i. Finally, we say it is in its *remainder section* after α if the last critical step is rem_i, or p_i performs no critical steps in α. Intuitively, a try_i step is an indication by p_i that it wants to enter the critical section. An enter_i step indicates that p_i has entered the critical section, and exit_i indicates that p_i has exited the critical section. Finally, a rem_i step indicates that p_i has finished performing all the cleanup actions needed to ensure that another process can safely enter the critical section.

We now define a *fairness* condition on runs of a mutual exclusion algorithm. The condition roughly says that a run is fair if for every process, either the process ends in a state where it does not want to enter the critical section, or, if the process wants to enter the critical section infinitely often in the run, then it is given infinitely many steps to do so. Formally, we have the following.

Definition 4.3.2 *Let $\alpha = e_1 e_2 \ldots \in runs(\mathcal{A})$. Then we say α is* fair *if for every process $i \in [n]$, we have the following.*

1. *If α is finite, then p_i is in its remainder section at the end of α.*

2. *If α is infinite, then one of the following holds.*

   (a) p_i takes no steps in α.

(b) There exists a $j \geq 1$ such that $e_j = \mathsf{rem}_i$, and for all $k > j$, we have $proc(e_k) \neq i$.

(c) p_i takes an infinite number of steps in α.

We define $fair(\mathcal{A})$ to be the set of fair runs of $\mathcal{A}$.

We now define the correctness property for a mutual exclusion algorithm.

Definition 4.3.3 *We say that a mutual exclusion algorithm $\mathcal{A}$ solves the* mutual exclusion problem *if any run $\alpha = e_1 e_2 \ldots \in runs(\mathcal{A})$ satisfies the following properties.*

- ***Well Formedness**: Let p_i be any process, and consider the subsequence γ of α consisting only of p_i's critical steps. Then γ forms a prefix of the sequence $\mathsf{try}_i \circ \mathsf{enter}_i \circ \mathsf{exit}_i \circ \mathsf{rem}_i \circ \mathsf{try}_i \circ \mathsf{enter}_i \circ \mathsf{exit}_i \circ \mathsf{rem}_i \ldots$[4].*
- ***Mutual Exclusion**: For any $t \geq 1$, and for any two processes $p_i \neq p_j$, if the last occurrence of a critical step by p_i in $\alpha(t)$ is enter_i, then the last critical step by p_j in $\alpha(t)$ is* not *enter_j.*
- ***Progress**: Suppose $\alpha \in fair(\mathcal{A})$, and suppose there exists $j \geq 1$ such that $(\forall k \geq j)(\forall i \in [n])$: $e_k \neq \mathsf{try}_i$. Then α is finite.*

The well formedness condition says that every process behaves in a syntactically correct way. That is, if a process wishes to enter the critical section, it first enters its trying section, then enters the critical section, exits, and finally enters its remainder section after it has performed all its cleanup actions. The mutual exclusion property says that no two processes can be in their critical sections at the same time. The progress property says that in any fair run α, if there is a point in α beyond which no processes try to enter the critical section, then α is finite. By Definition 4.3.2, this means that all processes that want to enter the critical section in α do so, and finish in their remainder sections.

Our definition of progress is slightly different from the typical *livelock-freedom* or *starvation-freedom* progress properties for mutual exclusion. If a set of processes try to enter the critical section, then livelock-freedom requires that after a sufficiently large number of steps, *some* process finishes its critical and remainder sections; starvation-freedom requires that *every* process finishes its critical and remainder sections. Note that livelock-freedom is a weaker property than starvation-freedom. Since we only consider canonical executions in which each process tries to enter the critical section once, then we can see that any mutual exclusion algorithm satisfying livelock-freedom will also satisfy our progress property, in canonical executions. Thus, a lower bound for algorithms

[4]Note that strictly speaking, $\mathcal{A}$ cannot *guarantee* well formedness, but merely *preserve* it. This is because, typically, the steps try_i and exit_i (for $i \in [n]$) are regarded as inputs from the environment. Thus, $\mathcal{A}$ can only ensure well formedness if the environment executes try_i and exit_i in an alternating manner. For our lower bound, we have adversarial control over the environment, and will guarantee that try_i and exit_i occur in alternating order. Thus, we can now require that $\mathcal{A}$ guarantees well formedness. For further discussion about environment-controlled steps, please see the end of this section.

satisfying our progress property in canonical executions implies the same lower bound for lower bound for algorithms satisfying livelock or starvation freedom. We work with our definition of progress because it fits more conveniently with our proof.

We now define a set of runs $\mathcal{C}$, which we call the *canonical runs*. Our lower bound shows that for any algorithm $\mathcal{A}$ solving the mutual exclusion problem, there exists some $\alpha \in \mathcal{C} \cap fair(\mathcal{A})$ such that α has $\Omega(n \log n)$ cost in the state change model. $\mathcal{C}$ consists of runs in which each process $p_1, \ldots, p_n$ completes the critical section exactly once. In addition, no process lingers in the critical section: a process that enters the critical section exits in its next step.

Definition 4.3.4 (Canonical Runs) *Let $\mathcal{A}$ be an algorithm solving the mutual exclusion problem, and let $\alpha = e_1 e_2 \ldots \in fair(\mathcal{A})$. Then α is a* canonical run *if it satisfies the following properties.*

1. *For every $i \in [n]$, try_i occurs exactly once in α, and it is the first step of process p_i in α.*
2. *For any $i \in [n]$, if $e_j = \mathsf{enter}_i$ for some $j \geq 1$, then $e_k = \mathsf{exit}_j$, where k is the minimum integer κ larger than j such that $proc(e_\kappa) = i$.*

We define $\mathcal{C}$ to be the set of canonical runs of $\mathcal{A}$.

The reason we study canonical runs is that they focus exclusively on the cost of the synchronization needed between processes to achieve mutual exclusion. Indeed, since all the processes try to enter the critical section in a canonical run, and since they try to enter in a "balanced" way (*i.e.*, all processes try to enter the same number of times), then it creates a situation requiring maximal synchronization and maximal time for completion. Also, since a process immediately exits the critical section after entering, all the costs in a canonical run can be attributed to the cost of synchronization.

Finally, we discuss a subtle issue regarding the modeling of critical steps. Consider any process p_i. Then the steps enter_i and rem_i are *enabled* by p_i. That is, p_i decides, using the function $\delta(\cdot, i)$, when it wants to enter the critical and remainder sections. On the other hand, the steps try_i and exit_i are typically modeled as inputs from the *environment*. That is, we imagine that there is an external "user", for example, a thread in a multithreaded computation, that "causes" p_i to execute try_i, so that the thread can obtain exclusive access to a resource. If p_i manages to enter the critical section on behalf of the thread (*i.e.*, p_i enables enter_i), then the thread may later relinquish the resource by causing p_i to execute exit_i. Since we are proving a lower bound for canonical runs, we want to ensure that if enter_i occurs, then exit_i also occurs, as soon as possible (in p_i's next step). In addition, for the purposes of our lower bound, it suffices to assume that p_i itself can enable its try_i and exit_i steps. We model our requirements is as follows. First, we assume that $\delta(\hat{s}_0^i, i) = \mathsf{try}_i$. That is, we assume that the first step that p_i wants to execute in any run is try_i. Next, let $s_i \in S_i$, and suppose that $\delta(s_i, i) = \mathsf{enter}_i$. Then we assume that for all $s \in S$ such that $st(s, i) = s_i$, we have $\Delta(s, \mathsf{enter}_i, i) = s'_i$, such that $\delta(s'_i, i) = \mathsf{exit}_i$. That is, if s_i is a state of p_i in which it wants to execute enter_i, then in any state s'_i of p_i after p_i executes enter_i, p_i wants to execute exit_i.

4.3.3 The State Change Cost Model

In this section, we define the state change cost model for measuring the cost of a shared memory algorithm. In [1], it was proven that the cost of any shared memory mutual exclusion algorithm is unbounded if we count every shared memory access. To obtain a meaningful measure for cost, researchers have focused on models in which some memory accesses are discounted (assigned zero or unit cost). Two important models that have been studied are the *distributed shared memory (DSM)* model and the *cache coherent (CC)* model [5, 29, 4]. The main feature of both of these models is that, during the course of a run, a register is sometimes considered *local* to a process[5]. Any access by a process to its local registers is free. This is intended to model a situation in hardware in which a piece of memory and a processor are physically located close together, making accesses to that memory very efficient. A generic algorithm in the DSM or CC model works by reading and writing to registers, and also *busywaiting* on some registers. The latter operation means that a process continuously reads some registers, evaluating some predicate on the values of those registers after each read. The process is stuck in a loop while it is busywaiting, and only breaks out of the loop when the busywaiting predicate is satisfied. As long as a process busywaits on local registers, all the reads done during the busywait have a combined *constant* cost[6].

In this chapter, we define a new cost model, called the *state change (SC)* cost model, which is related to the DSM and CC models. The state change cost model charges an algorithm only for steps that change the system state. In particular, we charge the algorithm a unit cost for each write performed by a process[7]. If a process performs a read step and changes its state after the read, then the algorithm is charged a unit cost. If the process does not change its state after the read, the algorithm is not charged. This charging scheme in effect allows a process to busywait on one register at unit cost. For example, suppose the value of a register ℓ is currently 0, and a process p_i repeatedly reads ℓ, until its value becomes 1. As long as ℓ's value is not 1, p_i does not change its state, and thus, continues to read ℓ. If ℓ eventually becomes 1, then the algorithm is charged one unit for all reads up to when p_i reads ℓ as 1. The difference between the state change and the CC or DSM model is that a process in the CC or DSM model could potentially busywait on *several* registers at unit cost. For example, in the CC model, a process can busywait on all its registers, until the

[5]The DSM and CC models differ in how they define locality. In DSM, each process has a fixed set of local variables. In CC, a variable can be local to different processes at different times.

[6]The busywaiting reads do not have zero cost, because typically the registers being busywaited on have to be made local to the busywaiting process, *e.g.* by moving some register values from main memory to a processor's local cache. The move operation is assigned unit cost.

[7]We can show that for any algorithm solving the mutual exclusion problem, a process must change its state after performing a write step. Roughly speaking, this is because if a process does not change its state after a write, then it may stay in the same writing state forever, violating the progress property of mutual exclusion. We show formally in Lemma 4.7.8 that a writing process changes its state.

first one of them satisfies the process's busywaiting predicate. It is not clear what additional power the ability to busywait on multiple registers gives an algorithm. In fact, in almost all algorithms designed for the DSM and CC models, processes busywait on one variable at a time. The mutual exclusion algorithm of Yang and Anderson [40] is one such algorithm, and it incurs $O(n \log n)$ cost in all canonical runs in the SC cost model. Formally, the system state change cost model is defined as follows.

Definition 4.3.5 (System State Change Cost Model) *Let $\mathcal{A}$ be an algorithm, and let $\alpha = e_1 e_2 \ldots e_t \in runs(\mathcal{A})$ be a finite run.*

1. *Let $j \in [t]$, and define $sc(\alpha, j) = 1$ if $st(\alpha(j-1)) \neq st(\alpha(j))$, and $sc(\alpha, j) = 0$ otherwise.*

2. *We define the (system state change)* cost *of run α to be $C^s(\alpha) = \sum_{j \in [t]} sc(\alpha, j)$.*

While charging an algorithm for steps that change the system state is a natural cost measure, it turns out to be more convenient in our proofs to charge the algorithm for steps that change the state of *some process*. Thus, we define the following.

Definition 4.3.6 (Process State Change Cost Model) *Let $\mathcal{A}$ be an algorithm, and let $\alpha = e_1 e_2 \ldots e_t \in runs(\mathcal{A})$ be a finite run.*

1. *Let p_i be a process, and let $j \in [t]$. We define $sc(\alpha, i, j) = 1$ if $st(\alpha(j-1), i) \neq st(\alpha(j), i)$, and $sc(\alpha, i, j) = 0$ otherwise.*

2. *We define the (process state change)* cost *of run α to be $C^p(\alpha) = \sum_{j \in [t]} \sum_{i \in [n]} sc(\alpha, i, j)$.*

Since a system state contains the state of each process, then it is easy to see that $C^p(\alpha) \leq C^s(\alpha)$, for all $\alpha \in runs(\mathcal{A})$. Thus, a cost lower bound for the process state change model implies the same lower bound for the system state change model. In the remainder of this paper, we will only work with the process state change cost model. We write $C(\alpha) \equiv C^p(\alpha)$, for any run $\alpha \in runs(\mathcal{A})$.

In Table 4-1, we provide a summary of the notation we have introduced. The table also includes all the notation introduced in later parts of the chapter.

4.4 Overview of the Lower Bound

In this section, we give a detailed overview of our lower bound proof. For the remainder of this paper, fix $\mathcal{A}$ to be any algorithm solving the mutual exclusion problem. The proof consists of three steps, which we call the *construction step*, the *encoding step*, and the *decoding step*. The construction step builds an equivalence class of finite runs $A_\pi \subseteq runs(\mathcal{A})$ for each permutation $\pi \in S_n$, such that for permutations $\pi_1 \neq \pi_2$, we have $A_{\pi_1} \cap A_{\pi_2} = \emptyset$. All runs in A_π have the same state change cost C_π. The encode step produces a string E_π of length $O(C_\pi)$ for each A_π,. The decode step reproduces an

Notation	**Location of definition**
$n, p_1, \ldots, p_n, \mathcal{A}$	Page 48
V, L, v, ℓ	Page 48
$S, S_i, \hat{s}_0, \hat{s}_0^i, s, st(s,i), st(s,\ell)$	Page 48
E, E_i, read and write steps, $\mathsf{read}_i(\ell), \mathsf{write}_i(\ell, v)$	Page 48
$proc(e), reg(e), val(e), type(e)$	Page 48
$\delta(s_i, i), \delta(\alpha, i), \Delta(s, e_i, i), \Delta(\alpha, e_i, i)$	Page 49
α, execution, $execs(\mathcal{A})$, extension, $len(\alpha)$	Page 49
$st(\alpha, i), st(\alpha, \ell), \delta(\alpha, i)$	Page 49
run, $run(\alpha)$, $runs(\mathcal{A})$, step sequence, $spseq(\mathcal{A})$, $acc(\alpha)$, $procs(\alpha)$, $\alpha \circ \beta$	Pages 49-50
$\pi, \pi^{-1}, i \leq_\pi j, \min_\pi S$	Page 51
$i \preceq j$, $(i,j) \in \preceq$, prefix, consistent total order	Page 51
$\min(S, \preceq), \max_\preceq S, \min_\preceq S, \diamond(S)$	Pages 51-52
critical steps, $\mathsf{try}_i, \mathsf{enter}_i, \mathsf{exit}_i, \mathsf{rem}_i$	Page 52
trying, critical, exit, remainder sections	Page 52
fair run, $fair(\mathcal{A})$, canonical runs, $\mathcal{C}$	Definitions 4.3.2, 4.3.4
mutual exclusion algorithm, well formedness, mutual exclusion, progress	Definition 4.3.3
$C^s(\alpha), C^p(\alpha), C(\alpha)$	Definitions 4.3.5, 4.3.6, page 56
metastep, $\mathcal{M}$, attributes of metasteps	Definition 4.5.1
Construct algorithm, $\langle r \rangle$	Figure 4-4, page 63
iteration, $\iota, \mathcal{I}, \iota \oplus 1, \iota \ominus 1, \iota^+, \iota^-, \iota \oplus r, \iota \ominus r, j_i, \iota^n$	Page 67
$M_\iota, \preceq_\iota, \check{m}_\iota, e_\iota, \alpha_\iota, N_\iota, R_\iota, R_\iota^*, W_\iota, W_\iota^s$	Definition 4.6.1
version of metastep, m^ι, N^ι	Definition 4.6.2
critical/read/write create iteration, read/write modify iteration	Page 69
execution γ and output α of Lin, γ order of N, γ order of m, $\text{Lin}(N^\iota, \preceq_\iota)$	Page 70
$\Phi(\iota, N), \Phi(\iota, N, k), \phi(\iota, N), \phi(\iota, N, k)$	Definition 4.6.7
$\Psi(\iota, \ell), \Psi^w(\iota, \ell), \Upsilon(\iota, \ell, m), \Upsilon(\iota, m), acc(N)$	Definitions 4.6.15, 4.6.16, page 79
$G((M_\iota)^\iota), G, \mathcal{L}(\iota, N), \lambda(\iota, N, k), \lambda(\iota, N)$	Definitions 4.7.1, 4.7.2, 4.7.3
next π_k step/metastep after (ι, N), v-reads ℓ after (ι, N)	Definition 4.7.4
$readers(\iota, N, \ell, v)$, $wwriters(\iota, N, \ell)$, $preads(\iota, N, \ell)$, unmatched preread	Definition 4.7.4, 4.7.5
extended type, $\mathcal{T}$, $xtype(e, m)$	Definition 4.8.1
Encode algorithm	Figure 4-5
Decode algorithm	Figure 4-7
iteration of Decode, $\langle r \rangle_{\text{D}}$, ϑ, state of ϑ, σ, $\sigma.x$, N-correct	Page 126, Definition 4.11.1

Figure 4-1: Summary of the notation in this chapter and the location of their definitions.

$\alpha_\pi \in A_\pi$ using only input E_π. Since different A_π's are disjoint, each E_π uniquely identifies one of $n!$ different permutations. Thus, there exists some $\pi \in S_n$ such that E_π has length $\Omega(n \log n)$. Then, the run α_π corresponding to this E_π must have cost $\Omega(n \log n)$.

Fix a permutation $\pi = (\pi_1, \ldots, \pi_n) \in S_n$. We say that a process p_i has *lower (resp., higher) index* (in π) than process p_j if i comes before (resp., after) j in π, *i.e.* $i <_\pi j$ (resp., $j <_\pi i$). For ease of exposition, we will describe the construction step twice, first at a high level, and in a slightly inaccurate way, to convey the general idea, then subsequently in an accurate and more detailed way. In the high level description, we will pretend that each equivalence A_π consists of only one run α_π. Then, in the construction step, we build in n stages n different finite runs, $\alpha_1, \ldots, \alpha_n \in runs(\mathcal{A})$, where $\alpha_n = \alpha_\pi$. In each α_i, only the first i processes in the permutation, $p_{\pi_1}, \ldots, p_{\pi_i}$, take steps. Thus, α_1 is a solo run by process p_{π_1}. Each process runs until it has completed its trying, critical and exit sections once. We will show that the processes in α_i complete their critical sections in the order given by π, that is, p_{π_1} first, then p_{π_2}, etc., and finally, p_{π_i}. Next, we construct run α_{i+1} in which process $p_{\pi_{i+1}}$ also takes steps, until it completes its trying, critical, and exit sections. α_{i+1} is constructed by starting with α_i, and then inserting steps by $p_{\pi_{i+1}}$, in such a way that $p_{\pi_{i+1}}$ is *not seen* by any of the lower indexed processes $p_{\pi_1}, \ldots, p_{\pi_i}$. Roughly speaking, this is done by placing some of $p_{\pi_{i+1}}$'s writes immediately before writes by lower indexed processes, so that the latter writes overwrite any trace of $p_{\pi_{i+1}}$'s presence.

The preceding paragraph described some of the intuition for the construction step. It was inaccurate because it constructed only one run α_π, instead of a class of runs A_π. We now give a more detailed and accurate description of the construction step. Instead of directly generating a run α_i in stage i, we actually generate a set of *metasteps* M_i and a partial order $\preceq_i$ on M_i in stage i. Roughly speaking, a metastep consists of two sets of steps, the *read* steps and the *write* steps, and a distinguished step among the write steps that we call the *winning* step[8]. All steps access the same register, and each process performs at most one step in a metastep. We say a process *appears* in the metastep if it takes a step in the metastep, and we say the *winner* of the metastep is the process performing the winning step. The purpose of a metastep is to hide, from every process $p_1, \ldots, p_n$, the presence of all processes appearing in the metastep, except possibly the winner.

Given a set of metasteps M_i and a partial order $\preceq_i$ on M_i, we can generate a run from $(M_i, \preceq_i)$ by first ordering M_i using *any* total order consistent with $\preceq_i$, to produce a sequence of metasteps. Then, for each metastep in the sequence, we expand the metastep into a sequence of steps, consisting of the non-winning write steps of the metastep, ordered arbitrarily, followed by the winning step, followed by the read steps, ordered arbitrarily. Notice that this sequence hides the presence of all processes except possibly the winner. That is, if a process p_i did not see another process p_j before the metastep sequence, then p_i does not see p_j after the metastep sequence either, unless p_j is the winner of the metastep sequence. The overall sequence of steps resulting from totally ordering M_i, and then expanding each metastep, is a run which we call a *linearization* of $(M_i, \preceq_i)$. Of course, there may be many total orders consistent with $\preceq_i$, and many ways to expand each metastep, leading to many different linearizations. However, we will show that for the particular M_i and $\preceq_i$ we construct, all linearizations are essentially "the same". For example, at the end of all linearizations, all processes have the same state, and all registers have the same values. Also, in all linearizations, the processes $p_{\pi_1}, \ldots, p_{\pi_i}$ each complete their critical sections once, and they do so in that order. It is the set M_n and partial order $\preceq_n$, generated at the end of stage n in the construction step, that we eventually encode in the encoding step. The set A_π is the set of all possible linearizations of $(M_n, \preceq_n)$[9]. We show that all linearizations of $(M_n, \preceq_n)$ have the same (state change) cost, and we call this cost C_π.

The reason we construct a partial order of metasteps instead of constructing a run, *i.e.*, a total

[8]A metastep actually has other properties which are described in detail in Section 4.5. However, the current simplified description of a metastep will suffice for this proof overview.

[9]However, as stated, we do not directly encode A_π, but rather, encode $(M_n, \preceq_n)$.

ordering of steps, is that the partial order $\preceq_n$ on the metasteps of M_n contains fewer orderings between the steps contained in (all the metasteps of) M_n than a total ordering on the steps contained in M_n. In fact, the orderings contained in $\preceq_n$ can be seen as representing precisely the information acquired by $p_1, \ldots, p_n$ in the course of a run produced by linearizing $(M_n, \preceq_n)$. It is because of this that we can encode $(M_n, \preceq_n)$ using a string with length proportional to C_π.

We now describe the encoding step. This step produces a string E_π, from input $(M_n, \preceq_n)$. For any process p_i, we show that all the metasteps containing p_i in M_n are totally ordered in $\preceq_n$. Thus, for any metastep containing p_i, we can say the metastep is p_i's *j'th metastep*, for some j. The encoding algorithm uses a table with n columns and an infinite number of rows. In the j'th row and i'th column of the table, which we call cell $T(i, j)$, the encoder records what process p_i does in its j'th metastep. However, to make the encoding short, we only record, roughly speaking, the *type*, either read, write or critical, of the step that p_i performs in its j'th metastep. That is, we simply record a symbol R, W or C[10]. In addition, if p_i is the winner of the metastep, we also record a *signature* of the entire metastep. The signature basically contains a *count* of how many processes in the metastep perform read steps, and how many perform write steps (including the winning step). Note that the signature does not specify *which* processes read or write in the metastep, nor the register or value associated with any step. Now, if there are k processes involved in a metastep, the total number of bits we use to encode the metastep is $O(k) + O(\log k) = O(k)$. Indeed, for each non-winner process in the metastep, we use $O(1)$ bits to record its step type. For the winner process, we record its step type, and use $O(\log k)$ bits to record how many readers and writers are in the metastep. We can show that the state change cost to the algorithm for performing this metastep is k. In particular, each read and write step in the metastep causes a state change. Informally, this shows that the size of the encoding is proportional to the cost incurred by the algorithm. The final encoding of $(M_n, \preceq_n)$ is formed by iterating over all the metasteps in M_n, each time filling the table as described above. Then, we concatenate together all the nonempty cells in the table into a string E_π.

Lastly, we describe how, using E_π as input, the decoding step constructs a run α_π that is a linearization of $(M_n, \preceq_n)$[11]. Roughly speaking, at any time during the decoding process, the decoder algorithm has produced a linearization of a *prefix* N of $(M_n, \preceq_n)$. Recall that N is a prefix of $(M_n, \preceq_n)$ if $N \subseteq M_n$, and whenever $m \in N$ and $m' \preceq_n m$, then $m' \in N$ as well. We say all metasteps in N have been *executed*. The linearization of N is a prefix α (in the normal sense) of run α_π. Using N and E_π, the decoder tries to find a minimal (with respect to $\preceq_n$) unexecuted metastep m, *i.e.*, a minimal metastep not contained in N. The decoder executes m, by linearizing m and appending the steps to α. After doing this, the decoder has executed prefix $N \cup \{m\}$; the decoder

[10] We sometimes also use a fixed set of other symbols, such as PR or SR, to represent the type of a metastep. This is described in detail in Section 4.8. For the purposes of this proof overview, our current simplified description suffices.

[11] Note that even though our discussion involves π, the decoder does *not* know π. The only input to the decoder is the string E_π.

then restarts the decoding loop.

To find a minimal unexecuted metastep, the decoder applies the step functions $\{\delta(\alpha, i)\}_{i\in[n]}$ of $\mathcal{A}$ to the prefix α to compute each process p_i's next step after α. This is the step that p_i takes in the minimum unexecuted metastep containing p_i. We call this metastep p_i's *next* metastep, and denote it by m_i. m_i may be different for different i. Let $\lambda = \{m_i\}_{i\in[n]}$ be the set of next metasteps for all processes $p_1, \ldots, p_n$. Note that not every metastep in λ is necessarily a minimal unexecuted metastep (rather, it is only the minimum unexecuted metastep containing a particular process). However, we show that there exists some $m \in \lambda$ that *is* a minimal unexecuted metastep. The decoder does not directly know λ or m. Rather, the decoder only knows the next step of each process after α. In order to deduce m, the decoder reads E_π. Suppose the decoder finds a signature in column i of E_π, and the signature indicates there are r reads and w writes in the metastep corresponding to the signature. Suppose also that p_i's next step accesses register ℓ. Then the decoder will know the following.

- p_i's next metastep m_i accesses ℓ.
- p_i is the winner of m_i.
- There are r readers, and $w - 1$ other writers besides p_i that appear in m_i.

The decoder looks at the next step that each process will perform, and checks whether there are indeed r processes whose next step is a read on ℓ, and $w - 1$ processes besides p_i whose next step is a write to ℓ[12]. Suppose this is the case. Then, these next steps on ℓ are precisely the steps contained in a minimal unexecuted metastep. That is, $m_i \in \lambda$ is a minimal unexecuted metastep, and the steps contained in m_i are the next steps that access ℓ. The decoder executes m_i, by appending the r next read steps and w next write steps on ℓ to the current run, placing all the writes before all the reads, and placing the winning write by p_i last among the writes. Having done this, the decoder has completed one iteration of the decoding loop. The decoder proceeds to the next iteration, and continues until it has read all of E_π. We can summarize the decoding algorithm as follows.

1. The decoder computes the next step that each process will take, based on the current run the decoder has generated.
2. The decoder reads E_π to find signatures of unexecuted metasteps.
3. If the signature for a register ℓ is *filled*, *i.e.*, the *number* of processes whose next step reads or writes to ℓ matches the numbers indicated by the signature, then these steps are equal to the steps in some minimal unexecuted metastep m.

[12]Actually, the decoder also checks whether the number of *prereads* matches the number indicated by the signature. Prereads are discussed in Section 4.5. Section 4.10 describes the decoding algorithm in more detail. For this overview, our simplified presentation suffices to convey the main ideas for the decoding.

4. The decoder linearizes m and appends the steps to the current run. Then the decoder begins the next iteration of the decoding loop, or terminates, if it has read all of E_π.

The run α_π that the decoder produces after termination is a linearization of $(M_n, \preceq_n)$. As stated earlier, α_π can be used to uniquely identify π. Hence, E_π also identifies π. Thus, there must exist some $\pi \in S_n$ such that $|E_\pi| = \Omega(n \log n)$. Since $|E_\pi| = O(C(\alpha_\pi))$, then the state change cost of α_π is $\Omega(n \log n)$.

4.5 The Construction Step

4.5.1 Preliminary Definitions

In this section, we present the algorithm for the construction step. For the remainder of this chapter, fix $\mathcal{A}$ to be any algorithm solving the mutual exclusion problem.

Recall from our discussion in Section 4.4 that a metastep is, roughly speaking, a set of steps, all performed by different processes and accessing the same register, whose aim is to hide the presence of all but at most one of the processes taking part in the metastep. More precisely, a metastep has one of three types: *read*, *critical*, or *write*. A read (resp., critical) metastep contains only one step, which is a read (resp., critical) step. Notice that since a read or critical step does not change the value of any registers, it does not reveal the presence of any process (that is not already revealed). A write metastep may contain read and write steps. It always contains a write step, which we call the *winning step*. A write metastep can only reveal the presence of the process, called the *winner*, performing the winning step. In addition to containing read and write steps, a write metastep m may be associated with a set of read *metasteps*, which we call the *preread set* of m. The (read steps in the) metasteps in the preread set of m are not actually contained in m. Rather, the association of *preads*(m) to m is based on the fact that in the partial ordering on metasteps that we create, the preread metasteps of m are always ordered before m. We now formalize the preceding description.

Definition 4.5.1 (Metastep) *A* metastep *is identified by a label $m \in \mathcal{M}$, where $\mathcal{M}$ is an infinite set of labels. For any metastep m, we define the following attributes.*

1. *We let type$(m) \in \{\texttt{R}, \texttt{W}, \texttt{C}\}$. If type$(m) = \texttt{R}$ (resp., $\texttt{W}$, $\texttt{C}$), we say m is a* read *(resp.,* write*,* critical*) metastep.*

2. *If type$(m) = \texttt{C}$, then crit(m) is a singleton set containing a critical step of some process.*

3. *If type$(m) = \texttt{R}$, then reads(m) is a singleton set containing a read step of some process.*

4. *If type$(m) = \texttt{W}$, then we define the following attributes for m.*

(a) *$reads(m)$ is a set of read steps, $writes(m)$ and $win(m)$ are sets of write steps, and $|win(m)| = 1$. $reads(m)$ is called the* read steps *contained in m, $writes(m)$ is called the (non-winning)* write steps *contained in m, and $\diamond(win(m))$*[13] *is called the* winning step *in m.*

(b) *$reads(m)$, $writes(m)$ and $win(m)$ are mutually disjoint.*

(c) *All steps in $reads(m) \cup writes(m) \cup win(m)$ access the same register, and any process performs at most one step in $reads(m) \cup writes(m) \cup win(m)$.*

(d) *$readers(m)$ is the set of processes performing the steps in $reads(m)$, and is called the* readers *of m. $writers(m)$ is the set of processes performing the steps in $writes(m)$, and is called the* writers *of m. $winner(m)$ is the singleton set containing the process performing the step in $win(m)$. We call $\diamond(winner(m))$ the* winner *of m.*

(e) *We say that any process $i \in readers(m) \cup writers(m) \cup winner(m)$* appears *in m. For idiomatic reasons, we also sometimes say that such a process is* contained *in m.*

(f) *We say the* value *of m, written $val(m)$, is the value written by the step in $win(m)$.*

5. *If m is a read (resp., critical) metastep, then we let $steps(m)$ be the singleton set containing the read (resp., critical) step in m, and we let $procs(m)$ be the singleton set containing the process performing the step in m.*

6. *If m is a write metastep, then we let $steps(m) = reads(m) \cup writes(m) \cup win(m)$ be the set of all steps contained in m, and we let $procs(m) = readers(m) \cup writers(m) \cup winner(m)$ be the set of all processes appearing in m.*

7. *If the steps in m access a register (that is, if $type(m) \in \{\mathtt{R}, \mathtt{W}\}$), we let $reg(m)$ be the register accessed by these steps, and we say m* accesses *$reg(m)$. For idiomatic reasons, we also sometimes say m is a metastep* on *$reg(m)$.*

8. *For any $i \in procs(m)$, we write $step(m, i)$ for the step that process p_i takes in m.*

9. *If $type(m) = \mathtt{W}$, we let $preads(m)$ be a set of read metasteps, and we call this the* preread set *of m. If a (read) metastep m is contained in the preread set of some other metastep, then we say m is a* preread metastep *(in addition to being a read metastep).*

10. *Regardless of the type of m, all the attributes listed above (*e.g. *$reads(m)$, $val(m)$, $preads(m)$, etc.) are defined for m. Each attribute is initialized to $\emptyset$, $\perp$, or a string, depending on the type of the attribute.*

[13]Recall that $\diamond(M) = m$, for any singleton set $M = \{m\}$.

Variable	**Type**
π	A permutation in S_n.
j	A process in $[n]$.
$M_i, i \in [n], M, R, R^*, W, W^s$	A set of metasteps.
$\preceq_i, i \in [n]$	A partial order on a set of metasteps.
$m, \check{m}, m_w, m_{ws}$	A metastep, or $\emptyset$.
α	A run of $\mathcal{A}$.
e	A step in E.
ℓ	A register in ℓ.

Figure 4-2: The types and meanings of variables used in Construct and Generate.

Procedure	**Input type(s)**	**Output type(s)**
Construct(π)	A permutation in S_n.	A set of metasteps, a p.o. on the set.
Seq(m)	A metastep.	A step sequence.
Lin($M, \preceq$)	A set of metasteps, a p.o. on the set.	A step sequence.
Plin($M, \preceq, m$)	A set of metasteps, a p.o. on the set, a metastep.	A step sequence.
SC(α, m, i)	A run, a metastep, a process.	A boolean.

Figure 4-3: Input and output types of procedures in Figure 4-4. We write "p.o." for partial order.

Given a metastep m, the attributes of m may change during the construction step. For example, at the beginning of the construction step, m may not contain any read or write steps. As the construction progresses, read and write steps may be added to m. However, whatever values its attributes have, the label (*i.e.*, name) of the metastep remains m.

Let M be a set of metasteps, and let $\preceq$ be a partial order on M. Then a *linearization* α of $(M, \preceq)$ is any step sequence produced by the procedure Lin($M, \preceq$), shown in Figure 4-4[14]. If $m \in M$, then we say m is *linearized* in α. Lin($M, \preceq$) works by first ordering the metasteps of M using any total order consistent with $\preceq$. Then it produces a sequence of steps from this sequence of metasteps, by applying the procedure Seq($\cdot$) to each metastep. Given a metastep m, Seq(m) returns a sequence of steps consisting of the write steps of m, then the winning step of m, then the read steps. It uses the (nondeterministic) helper function *concat*, which totally orders a set of steps, in an arbitrary order. The procedure Plin($M, \preceq, m$), where $m \in M$ is a metastep, works similarly to Lin($M, \preceq$), except that it only linearizes the metasteps in $\mu \in M$ such that $\mu \preceq m$.

4.5.2 The Construct Algorithm

In this section, we show how to create a set of metasteps M_i and a partial order $\preceq_i$ on M_i, for every $i \in [n]$, with the properties described earlier. For the remainder of this section, fix an arbitrary permutation $\pi \in S_n$. This is the input to Construct. For every $i \in [n]$, the only processes that take steps in any metastep of M_i are processes $p_{\pi_1}, \ldots, p_{\pi_i}$. In any linearization of $(M_i, \preceq_i)$, each

[14]Note that *a priori*, we do not know α is necessarily a run, *i.e.*, that α corresponds to an execution of $\mathcal{A}$. We prove in Section 4.6.4 that α is in fact a run.

process $p_{\pi_1}, \ldots, p_{\pi_i}$ completes its trying, critical, and exit section once. The construction algorithm is shown in Figure 4-4. Also, Figures 4-2 and 4-3 show the types of the variables used in Figure 4-4, and the input and return types of the procedures in Figure 4-4.

```
**procedure** Construct($\pi$)
  $M_0 \leftarrow \emptyset$;   $\preceq_0 \leftarrow \emptyset$
  **for** $i \leftarrow 1, n$ **do**
    $(M_i, \preceq_i) \leftarrow$ Generate$(M_{i-1}, \preceq_{i-1}, \pi_i)$
  **end for**
  **return** $M_n$, and the reflexive, transitive closure of $\preceq_n$

**procedure** Generate$(M, \preceq, j)$
  $m \leftarrow$ new metastep;   $crit(m) \leftarrow \{\mathsf{try}_j\}$;   $type(m) \leftarrow \mathtt{C}$
  $M \leftarrow M \cup \{m\}$;   $\check{m} \leftarrow m$
  **repeat**
    $\alpha \leftarrow$ Plin$(M, \preceq, \check{m})$;   $e \leftarrow \delta(\alpha, j)$;   $\ell \leftarrow reg(e)$
    **switch**
      **case** $type(e) = \mathtt{W}$:
        $W \leftarrow \{\mu \mid (\mu \in M) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \not\preceq \check{m})\}$
        $m_w \leftarrow \min_\preceq W$
        **if** $m_w \neq \emptyset$ **then**
          $writes(m_w) \leftarrow writes(m_w) \cup \{e\}$
          $\preceq \leftarrow \preceq \cup \{(\check{m}, m_w)\}$;   $\check{m} \leftarrow m_w$
        **else**
          $m \leftarrow$ new metastep;   $win(m) \leftarrow \{e\}$
          $reg(m) \leftarrow \ell$;   $type(m) \leftarrow \mathtt{W}$
          $M \leftarrow M \cup \{m\}$
          $R \leftarrow \{\mu \mid (\mu \in M) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{R}) \wedge (\mu \not\preceq \check{m})\}$
          $R^* \leftarrow \max_\preceq R$;   $preads(m) \leftarrow R^*$
          **for all** $\mu \in R^*$ **do**
            $\preceq \leftarrow \preceq \cup \{(\mu, m)\}$ **end for**
          $\preceq \leftarrow \preceq \cup \{(\check{m}, m)\}$;   $\check{m} \leftarrow m$
      **case** $type(e) = \mathtt{R}$:
        $W^s \leftarrow \{\mu \mid (\mu \in M) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \not\preceq \check{m}) \wedge \mathrm{SC}(\alpha, \mu, j)\}$
        $m_{ws} \leftarrow \min_\preceq W^s$
        **if** $m_{ws} \neq \emptyset$ **then**
          $reads(m_{ws}) \leftarrow reads(m_{ws}) \cup \{e\}$
          $\preceq \leftarrow \preceq \cup \{(\check{m}, m_{ws})\}$;   $\check{m} \leftarrow m_{ws}$
        **else**
          $m \leftarrow$ new metastep;   $reads(m) \leftarrow \{e\}$
          $reg(m) \leftarrow \ell$;   $type(m) \leftarrow \mathtt{R}$;   $M \leftarrow M \cup \{m\}$
          $\preceq \leftarrow \preceq \cup \{(\check{m}, m)\}$;   $\check{m} \leftarrow m$
        **end if**
      **case** $type(e) = \mathtt{C}$:
        $m \leftarrow$ new metastep;   $crit(m) \leftarrow \{e\}$;   $type(m) \leftarrow \mathtt{C}$
        $M \leftarrow M \cup \{m\}$;   $\preceq \leftarrow \preceq \cup \{(\check{m}, m)\}$;   $\check{m} \leftarrow m$
    **end switch**
  **until** $e = \mathsf{rem}_j$
  **return** $M$ and $\preceq$
**end procedure**

**procedure** Seq$(m)$
  **if** $type(m) \in \{\mathtt{W}, \mathtt{R}\}$ **then**
    **return** $concat(writes(m)) \circ win(m) \circ concat(reads(m))$
  **else return** $crit(m)$
**end procedure**

**procedure** Lin$(M, \preceq)$
  let $\leq^M$ be a total order on $M$ consistent with $\preceq$
  order $M$ using $\leq^M$ as $m_1, m_2, \ldots, m_u$
  **return** Seq$(m_1) \circ \ldots \circ$ Seq$(m_u)$
**end procedure**

**procedure** Plin$(M, \preceq, m)$
  $N \leftarrow \{\mu \mid (\mu \in M) \wedge (\mu \preceq m)\}$
  **return** Lin$(N, \preceq|_N)$
**end procedure**

**procedure** SC$(\alpha, m, i)$
  $\ell \leftarrow reg(m)$; $v \leftarrow val(m)$
  choose $s \in S$ s.t. $(st(s, i) = st(\alpha, i)) \wedge (st(s, \ell) = v)$
  **return** $\Delta(s, \mathsf{read}_i(\ell), i) \neq st(\alpha, i)$
**end procedure**
```

Figure 4-4: Stage i of the construction step.

The procedure Construct operates in n stages. In stage i, Construct builds M_i and $\preceq_i$ by calling the procedure Generate with inputs M_{i-1} and $\preceq_{i-1}$ (constructed in stage $i-1$) and π_i, the image of i under π. We define $M_0 = \preceq_0 = \emptyset$. We now describe Generate$(M_i, \preceq_i, \pi_i)$. For simplicity, we write M for M_i, $\preceq$ for $\preceq_i$, and j for π_i in the remainder of this section. We will refer to line numbers in Figure 4-4 in angle brackets. For example, $\langle 8 \rangle$ refers to line 8, and $\langle 9-12 \rangle$ refers to lines 9 through 12. We sometimes write line numbers within a sentence, to indicate the line in Figure 4-4 that the sentence refers to.

The main body of Generate proceeds in a loop. The loop ends when process p_j performs its rem_j action, that is, enters its remainder section. Before entering the main loop within $\langle 10-43 \rangle$, Generate first creates a new critical metastep m containing try_j, indicating that p_j starts in its trying section $\langle 8 \rangle$. We add m to M, and set $\check{m}$ to m $\langle 9 \rangle$. $\check{m}$ keeps track of the metastep created or modified during the previous or current iteration of the main loop, depending on where we are in the loop[15]. We call $\langle 8-9 \rangle$ the *zeroth iteration* of Generate.

Next, we begin the main loop between $\langle 10 \rangle$ and $\langle 43 \rangle$. We will call each pass through $\langle 10-43 \rangle$ an *iteration* of Generate[16]. The k'th pass through $\langle 10-43 \rangle$ is the k'th iteration. Each iteration updates M and $\preceq$, by adding or modifying metasteps in M, and adding (but never modifying) relations to $\preceq$. Let $\iota \geq 1$ denote an iteration of Generate, and let ι^- denote the iteration of Generate preceding ι (if $\iota = 1$, then ι^- is the zeroth iteration).

In order for the operations performed in iteration ι to be well defined, we require that certain properties hold about the values of M, $\preceq$ and $\check{m}$ at the end of ι^-. In particular, we make the following assumptions.

Assumption 2 (Correctness of Iteration ι^- of Generate) *Let $M_{\iota^-}, \preceq_{\iota^-}$ and $\check{m}_{\iota^-}$ denote the values of $M, \preceq$ and $\check{m}$ at the end of iteration ι^-.*

1. *Any output of* Plin$(M_{\iota^-}, \preceq_{\iota^-}, \check{m}_{\iota^-})$ *is a run of $\mathcal{A}$.*

2. *For any $\ell \in L$, the set of* write *metasteps in M_{ι^-} accessing ℓ are totally ordered by $\preceq_{\iota^-}$.*

Technically, we should first prove that these properties hold after ι^-, before describing iteration ι. That is, we should present the proof of correctness for earlier iterations of Generate, before describing the current iteration of Generate. However, such a presentation would be both complicated and confusing. Therefore, in the interest of expositional clarity, we defer the proofs of properties 1 and 2 of Assumption 2 to parts 1 and 6 of Lemma 4.6.17, respectively, in Section 4.6.4. Both proofs proceed by induction on the iterations of Generate. That is, to show that Assumption 2 holds for iteration ι^-, parts 1 and 6 of 4.6.17 assume that Generate is well defined for ι^-. This in turn

[15] In the first iteration of the main loop, $\check{m}$ is simply the metastep created in $\langle 8 \rangle$.

[16] We will give a slightly expanded definition of an iteration, taking into account the multiple calls to Generate made by Construct, in Section 4.6.1. For our present discussion, it suffices to consider only the passes through $\langle 10-43 \rangle$ in the current call to Generate by Construct.

requires showing that Assumption 2 holds for iteration $\iota - 2$, for which we need Generate be well defined for iteration $\iota - 2$, etc. Eventually, in the base case, we prove parts 1 and 6 of Lemma 4.6.17 hold for the zeroth iteration (*i.e.*, after $\langle 9 \rangle$), which does not require any assumptions. Thus, while the validity of Generate and the validity of Assumption 2 are mutually dependent, the dependence is inductive, not circular. We will now proceed to describe what happens in the current iteration of Generate, supposing Assumption 2 for the previous iteration.

In $\langle 11 \rangle$, we first set α to be a linearization of all metasteps in $\mu \in M$ such that $\mu \preceq \check{m}$. This is computed by the function $\text{PLIN}(M, \preceq, \check{m})$. We have $\alpha \in runs(\mathcal{A})$, by part 1 of Assumption 2. Using α, we can compute p_j's next step e as $\delta(\alpha, j)$[17]. Let ℓ be the register that e accesses, if e is a read or write step[18].

We split into three cases, depending on e's type. If e is a write step $\langle 13 \rangle$, then we set m_w to be the minimum write metastep in M that accesses ℓ, and that $\not\preceq \check{m}$ $\langle 15 \rangle$. By part 2 of Assumption 2, the set of write metasteps on ℓ is totally ordered, and so either m_w is a metastep, or $m_w = \emptyset$[19]. When $m_w \neq \emptyset$, we *insert* e into m_w, by adding e to $writes(m_w)$ $\langle 17 \rangle$. The idea is that this hides p_i's presence, because e will be overwritten by the winning step in m_w before it is read by any process, when we linearize any set of metasteps including m_w. Next, we add the relation $(\check{m}, m_w)$ to $\preceq$, indicating that $\check{m} \preceq m_w$. Finally, we set $\check{m}$ to be m_w.

In the case where $m_w = \emptyset$ $\langle 19 \rangle$, we create a new write metastep m containing only e, with e as the winning step. Then, we compute the *set* R^* of the *maximal read* metasteps in M accessing ℓ that $\not\preceq \check{m}$. The read metasteps on ℓ are not necessarily totally ordered, so R^* may contain several metasteps. We order m after every metastep in R^* $\langle 26 \rangle$. If we did not do this, the processes performing the read metasteps may be able to see p_j in some linearizations. We record having ordered m after all the metasteps in R^*, by setting $preads(m)$ to R^* $\langle 24 \rangle$. Lastly, in $\langle 27 \rangle$, we order m after $\check{m}$, then set $\check{m}$ to m.

The case when e is a read step is similar. Here, we begin by computing m_{ws}, the minimum write metastep in M accessing ℓ that $\not\preceq \check{m}$, and that would cause p_j to *change its state* if p_j read the value of the metastep $\langle 30 \rangle$. Since we assumed the set of write metasteps on ℓ is totally ordered, then either m_{ws} is a metastep, or $m_{ws} = \emptyset$. We use the helper function $\text{SC}(\alpha, m, j)$, which returns a Boolean value indicating whether process p_j would change its state if it read the value of metastep m when it is in state $st(\alpha, j)$. If $m_{ws} \neq \emptyset$, then we add e to $reads(m_{ws})$. Otherwise, we create a new read metastep m containing only e, and set $reads(m) = \{e\}$.

Lastly, if e is a critical step $\langle 39 \rangle$, then we simply make a new metastep for e and order it after $\check{m}$.

After n stages of the Construct procedure, we output M_n and $\preceq_n$.

[17]Recall that $\delta(\alpha, j)$ computes the next step of p_j, using the final state of p_j in α.

[18]Recall that by definition, $reg(e) = \perp$ for a critical step e.

[19]Recall that by definition, $\min_{\preceq} S$ can either returns the *set* of minimal elements in S, if there is more than one or no minimal element, or it can return the unique minimum *element* in S.

4.6 Correctness Properties of the Construction

In this section, we prove a series of properties about Construct. The main goal of this section is to prove Theorem 4.6.20, which states that in any linearization of an output of Construct(π), all the processes $p_1, \ldots, p_n$ enter the critical section, in the order given by π. We first introduce the notation we will use in our proof, and in the remainder of this chapter, and also give an outline of the structure of the proof.

4.6.1 Notation

In the remainder of this section, fix an arbitrary execution θ of Construct. Many of the proofs in this section use induction on θ. We first define terminology to refer to the portions of θ that we induct over. Notice that the Construct algorithm has a two level iterative structure. That is, $\langle 3-5 \rangle$ of Construct consists of a loop, calling the function Generate n times. Each call to Generate itself loops through $\langle 10-43 \rangle$. We will show in Lemma 4.6.19 that every call to Generate in θ terminates. Assuming this, we define j_i, for any $i \in [n]$, to be the number of times Generate loops through $\langle 10-43 \rangle$, during the i'th call to Generate from Construct in θ.

Let $i \geq 1$, $j \in [j_i]$, and consider the i'th time that Construct calls Generate in θ. Then we call $\langle 8-9 \rangle$ of Generate *iteration* $(i, 0)$, and we call the j'th execution of $\langle 10-43 \rangle$ of Generate *iteration* (i, j). We often use the symbol ι (or ι', ι_1, etc.) to denote an iteration when the actual values of i and j do not matter. Let $\iota = (i, j)$ be an iteration, for some $i \in [n]$. If $j < j_i$, then we say the *next* iteration after ι is $(i, j+1)$. If $j = j_i$, then we say the *next* iteration after ι is $(i+1, 0)$ (unless $i = n$, in which case there is no next iteration after (n, j_n)). For any $i \in [n]$, we define $\iota^i = (i, j_i)$ for the last iteration in the i'th call to Generate by Construct. We denote the set of all iterations in θ by $\mathcal{I} = \bigcup_{i \in [n], 0 \leq j \leq j_i} \{(i, j)\}$. In the remainder of this chapter, when we say that ι is an iteration, we mean that $\iota \in \mathcal{I}$.

Using the definition of "next" iteration above, we can order $\mathcal{I}$ in increasing order as

$$(1,0), (1,1), \ldots, (1, j_1), (2,0), (2,1), \ldots, (n-1, j_{n-1}), (n, 0), \ldots, (n, j_n).$$

When we say that we induct over the execution θ of Construct, we mean that we induct over the iterations in $\mathcal{I}$, ordered as above. Notice that this ordering is lexicographic. That is, given two iterations $\iota_1 = (i_1, j_1)$ and $\iota_2 = (i_2, j_2)$, we have $\iota_1 < \iota_2$ in the above ordering if either $i_1 < i_2$, or $i_1 = i_2$ and $j_1 < j_2$.

Given an iteration ι, if $\iota \neq \iota^n$, we define $\iota \oplus 1$ as the next iteration in the above ordering. If $\iota = \iota^n$, then we define $\iota^n \oplus 1 = \iota^n$. If $\iota \neq (1, 0)$, then we define $\iota \ominus 1$ to be the iteration before ι. If $\iota = (1, 0)$, then we define $\iota \ominus 1 = \iota$. We sometimes write ι^+ for $\iota \oplus 1$, and ι^- for $\iota \ominus 1$. Let ι_1 and ι_2 be two iterations, such that $\iota_1 < \iota_2$. Then we defined $\iota_2 - \iota_1 = \varsigma$ to be the *number of iterations* between ι_1 and ι_2 (in θ). That is, ς is such that $\iota_2 = \iota_1 \underbrace{\oplus 1 \ldots \oplus 1}_{\varsigma \text{ times}}$. Also, if ι is an iteration, and $\varsigma \in \mathbb{N}$, then we define $\iota \ominus \varsigma = \iota \underbrace{\ominus 1 \ldots \ominus 1}_{\varsigma \text{ times}}$, and $\iota \oplus \varsigma = \iota \underbrace{\oplus 1 \ldots \oplus 1}_{\varsigma \text{ times}}$.

We now define notation for the values of the variables of Construct during an iteration ι.

Definition 4.6.1 *Let $\iota = (i, j)$ be any iteration. Then we define the following.*

1. *If $\iota = (i, 0)$, then we let M_ι, $\preceq_\iota$ and $\check{m}_\iota$ be the values of M, $\preceq$ and $\check{m}$, respectively, at the* end *of $\langle 9 \rangle$ in ι. Also, we let $\alpha_\iota = \varepsilon$ (the empty run), and $e_\iota = \mathsf{try}_{\pi_i}$.*

2. *If $\iota \neq (i, 0)$, then we let M_ι, $\preceq_\iota$, $\check{m}_\iota$, e_ι and α_ι be the values of M, $\preceq$, $\check{m}$, e and α, respectively, at the* end of *$\langle 42 \rangle$ in ι.*

3. *We define $N_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (\mu \preceq_{\iota^-} \check{m}_{\iota^-})\}$.*

4. *(a) If $j > 0$, then we define*

$$R_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{R}) \wedge (\mu \not\preceq_{\iota^-} \check{m}_{\iota^-})\}$$

to be value of R in $\langle 23 \rangle$ of ι, and we define R^*_ι to be the value of R^* in $\langle 24 \rangle$ of ι.

   (b) If $j = 0$, then we define $R_\iota = R^*_\iota = \emptyset$.

5. *(a) If $j > 0$, then we define*

$$W_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \not\preceq_{\iota^-} \check{m}_{\iota^-})\}$$

to be value of W in $\langle 14 \rangle$ of iteration ι. We also define

$$W^s_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \not\preceq_{\iota^-} \check{m}_{\iota^-}) \wedge \mathrm{SC}(\alpha_{\iota^-}, \mu, \pi_i)\}$$

to be the value of W^s $\langle 29 \rangle$ of iteration ι.

   (b) If $j = 0$, then we define $W_\iota = W^s_\iota = \emptyset$.

Notice that in Definition 4.6.1, $M_\iota, \preceq_\iota$ and $\check{m}_\iota$ always represent the values of $M, \preceq$ and $\check{m}$ at the end of some iteration, be it an iteration of the form $(i, 0)$ for $i \in [n]$, or (i, j) for $i, j \geq 1$. Also, $\check{m}_\iota$ is the metastep that was either created or modified in $\langle 8 \rangle$, $\langle 15 \rangle$, $\langle 20 \rangle$, $\langle 30 \rangle$, $\langle 35 \rangle$ or $\langle 40 \rangle$ of iteration ι, depending on the behavior of Generate in ι. Lastly, for any $i \in [n]$, we define $M_i = M_{\iota^i}$ and $\preceq_i = \preceq_{\iota^i}$. M_i contains all the metasteps created in iteration ι^i or earlier. Also, it contains all the metasteps that contain process π_i.

Let ι_1 and ι_2 be two different iterations, and let m be a metastep, such that $m \in M_{\iota_1}$ and $m \in M_{\iota_2}$. Then this means that there is a metastep with *label* m in both M_{ι_1} and M_{ι_2}. However, the values of the *attributes* of m may be different in iterations ι_1 and ι_2. For example, the set of processes appearing in m, $procs(m)$, may be different in ι_1 and ι_2. We now define notation to refer to the values of the attributes of m in an iteration ι.

Definition 4.6.2 *Let ι be any iteration. Then we define the following.*

1. *If $\iota = (i, 0)$, for some $i \in [n]$, then we define the* version *of m, written $vers(m, \iota)$, as a record consisting of the values of all the attributes of m, at the end of $\langle 9 \rangle$.*

2. *If $\iota \neq (i, 0)$, for some $i \in [n]$, then we define the* version *of m, written $vers(m, \iota)$, as a record consisting of the values of all the attributes of m, at the end of $\langle 42 \rangle$.*

3. *Given the name of any attribute of m, such as procs, we write $vers(m, \iota).procs$ to refer to the value of $procs(m)$ in ι (either at the end of $\langle 9 \rangle$ or $\langle 42 \rangle$, depending on whether ι equals $(i, 0)$).*

Since we talk about the versions of metasteps extensively in the remainder of the chapter, we will write $vers(m, \iota)$ more concisely as m^ι. Given the name of any attribute of m, such as *procs*, we write $procs(m^\iota)$ to mean $vers(m, \iota).procs$. As another example, if ι_1 and ι_2 are two iterations, then $reads((\check{m}_{\iota_2})^{\iota_1}) = vers(\check{m}_{\iota_2}, \iota_1).reads$ is the set of read steps contained in $\check{m}_{\iota_2}$, after iteration ι_1 (*i.e.*, at the end of $\langle 9 \rangle$ or $\langle 42 \rangle$). Recall that $\check{m}_{\iota_2}$ is the value of the variable $\check{m}$ after iteration ι_2. The value of $\check{m}$ is, in turn, the label of the metastep that was created or modified in iteration ι_2. Thus, $\check{m}_{\iota_2}$ is itself the label of a metastep.

If all the attributes of a metastep m are the same in two iterations ι_1 and ι_2, then we write $m^{\iota_1} = m^{\iota_2}$. Certain attributes of a metastep, such as the value *val* of a write metastep, once set to a non-initial value in some iteration, do not change in any subsequent iteration. In this case, we may omit the version of metastep when referring to this attribute. For example, if m is a write metastep, then we simply write $val(m)$, for the value of m in any iteration. If $m \notin M_\iota$, then we define $m^\iota = \perp$, so that all attributes of m have the value $\perp$. Finally, if N is a set of metasteps, then we write $N^\iota = \{\mu^\iota \mid \mu \in N\}$ for the iteration ι versions of all metasteps in N.

By inspection of the Construct algorithm, we see that each iteration ι belongs to one of several types. If $\iota = (i, 0)$, for some $i \in [n]$, then a critical metastep is created in ι. Thus, we say that ι is a *critical create* iteration. If $\iota \neq (i, 0)$, for any $i \in [n]$, then we define the type of ι as follows. In $\langle 11 \rangle$ of ι, Construct computes e_ι. Then, if the tests on $\langle 13 \rangle$ and $\langle 16 \rangle$ are true (so that $type(e_\iota) = \mathtt{W}$, and $m_w \neq \emptyset$), we say that ι is a *write modify* iteration. If the tests on $\langle 28 \rangle$ and $\langle 31 \rangle$ are true (so that $type(e_\iota) = \mathtt{R}$, and $m_{ws} \neq \emptyset$), then we say ι is a *read modify* iteration. If the tests on $\langle 13 \rangle$ and $\langle 19 \rangle$ are true (so that $type(e_\iota) = \mathtt{W}$, and $m_w = \emptyset$), then we say ι is *write create* iteration. If the tests on $\langle 28 \rangle$ and $\langle 34 \rangle$ are true (so that $type(e_\iota) = \mathtt{R}$, and $m_{ws} = \emptyset$), we say ι is a *read create* iteration. Finally, if the test on $\langle 39 \rangle$ is true (so that $type(e_\iota) = \mathtt{C}$), then we say ι is a *critical create* iteration. If ι is either a read or write modify iteration, we also say ι is a *modify* iteration. Otherwise, we also say ι is a *create* iteration.

Finally, we define notation associated with an execution of the helper function Lin. Let M be a set of metasteps, let $\preceq$ be a partial order on M, and let γ represent an execution of $\text{Lin}(M, \preceq)$. Recall that γ works by first ordering M using any total order on M consistent with $\preceq$. We call this total order the γ *order of* M. Having ordered M, γ next calls $\text{Seq}(m)$, for every $m \in M$. Notice that Seq works with a particular *version* of m. That is, if γ occurs at the end of iteration ι, then $\text{Seq}(m^\iota)$ works by ordering the steps in $steps(m^\iota)$, so that all steps in $writes(m^\iota)$ precede $\diamond(win(m^\iota))$, which precedes all steps in $reads(m^\iota)$. We call this ordering on $steps(m^\iota)$ the γ *order of* m^ι. Let α be the step sequence that is produced by execution γ of Lin. Then we call α the *output* of γ. We also say that α is *an output* of $\text{Lin}(M, \preceq)$, since Lin is nondeterministic, and may return different outputs on the same input. In the remainder of this chapter, we will write $\text{Lin}(M^\iota, \preceq)$ (instead of simply $\text{Lin}(M, \preceq)$) to denote the execution of Lin, working with the iteration ι versions of the metasteps in M. Lastly, given an $m \in M$, we write $\text{Plin}(M^\iota, \preceq, m) = \text{Lin}(N^\iota, \preceq)$, where $N = \{\mu \,|\, (\mu \in M) \wedge (\mu \preceq m)\}$.

4.6.2 Outline of Properties

In this section, we give an outline of the lemmas and theorems appearing in Sections 4.6.3 to 4.6.5. The lemmas are primarily used to prove Theorems 4.6.20 and 4.6.21, though some lemmas, particularly Lemma 4.6.17, are also used in later sections. We will use M and $\preceq$ to denote the values of M_ι and $\preceq_\iota$, in some generic iteration ι. The descriptions in this section are meant to convey intuition and to highlight the general logical relationship between the lemmas. They may not correspond exactly with the formal statements of the lemmas. More precise descriptions of the lemmas will be presented when the lemmas are formally stated.

The main goal of the next three subsections is to prove Theorem 4.6.20, which states that in any linearization of $((M_n)^{\iota^n}, \preceq_n)$, all the processes $p_1, \ldots, p_n$ enter the critical section, and they do so in the order π. To prove this theorem, we first show some basic properties about Construct in Section 4.6.3. For example, we show that $\preceq$ is a partial order on M (Lemma 4.6.6), and that the set of metasteps containing any process is totally ordered by $\preceq$ (Lemma 4.6.8). Section 4.6.4 shows more advanced properties of Construct. Most of the properties in this section are listed in Lemma 4.6.17. Lemma 4.6.17 is proved inductively; that is, it shows the properties hold in some iteration ι, assuming they hold in iteration $\iota \ominus 1$. The reason we list most of the properties in Section 4.6.4 in one lemma, instead of dividing them into multiple lemmas, is that the properties are interdependent. For example, proving Part 9 of Lemma 4.6.17 for iteration ι requires first proving Part 5 of the lemma for ι, which requires proving Part 1 for ι, which in turn requires proving Part 9 of the lemma for iteration $\iota \ominus 1$. We now describe the main parts of Lemma 4.6.17.

Let α be a linearization of $((M_n)^{\iota^n}, \preceq_n)$, and let p_{π_i} and p_{π_j} be two processes, such that $1 \leq i < j \leq n$. Recall that Theorem 4.6.20 asserts that p_{π_i} enters the critical section before p_{π_j} in

α. Intuitively, the reason for this is that p_{π_i} does not see p_{π_j}, and so p_{π_i} will not wait for p_{π_j} before p_{π_i} enters the critical section. Formalizing this idea involves the following two strands of argument. Firstly, we need to show that p_{π_i} and p_{π_j} actually enter the critical section in α. This is done by appealing to the progress property of mutual exclusion, in Definition 4.3.3. However, in order to invoke the progress property, we first need to show that α is a run of $\mathcal{A}$. Indeed, since α is a linearization of $((M_n)^{\iota^n}, \preceq_n)$, we only know *a priori* that α is step sequence. Showing that $\alpha \in runs(\mathcal{A})$ is the content of Part 1 of Lemma 4.6.17.

In addition to showing that p_{π_i} and p_{π_j} enter the critical section, we need to formalize the idea that p_{π_i} does not see p_{π_j}. This is done in Part 9 of Lemma 4.6.17, which essentially shows that we can *pause* processes $p_{\pi_{i+1}}, \ldots, p_{\pi_j}, \ldots, p_{\pi_n}$ at any point in a run, while continuing to run processes $p_{\pi_1}, \ldots, p_{\pi_i}$, and guarantee that $p_{\pi_1}, \ldots, p_{\pi_i}$ all still enter the critical section. Thus, processes $p_{\pi_1}, \ldots, p_{\pi_i}$ are oblivious to the presence of processes $p_{\pi_{i+1}}, \ldots, p_{\pi_n}$, and will take steps whether or not the latter set of processes take steps. Part 9 of Lemma 4.6.17 relies on Part 5 of the lemma, which shows that the states of $p_{\pi_1}, \ldots, p_{\pi_i}$ and the values of the registers accessed by $p_{\pi_1}, \ldots, p_{\pi_i}$ depend only on what steps $p_{\pi_1}, \ldots, p_{\pi_i}$ took, and not on what steps $p_{\pi_{i+1}}, \ldots, p_{\pi_n}$ took. That is, given two runs, in which processes $p_{\pi_1}, \ldots, p_{\pi_i}$ take the same set of steps, but $p_{\pi_{i+1}}, \ldots, p_{\pi_n}$ take different steps, the states of $p_{\pi_1}, \ldots, p_{\pi_i}$ and the values of the registers they access are the same at the end of both runs. Part 5 uses Part 4 of Lemma 4.6.17, which gives a convenient way to compute the state of a process after a run. There are several other parts of Lemma 4.6.17 that we will describe when we formally present the lemma in Section 4.6.4.

4.6.3 Basic Properties of Construct

This section presents some basic properties of the Construct algorithm. Recall that θ is a fixed execution of Construct(π), for some $\pi \in S_n$, and that an iteration always refers to an iteration of θ.

The first lemma shows how M_ι and $\preceq_\iota$ change during an iteration ι. That is, it shows what happens when we move up one iteration in Construct. It says that, except in some boundary cases (when $i = (i, 0)$), we have the following: α_ι is computed by linearizing all the metasteps $m \in M_{\iota^-}$ such that $m \preceq_{\iota^-} \check{m}_{\iota^-}$; e_ι is a step of π_i computed from α_ι; e_ι is a step in $\check{m}_\iota$; $\preceq_\iota$ contains all the relations in $\preceq_{\iota^-}$, plus the relation $(\check{m}_{\iota^-}, \check{m}_\iota)$ (plus possibly some relations of the form $(\mu, \check{m}_\iota)$, for $\mu \in M_{\iota^-}$, if ι is a write create iteration); for any $m \in M_\iota$ other than $\check{m}_\iota$, the ι and ι^- versions of m are the same.

Lemma 4.6.3 (Up Lemma) *Let $\iota = (i, j)$ be any iteration. Then we have the following.*

1. *If $\iota \neq (i, 0)$, then α_ι is an output of* Plin$((M_{\iota^-})^{\iota^-}, \preceq_{\iota^-}, \check{m}_{\iota^-}) \equiv$ Lin$((N_\iota)^{\iota^-}, \preceq_{\iota^-})$[20]. *If $\iota =$*

[20]Recall from Definition 4.6.1 that $N_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (\mu \preceq_{\iota^-} \check{m}_{\iota^-})\}$.

$(i, 0)$, then $\alpha_\iota = \varepsilon$.

2. *$e_\iota = \delta(\alpha_\iota, \pi_i)$, $e_\iota \in steps((\check{m}_\iota)^\iota)$, and $proc(e_\iota) = \pi_i$.*

3. *$M_\iota = M_{\iota^-} \cup \{\check{m}_\iota\}$.*

4. *If $\iota = (i, 0)$, then we have the following.*

   (a) ι is a critical create iteration.

   (b) $\check{m}_\iota \notin M_{\iota^-}$, $e_\iota = \mathsf{try}_{\pi_i}$, and $procs((\check{m}_\iota)^\iota) = \{\pi_i\}$.

   (c) For all $m \in M_{\iota^-}$, $m^\iota = m^{\iota^-}$.

   (d) $\preceq_{\iota^-} = \preceq_\iota$.

5. *If $\iota \neq (i, 0)$ and ι is a create iteration, then we have the following.*

   (a) $\check{m}_\iota \notin M_{\iota^-}$, and $procs((\check{m}_\iota)^\iota) = \{\pi_i\}$.

   (b) For all $m \in M_{\iota^-}$, $m^\iota = m^{\iota^-}$.

   (c) If $type(\check{m}_\iota) \in \{\mathtt{R}, \mathtt{C}\}$, then $\preceq_\iota = \preceq_{\iota^-} \cup \{(\check{m}_{\iota^-}, \check{m}_\iota)\}$.

   (d) If $type(\check{m}_\iota) = \mathtt{W}$, then $\preceq_\iota = \preceq_{\iota^-} \cup \{(\check{m}_{\iota^-}, \check{m}_\iota)\} \cup \bigcup_{\mu \in R^*_\iota} \{(\mu, \check{m}_\iota)\}$.

6. *If $\iota \neq (i, 0)$ and ι is a modify iteration, then we have the following.*

   (a) $M_{\iota^-} = M_\iota$, and $\check{m}_\iota \in M_{\iota^-}$.

   (b) $\check{m}_\iota \not\preceq_{\iota^-} \check{m}_{\iota^-}$.

   (c) For all $m \in M_\iota$ such that $m \neq \check{m}_\iota$, we have $m^\iota = m^{\iota^-}$.

   (d) $procs((\check{m}_\iota)^\iota) = procs((\check{m}_\iota)^{\iota^-}) \cup \{\pi_i\}$.

   (e) $\preceq_\iota = \preceq_{\iota^-} \cup \{(\check{m}_{\iota^-}, \check{m}_\iota)\}$.

Proof. This lemma essentially lists the different cases that can arise in iteration ι. By inspection of Figure 4-4, it is easy to check that all the statements are correct. □

The following lemma states that M and $\preceq$ are "stable". In particular, the lemma says that once a metastep is added to M in some iteration, it is never removed in any later iteration. Also, once two metasteps have been ordered in in some iteration, then their ordering never changes during later iterations.

Lemma 4.6.4 (Stability Lemma A) *Let ι_1 and ι_2 be two iterations, such that $\iota_1 < \iota_2$. Let $m_1, m_2 \in M_{\iota_1}$, and suppose that $m_1 \preceq_{\iota_1} m_2$, and $m_2 \not\preceq_{\iota_1} m_1$. Then we have the following.*

1. *$m_1, m_2 \in M_{\iota_2}$.*

2. *$m_1 \preceq_{\iota_2} m_2$, and $m_2 \not\preceq_{\iota_2} m_1$.*

Proof. We first prove that the lemma holds when ι_1 and ι_2 differ by one iteration.

Claim 4.6.5 *Let ι be any iteration, let $m_1, m_2 \in M_\iota$, and suppose that $m_1 \preceq_\iota m_2$ and $m_2 \not\preceq_\iota m_1$. Then we have the following.*

1. *$m_1, m_2 \in M_{\iota^+}$.*
2. *$m_1 \preceq_{\iota^+} m_2$, and $m_2 \not\preceq_{\iota^+} m_1$.*

Then, to get Lemma 4.6.4, we simply apply Claim 4.6.5 $\iota_2 \ominus \iota_1$ times, starting from iteration ι_1. We now prove Claim 4.6.5.

Proof of Claim 4.6.5. We prove each part of the claim separately.

- *Part 1.*

  By Lemma 4.6.3, we see that $M_\iota \subseteq M_{\iota^+}$, and so $m_1, m_2 \in M_{\iota^+}$.

- *Part 2, and ι^+ is a create iteration.*

  By part 5 of Lemma 4.6.3, we have $\preceq_{\iota^+} = \preceq_\iota \cup \bigcup_{\mu \in N}\{(\mu, \check{m}_{\iota^+})\}$, for some $N \subseteq M_\iota$, and $\check{m}_{\iota^+} \notin M_\iota$. By assumption, we have $m_1 \preceq_\iota m_2$, and $m_2 \not\preceq_\iota m_1$. Then we have $m_1 \preceq_{\iota^+} m_2$, because $\preceq_\iota \subseteq \preceq_{\iota^+}$. Also, we have $m_2 \not\preceq_{\iota^+} m_1$. Indeed, if $m_2 \preceq_{\iota^+} m_1$, then we must have $m_2 \preceq_{\iota^+} \mu$, for some $\mu \in N$, and $\check{m}_{\iota^+} \preceq_{\iota^+} m_1$. But we see that $\check{m}_{\iota^+} \not\preceq_{\iota^+} m$, for any $m \in M_{\iota^+}$. Thus, we have $m_2 \not\preceq_{\iota^+} m_1$.

- *Part 2, and ι^+ is a modify iteration.*

  By part 6 of Lemma 4.6.3, we have $\preceq_{\iota^+} = \preceq_\iota \cup \{(\check{m}_\iota, \check{m}_{\iota^+})\}$, where $\check{m}_{\iota^+} \in M_\iota$, and $\check{m}_{\iota^+} \not\preceq_\iota \check{m}_\iota$. We have $m_1 \preceq_{\iota^+} m_2$, because $\preceq_\iota \subseteq \preceq_{\iota^+}$. Also, we have $m_2 \not\preceq_{\iota^+} m_1$. Indeed, if $m_2 \preceq_{\iota^+} m_1$, then we must have $m_2 \preceq_\iota \check{m}_\iota$ and $\check{m}_{\iota^+} \preceq_\iota m_1$. Then, since $m_1 \preceq_\iota m_2$, we have $\check{m}_{\iota^+} \preceq_\iota m_1 \preceq_\iota m_2 \preceq_\iota \check{m}_\iota$, a contradiction. Thus, we have $m_2 \not\preceq_{\iota^+} m_1$.

□

Lemma 4.6.6 (Partial Order Lemma) *Let ι be any iteration. Then $\preceq_\iota$ is a partial order on M_ι.*

Proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if the lemma is true up to ι, then it is true for $\iota \oplus 1$. CONSTRUCT creates $\preceq_{\iota^+}$ based on $\preceq_\iota$ and the type of iteration ι^+. Thus, we consider the following cases.

If ι^+ is a modify iteration, then for any $m_1, m_2 \in M_{\iota^+}$, we have $m_1, m_2 \in M_\iota$. Since $\preceq_\iota$ is a partial order by the inductive hypothesis, then at most one of $m_1 \preceq_\iota m_2$ and $m_2 \preceq_\iota m_1$ holds. Then, by applying Lemma 4.6.4, we see that at most one of $m_1 \preceq_{\iota^+} m_2$ and $m_2 \preceq_{\iota^+} m_1$ holds as well. Thus, $\preceq_{\iota^+}$ is a partial order on M_{ι^+}.

If ι is a create iteration, then by Lemma 4.6.3, we have $\preceq_{\iota^+} = \preceq_\iota \cup \{(\check{m}_\iota, \check{m}_{\iota^+})\}$, where $\check{m}_{\iota^+} \notin M_\iota$. So, since $\preceq_\iota$ is a partial order on M_ι, then $\preceq_{\iota^+}$ is a partial order on M_{ι^+}. □

We want to show that for any process, the set of metasteps containing that process is totally ordered. We define the following.

Definition 4.6.7 (Function Φ) *Let $\iota = (i, j)$ be any iteration, $k \in [i]$, and $N \subseteq M_\iota$. Define the following.*

1. $\Phi(\iota, k) = \{\mu \,|\, (\mu \in M_\iota) \wedge (\pi_k \in procs(\mu^\iota))\}$*, and* $\phi(\iota, k) = |\Phi(\iota, k)|$.

2. $\Phi(\iota, N, k) = \{\mu \,|\, (\mu \in N) \wedge (\pi_k \in procs(\mu^\iota))\}$*, and* $\phi(\iota, N, k) = |\Phi(\iota, N, k)|$.

Thus, $\Phi(\iota, k)$ and $\phi(\iota, k)$ are the set and number of metasteps containing process π_k after iteration ι. $\Phi(\iota, N, k)$ and $\phi(\iota, N, k)$ are the set and number of metasteps in N containing π_k after ι.

The following lemma essentially states that the set of metasteps containing any process is totally ordered. More precisely, if $\iota = (i, j)$ is an iteration, then there are $j+1$ metasteps containing π_i in M_ι. Also, for any $k \in [i]$, the set of metasteps containing π_k consists of $\check{m}_{(k,h)}$, for $h = 0, \ldots, \phi(\iota, k) - 1$. Furthermore, these metasteps are ordered in increasing order of h. That is, we have $\check{m}_{(k,h-1)} \preceq_\iota \check{m}_{(k,h)}$, for any $h \in [\phi(\iota, k) - 1]$.

Lemma 4.6.8 (Order Lemma A) *Let $\iota = (i, j)$ be any iteration, and let $k \in [i]$. Then we have the following.*

1. $\phi(\iota, i) = j + 1$.

2. $\Phi(\iota, k) = \{\check{m}_{(k,h)} \,|\, 0 \leq h < \phi(\iota, k)\}$.

3. *For any $0 \leq h_1, h_2 < \phi(\iota, k)$ such that $h_1 < h_2$, we have $\check{m}_{(k,h_1)} \prec_\iota \check{m}_{(k,h_2)}$.*

Proof. We use induction on ι. If $\iota = (1, 0)$, the lemma is obvious. We show that if the lemma is true for ι, then it is true for $\iota \oplus 1$. Consider the following cases.

- $\iota^+ = (i + 1, 0)$.

  Consider two cases, either $k = i + 1$, or $k \in [i]$.

  In the first case, Lemma 4.6.3 shows that $\Phi(\iota^+, i + 1) = \{\check{m}_{\iota^+}\}$. Thus, there is only one metastep containing process π_{i+1}, and the lemma follows immediately.

  Next, let $k \in [i]$. Since $k < i + 1$, we only need to prove parts 2 and 3 of the lemma. Lemma 4.6.3 shows that $\Phi(\iota^+, k) = \Phi(\iota, k)$. Given $m_1, m_2 \in \Phi(\iota, k)$, m_1 and m_2 are ordered in $\preceq_\iota$ by the inductive hypothesis. By Lemma 4.6.4, m_1 and m_2 are ordered the same way in $\preceq_{\iota^+}$ as in $\preceq_\iota$. Thus, parts 2 and 3 of the lemma follow.

- $\iota^+ = (i, j + 1)$.

  Consider two cases, either $k \in [i - 1]$, or $k = i$.

First, let $k \in [i-1]$. Then it suffices to prove parts 2 and 3 of the lemma. By Lemma 4.6.3, we have $\Phi(\iota^+, k) = \Phi(\iota, k)$. Also, for any $m_1, m_2 \in \Phi(\iota, k)$, m_1 and m_2 are ordered in $\preceq_\iota$ by induction, and by Lemma 4.6.4, they are ordered the same way in $\preceq_{\iota^+}$. Thus, the lemma holds for all $k \in [i-1]$.

Next, let $k = i$. By Lemma 4.6.3, we have $\Phi(\iota^+, i) = \Phi(\iota, i) \cup \{\check{m}_{\iota^+}\}$. Also, $\pi_i \in \mathit{procs}((\check{m}_{\iota^+})^{\iota^+})$, and $\pi_i \notin \mathit{procs}((\check{m}_{\iota^+})^{\iota})$. So, there is one more metastep containing π_i in M_{ι^+} than in M_ι, and we have $\phi(\iota^+, i) = \phi(\iota, i) + 1 = j + 2$, where the second equation follows from the inductive hypothesis. Thus, part 1 of the lemma holds.

By parts 1 and 2 of the inductive hypothesis, we have $\Phi(\iota, i) = \{\check{m}_{(i,h)} \,|\, 0 \leq h \leq j\}$. Thus, $\Phi(\iota^+, i) = \{\check{m}_{(i,h)} \,|\, 0 \leq h \leq j+1\}$, and part 2 of the lemma holds.

By part 3 of the inductive hypothesis, for any $0 \leq h_1, h_2 \leq j$ such that $h_1 < h_2$, we have $\check{m}_{(i,h_1)} \prec_\iota \check{m}_{(i,h_2)}$. Then by Lemma 4.6.4, we have $\check{m}_{(i,h_1)} \prec_{\iota^+} \check{m}_{(i,h_2)}$. By Lemma 4.6.3, we have $\check{m}_\iota \prec_{\iota^+} \check{m}_{\iota^+}$. Thus, for any $0 \leq h < j+1$, we have $\check{m}_{(i,h)} \prec_{\iota^+} \check{m}_{\iota^+} = \check{m}_{(i,j+1)}$. Thus, part 3 of the lemma holds.

□

Let $\iota = (i,j)$ be any iteration. The next lemma compares a prefix N of $(M_\iota, \preceq_\iota)$, with $\check{N} = N \cap M_{\iota^-}$. First, it states that $\check{N}$ is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$. Next, it states that for any $k \in [i-1]$, N and $\check{N}$ contain the same set of metasteps containing process π_k. Finally, it states that if $\check{m}_\iota \notin N$, then N and $\check{N}$ contain the same set of metasteps containing π_i. Otherwise, if $\check{m}_\iota \in N$, then N contains one more metastep containing π_i than $\check{N}$, namely, $\check{m}_\iota$. Thus, the lemma compares a prefix with the "version" of the prefix moved down one iteration.

Lemma 4.6.9 (Down Lemma A) *Let $\iota = (i,j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\check{N} = N \cap M_{\iota^-}$. Then we have the following.*

1. *$\check{N}$ is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$.*

2. *If $\check{m}_\iota \notin N$, then for all $k \in [i]$, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$.*

3. *If $\check{m}_\iota \in N$, then for all $k \in [i-1]$, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$. Also, we have $\Phi(\iota, N, i) = \Phi(\iota^-, \check{N}, i) \cup \{\check{m}_\iota\}$.*

Proof. We use induction on ι. The lemma holds for $\iota = (1,0)$. We show that if the lemma holds up to iteration $\iota \ominus 1$, then it also holds for ι. Let N be a prefix of $(M_\iota, \preceq_\iota)$, and $\check{N} = N \cap M_{\iota^-}$. We prove each part of the lemma separately.

- *Part 1*

Let $m_1 \in \check{N}$, $m_2 \in M_{\iota^-}$, and suppose that $m_2 \preceq_{\iota^-} m_1$. To show that $\check{N}$ is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$, we need to show $m_2 \in \check{N}$. Since $m_1, m_2 \in M_{\iota^-}$ and $m_2 \preceq_{\iota^-} m_1$, then by Lemma 4.6.4, we have $m_1, m_2 \in M_\iota$, and $m_2 \preceq_\iota m_1$. Since $m_1 \in \check{N}$, then $m_1 \in N$. Since N is a prefix and $m_2 \preceq_\iota m_1$, we have $m_2 \in N$. Thus, $m_2 \in N \cap M_{\iota^-} = \check{N}$, and so $\check{N}$ is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$.

- *Part 2*

  From Lemma 4.6.3, we have that if $m \in M_\iota$ and $m \neq \check{m}_\iota$, then $m \in M_{\iota^-}$. Thus, since $\check{m}_\iota \notin N$, we have $N = \check{N}$. Also from Lemma 4.6.3, we get that if $m \in M_\iota$ and $m \neq \check{m}_\iota$, then $m^{\iota^-} = m^\iota$. Thus, for any $k \in [i]$, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$.

- *Part 3, ι is a create iteration.*

  From parts 4 and 5 of Lemma 4.6.3, we get the following. First, we have $M_\iota = M_{\iota^-} \cup \{\check{m}_\iota\}$, and $\check{m}_\iota \notin M_{\iota^-}$. Second, we have $procs((\check{m}_\iota)^\iota) = \{\pi_i\}$. Lastly, if $m \in M_\iota$ and $m \neq \check{m}_\iota$, then $m^{\iota^-} = m^\iota$. Thus, for all $k \in [i-1]$, we have $\Phi(\iota, N, k) = \Phi(\iota^-, N, k)$, and we also have $\Phi(\iota, N, i) = \Phi(\iota^-, \check{N}, i) \cup \{\check{m}_\iota\}$.

- *Part 3, ι is a modify iteration.*

  From part 6 of Lemma 4.6.3, we have $M_\iota = M_{\iota^-}$. Also, $procs((\check{m}_\iota)^\iota) = procs((\check{m}_\iota)^{\iota^-}) \cup \{\pi_i\}$, and $m^\iota = m^{\iota^-}$ for all $m \neq \check{m}_\iota$. Thus again, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$, and $\Phi(\iota, N, i) = \Phi(\iota^-, \check{N}, i) \cup \{\check{m}_\iota\}$.

□

Let ι, N and $\check{N}$ be defined as in Lemma 4.6.9. Recall that e_ι is the value of e at the end of $\langle 42 \rangle$ in iteration ι. Thus, e_ι is computed in $\langle 11 \rangle$ of iteration ι. Let α be a linearization of $(N^\iota, \preceq_\iota)$[21] , and let $\check{\alpha}$ be the same as α, but with step e_ι removed[22]. The next lemma states that $\check{\alpha}$ is a linearization of $(\check{N}^{\iota^-}, \preceq_{\iota^-})$.

Lemma 4.6.10 (Down Lemma B) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let α be an output of* Lin$(N^\iota, \preceq_\iota)$. *Let $\check{N} = N \cap M_{\iota^-}$, and let $\check{\alpha}$ be α with step e_ι removed. Then $\check{\alpha}$ is an output of of* Lin$(\check{N}^{\iota^-}, \preceq_{\iota^-})$

Proof. Let γ be the execution of Lin$(N^\iota, \preceq_\iota)$ that produced α. Let $<_N$ be the γ order of N, and for each $m \in N$, let $<_m$ be the γ order of m^ι. Since $\check{N} \subseteq N$, then $<_N$ is a total order on $\check{N}$. We claim $<_N$ is consistent with $\preceq_{\iota^-}$. Indeed, suppose $m_1, m_2 \in N$, and $m_1 <_N m_2$. Then, since $<_N$ is consistent with $\preceq_\iota$, we have $m_2 \not\preceq_\iota m_1$. Then by the contrapositive of Lemma 4.6.4, we have $m_2 \not\preceq_{\iota^-} m_1$, and

[21]Recall from the end of Section 4.6.1 that Lin$(N^\iota, \preceq_\iota)$ is formed by first ordering N with a total order consistent with $\preceq_\iota$, and then totally ordering $steps(m^\iota)$, the steps contained in m at the end of iteration ι, for all $m \in N$.

[22]If e_ι does not occur in α, then $\alpha = \check{\alpha}$.

so the claim holds. Now, define an execution $\check{\gamma}$ of $\textsc{Lin}(\check{N}^{\iota^-}, \preceq_{\iota^-})$ where we order $\check{N}$ using $<_N$, and for each $m \in \check{N}$, order m^{ι^-} using $<_m$. $\check{\gamma}$ is a valid execution of $\textsc{Lin}(\check{N}^{\iota^-}, \preceq_{\iota^-})$, because $<_N$ is a total order on $\check{N}$ consistent with $\preceq_{\iota^-}$, and because for all $m \in \check{N}$, we have $\mathit{steps}(m^{\iota^-}) \subseteq \mathit{steps}(m^{\iota})$, so that $<_m$ is a total order on $\mathit{steps}(m^{\iota^-})$. We claim that the output of $\check{\gamma}$ is $\check{\alpha}$. Consider two cases, either $\check{m}_\iota \notin N$, or $\check{m}_\iota \in N$.

Suppose first that $\check{m}_\iota \notin N$. Then, since e_ι is contained in $\mathit{steps}((\check{m}_\iota)^\iota)$, e_ι does not occur in α. Thus, $\alpha = \check{\alpha}$. By Lemma 4.6.3, we have $N = \check{N}$, and for all $m \in \check{N}$, we have $m^\iota = m^{\iota^-}$. Thus, the output of $\check{\gamma}$ is $\check{\alpha} = \alpha$.

Next, suppose that $\check{m}_\iota \in N$. Then α and $\check{\alpha}$ differ only in e_ι. Consider the following cases.

- *ι is a create iteration.*

  By Lemma 4.6.3, we have $N = \check{N} \cup \{\check{m}_\iota\}$, and $\mathit{steps}((\check{m}_\iota)^\iota) = \{e_\iota\}$. Also, if $m \in N$ and $m \neq \check{m}_\iota$, then $m^\iota = m^{\iota^-}$. Thus, the output of $\check{\gamma}$ equals α with step e_ι removed, which is $\check{\alpha}$.

- *ι is a modify iteration.*

  By Lemma 4.6.3, we have $N = \check{N}$, $\mathit{steps}((\check{m}_\iota)^\iota) = \mathit{steps}((\check{m}_\iota)^{\iota^-}) \cup \{e_\iota\}$, and for $m \in N$ and $m \neq \check{m}_\iota$, we have $m^\iota = m^{\iota^-}$. Thus, again the output of $\check{\gamma}$ equals α with step e_ι removed, which is $\check{\alpha}$.

$\square$

The next lemma essentially states that π_i does not affect the views of process p_{π_k}, for $k < i$. Recall that for a step sequence α, $\mathit{acc}(\alpha)$ is the set of registers accessed by the steps in α.

Lemma 4.6.11 (Down Lemma C) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and suppose $\check{m}_\iota \in N$. Let α be an output of $\textsc{Lin}(N^\iota, \preceq_\iota)$, and suppose $\alpha \in \mathit{runs}(\mathcal{A})$. Let $(\check{m}_\iota)^\iota$ be linearized as β in α*[23]*, and write $\alpha = \alpha^- \circ \beta \circ \alpha^+$. Let $\check{\beta}$ be β with step e_ι removed, let $\alpha_1 = \alpha^- \circ \beta$, and $\alpha_2 = \alpha^- \circ \check{\beta}$*[24]*. Then we have the following.*

1. *For any $k \in [i-1]$, $\mathit{st}(\alpha_1, \pi_k) = \mathit{st}(\alpha_2, \pi_k)$.*
2. *For any $\ell \in \mathit{acc}(\alpha^+)$, we have $\mathit{st}(\alpha_1, \ell) = \mathit{st}(\alpha_2, \ell)$.*

Proof. Consider two cases, either $\mathit{type}(e_\iota) = \mathtt{R}$, or $\mathit{type}(e_\iota) = \mathtt{W}$.

- $\mathit{type}(e_\iota) = \mathtt{R}$.

  Since e_ι is a read step, it does not change the state of any registers. Thus, since β contains at most one step by any process, both parts of the lemma follow immediately.

[23]Recall that this means that in the execution of $\textsc{Lin}(N^\iota, \preceq_\iota)$ that produced α, the output of $\textsc{Seq}((\check{m}_\iota)^\iota)$ is β.

[24]Notice that since we assume $\alpha \in \mathit{runs}(\mathcal{A})$, and since $\alpha^- \circ \beta = \alpha_1$ is a prefix of α, then we have $\alpha_1 \in \mathit{runs}(\mathcal{A})$. Also, since β is the linearization of m, it contains at most one step by any process. Thus, since $\check{\beta}$ and β differ in at most one step, and $\alpha^- \circ \beta \in \mathit{runs}(\mathcal{A})$, then we have $\alpha^- \circ \check{\beta} = \alpha_2 \in \mathit{runs}(\mathcal{A})$.

- $type(e_\iota) = \mathtt{W}$.

  Consider two cases, either $\diamond(winner(\check{m}_\iota)) \neq \pi_i$, or $\diamond(winner(\check{m}_\iota)) = \pi_i$.

  If $\diamond(winner(\check{m}_\iota)) \neq \pi_i$, let $e^* = \diamond(win(\check{m}_\iota))$ be the winning step in $\check{m}_\iota$. By the definition of $\textsc{Seq}((\check{m}_\iota)^\iota)$, the value written by e_ι is overwritten by the value written by e^* before it is read by any process π_k, $k \in [i-1]$. Thus, both parts of the lemma follow.

  If $\diamond(winner(\check{m}_\iota)) = \pi_i$, then let $\ell = reg(\check{m}_\iota)$. By Lemma 4.6.3, ι must be a write create iteration. Then, we have $procs((\check{m}_\iota)^\iota) = \{\pi_i\}$, and $\beta = e_\iota$. So, we have $\alpha_1 = \alpha^- \circ e_\iota$ and $\alpha_2 = \alpha^-$, and part 1 of the lemma follows. To show part 2 of the lemma, we prove the following.

  Claim 4.6.12 *Let e be any step in α^+. Then e does not access ℓ.*

  Proof. Suppose for contradiction that there is a step e in α^+ that accesses ℓ. Then either e is a write or a read step on ℓ.

  Suppose first that e writes to ℓ. Then e is contained in some write metastep $m \in M_{\iota^-}$. In addition, since e occurs in α^+, then $m \not\preceq_\iota \check{m}_\iota$. Indeed, if $m \preceq_\iota \check{m}_\iota$, then since $(\check{m}_\iota)^\iota$ is linearized as β in α, the linearization of m, and step e, must occur in α^-. Since $m \not\preceq_\iota \check{m}_\iota$, then we also have $m \not\preceq_{\iota^-} \check{m}_{\iota^-}$. But then, at ⟨15⟩ in iteration ι, we would have $m_w \neq \emptyset$, because m is a write metastep on register ℓ, and $m \not\preceq_{\iota^-} \check{m}_{\iota^-}$. Thus, the test at ⟨19⟩ in ι must have failed, and so ι could not have been a write create iteration, a contradiction. Thus, there are no write steps to ℓ in α^+.

  Next, suppose that e reads ℓ. Then e cannot be contained in a write metastep, by the same argument as above. Suppose e is contained in a read metastep m. Then we have $m \in R_\iota$[25]. In ⟨26⟩ in ι, we set $m \prec_\iota \check{m}_\iota$. But then, e cannot occur in α^+, since α^+ only contains (linearizations of) metasteps that $\not\preceq_\iota \check{m}_\iota$. Again, this is a contradiction. Together with the previous paragraph, this shows that any e in α^+ does not access ℓ. □

  Claim 4.6.12 is equivalent to saying that for all $\ell' \in acc(\alpha^+)$, $\ell' \neq \ell$. Thus, part 2 of the lemma follows.

□

Recall that M_k is the output of $\textsc{Generate}$ after iteration ι^k. The next lemma is similar to Lemma 4.6.9, but lets us move N "down" multiple iterations.

Lemma 4.6.13 (Down Lemma D) *Let $\iota = (i,j)$ be any iteration, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Let $k \in [i-1]$, and let $\check{N} = N \cap M_k$. Then we have the following*

[25]Recall from Definition 4.6.1 that $R_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{R}) \wedge (\mu \not\preceq_{\iota^-} \check{m}_{\iota^-})\}$.

1. *$\check{N}$ is a prefix of $(M_k, \preceq_k)$.*

2. *For all $h \in [k]$, we have $\Phi(\iota, N, h) = \Phi(\iota^k, \check{N}, h)$.*

Proof. Let $\varsigma = \iota - \iota^k$ be the number of iterations between ι and ι^k. Let $N_0 = N$, and for $r \in [\varsigma]$, inductively define $N_r = N_{r-1} \cap M_{\iota \ominus r}$. We prove the each part of the lemma separately.

- *Part 1.*

  We first prove the following.

  Claim 4.6.14 *For all $r \in [\varsigma]$, N_r is a prefix of $(M_{\iota\ominus r}, \preceq_{\iota\ominus r})$.*

  Proof. This follows from induction on r. Indeed, by Lemma 4.6.9, it holds for $r = 1$. Also, if it holds for r, then by Lemma 4.6.9, it holds for $r + 1$. □

  By Lemma 4.6.4, we have $M_{\iota\ominus r} \subseteq M_{\iota\ominus(r-1)}$, for all $r \in [\varsigma]$. Thus, since $N_r = N_{r-1} \cap M_{\iota\ominus r}$, we have $N_r = N \cap M_{\iota\ominus r}$. Thus, using Claim 4.6.14, where we let $r = \varsigma$, we get that $N_\varsigma = \check{N}$ is a prefix of $(M_{\iota\ominus\varsigma}, \preceq_{\iota\ominus\varsigma}) = (M_k, \preceq_k)$.

- *Part 2.*

  Let $r \in [\varsigma]$. Then since $h \in [k]$ and $k < i$, by Lemma 4.6.9, we have that $\Phi(\iota \ominus r, N_r, h) = \Phi(\iota \ominus (r-1), N_{r-1}, h)$. From this, we get

$$\Phi(\iota, N, h) = \Phi(\iota, N_0, h) = \Phi(\iota \ominus 1, N_1, h) = \ldots = \Phi(\iota \ominus \varsigma, N_\varsigma, h) = \Phi(\iota^k, \check{N}, h).$$

□

4.6.4 Main Properties of Construct

In this section, we formally state and prove the main properties that Construct satisfies. We first define the following.

For any iteration ι and any register ℓ, define $\Psi(\iota, \ell)$ to be the set of metasteps in M_ι that access ℓ, and define $\Psi^w(\iota, \ell)$ to be the set of write metasteps in M_ι that access ℓ. If $m \in M_\iota$, define $\Upsilon(\iota, \ell, m)$ to be the set of metasteps in M_ι that access ℓ, and that also $\preceq_\iota m$. Also, define $\Upsilon(\iota, m)$ to be the set of all metasteps μ such that $\mu \preceq_\iota m$. Formally, we have the following.

Definition 4.6.15 (Function Ψ) *Let ι be any iteration, let $N \subseteq M_\iota$, and let $\ell \in L$. Define the following.*

1. $\Psi(\iota, \ell) = \{\mu \,|\, (\mu \in M_\iota) \wedge (reg(\mu) = \ell)\}$.

2. $\Psi^w(\iota, \ell) = \{\mu \,|\, (\mu \in M_\iota) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W})\}$.

Definition 4.6.16 (Function Υ) *Let ι be any iteration, let $\ell \in L$, and let $m \in M_\iota$. Define the following.*

1. $\Upsilon(\iota, \ell, m) = \{\mu \,|\, (\mu \in \Psi(\iota, \ell) \wedge (\mu \preceq_\iota m)\}$.

2. $\Upsilon(\iota, m) = \{\mu \,|\, (\mu \in M_\iota) \wedge (\mu \preceq_\iota m)\}$.

Given a set of metasteps N, we write $acc(N) = \{reg(\mu) \,|\, \mu \in N\}$ for the set of all registers accessed by the metasteps in N. We now state the main properties that Construct satisfies in iteration ι.

Lemma 4.6.17 (Properties of Iteration ι of Construct)
Let $\iota = (i, j)$ be any iteration, let $k \in [i]$, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Let α be an output of Lin$(N^\iota, \preceq_\iota)$. *Then we have the following.*

1. (Run A) $\alpha \in runs(\mathcal{A})$.

2. (Run B) *Suppose $k \in [i-1]$. Let $h \in [k]$, and let α_k be an output of* Lin$((M_k)^{\iota^k}, \preceq_k)$. *Then the steps try_{π_h}, enter_{π_h}, exit_{π_h} and rem_{π_h} occur in α_k.*

3. (Read Step) *Suppose that $type(e_\iota) = \mathtt{R}$ and $W_\iota \neq \emptyset$*[26] . *Then we have $type(\check{m}_\iota) = \mathtt{W}$, and ι is a write modify iteration.*

4. (Down E) *Let $\check{\alpha}$ be α with step e_ι removed. Then we have the following.*

   (a) $\check{\alpha} \in runs(\mathcal{A})$.

   (b) *If $k \in [i-1]$, then $st(\alpha, \pi_k) = st(\check{\alpha}, \pi_k)$.*

   (c) *If $\check{m}_\iota \notin N$, then $st(\alpha, \pi_i) = st(\check{\alpha}, \pi_i)$.*

   (d) *If $\check{m}_\iota \in N$, $type(\check{m}_\iota) = \mathtt{W}$, and $type(e_\iota) = \mathtt{W}$, then we have $st(\alpha, \pi_i) = \Delta(\check{\alpha}, e_\iota, \pi_i)$.*

   (e) *If $\check{m}_\iota \in N$, $type(\check{m}_\iota) = \mathtt{W}$, and $type(e_\iota) = \mathtt{R}$, then let $\ell = reg(\check{m}_\iota)$, and let $v = val(\check{m}_\iota)$. Choose any $s \in S$ such that $st(s, \pi_i) = st(\check{\alpha}, \pi_i)$ and $st(s, \ell) = v$. Then we have $st(\alpha, \pi_i) = \Delta(s, e_\iota, \pi_i)$.*

   (f) *If $\check{m}_\iota \in N$ and $type(\check{m}_\iota) = \mathtt{R}$, then we have $st(\alpha, \pi_i) = \Delta(\check{\alpha}, e_\iota, \pi_i)$.*

5. (Consistency A) *Let $\iota_1 = (i_1, j_1) \leq \iota$ be an iteration, let N_1 be a prefix of $(M_{\iota_1}, \preceq_{\iota_1})$, and let α_1 be an output of* Lin$((N_1)^{\iota_1}, \preceq_{\iota_1})$. *Suppose $k \in [i_1]$. Then if $\Phi(\iota, N, k) = \Phi(\iota_1, N_1, k)$, we have $st(\alpha, \pi_k) = st(\alpha_1, \pi_k)$.*

6. (Order B) *Let $\ell \in L$, $m_1 \in \Psi(\iota, \ell)$, and let $m_2 \in \Psi^w(\iota, \ell)$. Then either $m_1 \preceq_\iota m_2$ or $m_2 \preceq_\iota m_1$.*

[26] Recall from Definition 4.6.1 that $W_\iota = \{\{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \npreceq_{\iota^-} \check{m}_{\iota^-})\}$.

7. (Order C) *Let $\iota_1 \le \iota$ be any iteration. Let $\ell \in L$, and let $m \in \Psi^w(\iota_1, \ell)$. Then $\Upsilon(\iota_1, \ell, m) = \Upsilon(\iota, \ell, m)$.*

8. (Consistency B) *Suppose $k \in [i-1]$. Let $N_1 = N \cap M_k$, and let α_1 be an output of* $\textsc{Lin}((N_1)^{\iota^k}, \preceq_k)$. *Then we have the following.*

   (a) For all $h \in [k]$, we have $st(\alpha, \pi_h) = st(\alpha_1, \pi_h)$.

   (b) For all $\ell \in acc(M_k \backslash N_1)$, we have $st(\alpha, \ell) = st(\alpha_1, \ell)$.

9. (Extension) *Suppose $k \in [i-1]$. Then there exist step sequences $\check{\alpha}$ and β, and an output α_k of* $\textsc{Lin}((M_k)^{\iota^k}, \preceq_k)$, *such that $\alpha_k = \check{\alpha} \circ \beta$ and $\alpha \circ \beta \in runs(\mathcal{A})$. Furthermore, if $m \in M_k \backslash N$, then the linearization of m^{ι^k} occurs in β.*

We first describe Lemma 4.6.17. Let $\iota = (i, j)$ be any iteration, and let $k \in [i]$. Let N be a prefix of $(M_\iota, \preceq_\iota)$, and let α be an output of $\textsc{Lin}(N^\iota, \preceq_\iota)$. Part 1 of the lemma says that α, in addition to being a step sequence, is actually a run of $\mathcal{A}$.

Part 2 of the lemma says that if $k \in [i-1]$ and $h \in [k]$, then any linearization $((M_k)^{\iota^k}, \preceq_k)$ contains the critical steps of process π_h, namely try_{π_h}, enter_{π_h}, exit_{π_h} and rem_{π_h}.

Part 3 says that if the step computed for π_i in iteration ι, e_ι, is a read step on some register ℓ, and if there exist any write metasteps on ℓ that $\not\preceq_{\iota^-} \check{m}_{\iota^-}$, then e_ι must be added to some such metastep. Thus, in particular, ι is a write modify iteration. Note that this does not immediately follow from the assumptions of part 3, because $W_\iota \neq \emptyset$ does not immediately imply that $W_\iota^s \neq \emptyset$, or $m_{ws} \neq \emptyset$ in $\langle 31 \rangle$ of ι.

Part 4 says that if $\check{\alpha}$ is equal to α with step e_ι removed, then $\check{\alpha}$ is a run of $\mathcal{A}$. The states of all processes other than π_i are the same after α and $\check{\alpha}$. Also, if $\check{m}_\iota \notin N$, then the state of π_i is the same after α and $\check{\alpha}$, and if $\check{m}_\iota \in N$, then the state of π_i after α can be computed from its state after $\check{\alpha}$, e_ι, and (possibly) the value of $\check{m}_\iota$. Note that e_ι does not necessarily occur at the end of α. Nevertheless, part 4 essentially allows us to move e_ι to the end of α, when we want to compute the state of π_i after α.

Part 5 essentially says that the state of a process after a linearization of a prefix from any iteration depends only on the set of metasteps in the prefix that contain the process. More precisely, if $\iota_1 \le \iota$ is any iteration, N_1 is any prefix of $(M_{\iota_1}, \preceq_{\iota_1})$, and α_1 is any linearization of $((N_1)^{\iota_1}, \preceq_{\iota_1})$, then as long as π_k is contained in the same set of metasteps in N and N_1, the state of π_k is the same after α and α_1.

Part 6 says that for any register ℓ, a write metastep on ℓ is ordered by $\preceq_\iota$ with respect to any other (read or write) metastep on ℓ.

Part 7 say that for a write metastep on register ℓ, the set of metasteps on ℓ that precede m in any two iterations is the same, as long as $m \in M$ during the smaller of the two iterations..

Part 8 says that if $k \in [i-1]$, $h \in [k]$ and $N_1 = N \cap M_k$, then the state of process π_h is the same after α as after a linearization α_1 of $((N_1)^{\iota^k}, \preceq_k)$. Also, the value of any register accessed by a metastep in $M_k \backslash N$ is the same after α and α_1.

Part 9 of the lemma says that if we start with the run α, in which processes $p_{\pi_1}, \ldots, p_{\pi_i}$ take steps, then for any $k \in [i-1]$, we can extend α to a run $\alpha \circ \beta$, such that only processes $p_{\pi_1}, \ldots, p_{\pi_k}$ take steps in β. Furthermore, $p_{\pi_1}, \ldots, p_{\pi_k}$ all perform their rem steps in $\alpha \circ \beta$.

Proof. We use induction on ι. All parts of the lemma are easy to verify for $\iota = (1,0)$. Indeed, when $\iota = (1,0)$, then M_ι contains one metastep, containing the critical step try_{π_1}, and $\preceq_\iota = \emptyset$. Thus, we have $\alpha = \varepsilon$ or $\alpha = \mathsf{try}_{\pi_1}$. Then, parts 1, 4 and 5 of the lemma clearly hold, while the other parts are vacuously satisfied. Next, suppose for induction that the lemma holds up to iteration $\iota \ominus 1$; then we show that it also holds for ι. We will call each part of the lemma a *sublemma*. Let γ be the execution of $\textsc{Lin}(N^\iota, \preceq_\iota)$ that produced α.

1. *Part 1,* $\textsc{Run A}$.

   Let $\check{N} = N \cap M_{\iota^-}$, and let $\check{\alpha}$ be α with step e_ι removed. Then by Lemma 4.6.9, $\check{N}$ is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$, and by Lemma 4.6.10, $\check{\alpha}$ is an output of $\textsc{Lin}(N^{\iota^-}, \preceq_{\iota^-})$. Then by the inductive hypothesis, we have $\check{\alpha} \in runs(\mathcal{A})$. If $\check{m}_\iota \notin N$, then since $e_\iota \in steps((\check{m}_\iota)^\iota)$, we have $\alpha = \check{\alpha}$, and so $\alpha \in runs(\mathcal{A})$. Thus, assume that $\check{m}_\iota \in N$.

   If $\iota = (i,0)$, then $e_\iota = \mathsf{try}_{\pi_i}$. e_ι does not affect the state of any other process or register. Conversely, the states of the other processes and registers do not affect the fact that e_ι is the first step by process π_i. Thus, since $\check{\alpha} \in runs(\mathcal{A})$ by induction, we also have $\alpha \in runs(\mathcal{A})$. Next, assume that $\iota \neq (i,0)$. Then, by Lemma 4.6.3, we have $\check{m}_{\iota^-} \preceq_\iota \check{m}_\iota$. Thus, since $\check{m}_\iota \in N$ and N is a prefix, we have $\check{m}_{\iota^-} \in N$.

   Now, to show that $\alpha \in runs(\mathcal{A})$, the main idea is the following. Let α^- and α^+ denote the parts of α before and after e_ι, respectively. Thus, we have $\alpha = \alpha^- \circ e_\iota \circ \alpha^+$. We first want to show that π_i indeed performs the step e_ι after α^-. That is, we want to show that $\delta(\alpha^-, \pi_i) = e_\iota$. To do this, let $N_1 \subseteq M_\iota$ denote the set of all metasteps that are linearized before $\check{m}_\iota$ in α. From Lemma 4.6.8, we can see that N_1 and N_ι[27] contain the same set of metasteps that contain process π_i. Then, using Part 5 of the inductive hypothesis, it follows that π_i is in the same state following α^- and α_ι. Thus, since e_ι is by definition the step that π_i performs after α_ι, e_ι is also the step that π_i performs after α^-, and so we have $\alpha^- \circ e_\iota \in runs(\mathcal{A})$. Now, to complete the proof that $\alpha \in runs(\mathcal{A})$, we use Lemma 4.6.11, which shows that inserting e_ι after α^- does not change the states of processes $\pi_1, \ldots, \pi_{i-1}$, nor the values of any registers accessed in α^+. Thus, since $\alpha^- \circ \alpha^+ = \check{\alpha} \in runs(\mathcal{A})$ by the inductive hypothesis, we also have $\alpha^- \circ e_\iota \circ \alpha^+ \in runs(\mathcal{A})$, by Theorem 4.3.1.

[27] Recall from Section 4.6.1 that $N_\iota = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (\mu \preceq_{\iota^-} \check{m}_{\iota^-})\}$.

We now present the formal proof of the lemma. Recall that $\check{m}_\iota \in N$, and $\iota \neq (i,0)$. Let $\check{m}_{\iota^-}$ and $\check{m}_\iota$ be linearized as β_1 and β_2 in α, respectively, and let $\check{\beta}_2$ be β_2 with step e_ι removed. Write $\alpha = \alpha_1^- \circ \beta_1 \circ \alpha_2^- \circ \beta_2 \circ \alpha^+$, and $\check{\alpha} = \alpha_1^- \circ \beta_1 \circ \alpha_2^- \circ \check{\beta}_2 \circ \alpha^+$. There are no steps by π_i in $\check{\beta}_2$, by definition. Also, there are no steps by π_i in α_2^-, since $\check{m}_\iota$ and $\check{m}_{\iota^-}$ are the last two metasteps (with respect to $\preceq_\iota$) containing π_i.

Let $<_\gamma$ be the γ order of N, and let

$$N_1 = \{\mu \,|\, (\mu \in M_\iota) \wedge (\mu \leq_\gamma \check{m}_{\iota^-})\}.$$

N_1 is a prefix of $(M_\iota, \preceq_\iota)$. Indeed, if $m_1 \in N_1$ and $m_2 \preceq_\iota m_1$, then we have $m_2 \leq_\gamma m_1$, since $<_\gamma$ is consistent with $\preceq_\iota$. So, we have $m_2 \in N_1$.

By Lemma 4.6.3, we have that α_ι is an output of $\textsc{Lin}((N_\iota)^{\iota^-}, \preceq_{\iota^-})$, and $e_\iota = \delta(\alpha_\iota, \pi_i)$. Since $\iota = (i,j)$, then using part 3 of Lemma 4.6.8, we have

$$\Phi(\iota^-, N_1, i) = \{\check{m}_{(i,h)} \,|\, 0 \leq h \leq j-1\} = \Phi(\iota^-, N_\iota, i).$$

Let γ_1 be an execution of $\textsc{Lin}((N_1)^{\iota^-}, \preceq_{\iota^-})$ that orders N_1 using $<_\gamma$, and orders every $m \in N_1$ using the γ order of m. Since $\alpha = \alpha_1^- \circ \beta_1 \circ \alpha_2^- \circ \beta_2 \circ \alpha^+$ is the output of γ, and $\check{m}_{\iota^-}$ is linearized as β_1 in α, and $m^\iota = m^{\iota^-}$ for all $m \in M_\iota \backslash \{\check{m}_\iota\}$, then $\alpha_1^- \circ \beta_1$ is the output of γ_1. Thus, since $\Phi(\iota^-, N_1, i) = \Phi(\iota^-, N_\iota, i)$, we have by part 5 of the inductive hypothesis that

$$st(\alpha_1^- \circ \beta_1, \pi_i) = st(\alpha_\iota, \pi_i).$$

Let $\alpha' = \alpha_1^- \circ \beta_1 \circ \alpha_2^- \circ \beta_2$, and $\check{\alpha}' = \alpha_1^- \circ \beta_1 \circ \alpha_2^- \circ \check{\beta}_2$. Since $\check{\alpha} = \check{\alpha}' \circ \alpha^+ \in runs(\mathcal{A})$, we have $\check{\alpha}' \in runs(\mathcal{A})$. Also, we have

$$st(\alpha_\iota, \pi_i) = st(\alpha_1^- \circ \beta_1, \pi_i) = st(\check{\alpha}', \pi_i).$$

Here, the second equality follows because there are no steps by π_i in α_2^- or in $\check{\beta}_2$. From this, we get that

$$\delta(\check{\alpha}', \pi_i) = \delta(\alpha_\iota, \pi_i) = e_\iota.$$

Thus, since $\check{\alpha}' \in runs(\mathcal{A})$, and $\check{\alpha}'$ equals α' with step e_ι removed, we get that

$$\alpha' \in runs(\mathcal{A}). \tag{4.6}$$

By Lemma 4.6.11, we have $\forall k \in [i-1] : st(\alpha', \pi_k) = st(\check{\alpha}', \pi_k)$, and $\forall \ell \in acc(\alpha^+) : st(\alpha', \ell) = st(\check{\alpha}', \ell)$. Also, there are no steps by process π_i in α^+. Thus, using the fact that $\check{\alpha} = \check{\alpha}' \circ \alpha^+ \in$

$runs(\mathcal{A})$ and Theorem 4.3.1, and using Equation 4.6, we have $\alpha' \circ \alpha^+ = \alpha \in runs(\mathcal{A})$.

2. *Part 2,* Run B.

   Since $\iota = (i, j)$ is an iteration of θ and $k < i$, then by inspection of Construct, the h'th call to Generate by Construct terminated. So, M_k contains a critical metastep containing rem_{π_h}, and so rem_{π_h} occurs in α_k. By Part 1 of the inductive hypothesis, $\alpha_k \in runs(\mathcal{A})$, and so α_k satisfies the well formedness property of Definition 4.3.3. Thus, α_k also contains the steps try_{π_h}, enter_{π_h} and exit_{π_h}.

3. *Part 3,* Read Step.

   The main idea is the following. Suppose for contradiction that $type(\check{m}_\iota) = \mathtt{R}$, so that ι is a read create iteration. Then this means that for every $m \in W_\iota$, process π_i does not change its state after reading, in step e_ι, the value written by m. Let the maximum metastep in W_ι, with respect to $\preceq_{\iota^-}$, be m^*, and let $v^* = val(m^*)$. By part 6 of the inductive hypothesis, W_ι is totally ordered by $\preceq_{\iota^-}$, and so m^* is well defined. Using Part 9 of the inductive hypothesis, we can construct a run α' in which e_ι occurs after all metasteps in M_{i-1} have occurred. In particular, e_ι occurs after all the writes in W_ι. The value of ℓ in any extension of α', in which only p_{π_i} take steps, is v^*. But since π_i does not change its state after reading value v^*, and since $p_{\pi_i}, \ldots, p_{\pi_{i-1}}$ are all in their remainder sections in any extension of α', then π_i will stay in the same state forever, contradicting the progress property in Definition 4.3.3.

   We now present the formal proof. By Part 6 of the inductive hypothesis, W_ι is totally ordered by $\preceq_{\iota^-}$. Let $m^* = \max_{\preceq_{\iota^-}} W_\iota$, $v^* = val(m^*)$, and let $\pi_k = \diamond(winner(m^*))$. Then $k < i$. Indeed, we have $m^* \not\preceq_{\iota^-} \check{m}_{\iota^-}$ by the definition of W_ι. But for any metastep m containing π_i, that is, for any $m \in \Phi(\iota^-, i)$, we have $m \preceq_{\iota^-} \check{m}_{\iota^-}$, by Lemma 4.6.8. Hence, $k < i$.

   By Lemma 4.6.3, we have that α_ι is an output of $\textsc{Lin}((N_\iota)^{\iota^-}, \preceq_{\iota^-})$. By Part 9 of the inductive hypothesis, there exists an execution of $\textsc{Lin}((M_{i-1})^{\iota^{i-1}}, \preceq_{i-1})$ with output $\alpha_{i-1} = \check{\alpha} \circ \beta$, such that $\alpha' = \alpha_\iota \circ \beta \in runs(\mathcal{A})$. We have $m^* \in M_{i-1}$, since $\pi_k = \diamond(winner(m^*))$ and $k < i$, so that M_{i-1} contains all metasteps that contain p_{π_k}. Also, we have $m^* \notin N_\iota$, since $m^* \not\preceq_{\iota^-} \check{m}_{\iota^-}$. Thus, we have $m^* \in M_{i-1} \backslash N_\iota$, and the second conclusion of Part 9 of the inductive hypothesis states that the linearization of m^* occurs in β. Then, since m^* is the maximum write metastep to ℓ in M_{i-1}, with respect to $\preceq_{\iota^-}$, we have $m \preceq_{\iota^-} m^*$, for every $m \in M_{\iota^-}$ that is a write metastep on ℓ. Thus, we have $st(\alpha', \ell) = v^*$.

   Let $s_i = st(\alpha_\iota, \pi_i)$ be π_i's state at the end of α_ι. By Lemma 4.6.3, we have $e_\iota = \delta(s_i, \pi_i)$. For any $v \in V$, let

   $$S_v = \{s \mid (s \in S) \wedge (st(s, \pi_i) = s_i) \wedge (st(s, \ell) = v)\}.$$

   That is, S_v is the set of system states in which π_i is in state s_i, and ℓ has value v. Now, suppose for contradiction that $type(\check{m}_\iota) = \mathtt{R}$. Then the test on $\langle 31 \rangle$ of iteration ι must have failed. Thus, by inspection of $\langle 29 \rangle$ and $\langle 31 \rangle$, we have

$$(\forall \mu \in W_\iota)(\forall s \in S_{val(\mu)}) : \Delta(s, e_\iota, \pi_i) = st(\alpha_\iota, \pi_i) = s_i.$$

That is, none of the write metasteps in W_ι write a value that causes π_i to change its state after α_ι. In particular, we have

$$\forall s \in S_{v^*} : \Delta(s, e_\iota, \pi_i) = s_i. \tag{4.7}$$

Notice that β does not contain any steps by π_i, since β comes from a linearization of $((M_{i-1})^{\iota_{i-1}}, \preceq_{i-1})$. Then, since $\delta(\alpha_\iota, \pi_i) = e_\iota$, and $\alpha' = \alpha_\iota \circ \beta \in runs(\mathcal{A})$, we have $\alpha' \circ e_\iota \in runs(\mathcal{A})$. Since $st(\alpha', \ell) = v^*$ and e_ι is a read step, we have $st(\alpha' \circ e_\iota, \ell) = v^*$. Then by Equation 4.7, we have $st(\alpha' \circ e_\iota, \pi_i) = st(\alpha_\iota \circ e_\iota, \pi_i) = s_i$. Thus, we have $st(\alpha' \circ e_\iota) \in S_{v^*}$.

For any $r \in \mathbb{N}$, let $(e_\iota)^r = \underbrace{e_\iota \circ \ldots \circ e_\iota}_{r \text{ times}}$. Since $\delta(s_i, \pi_i) = e_\iota$ and $st(\alpha' \circ e_\iota, \pi_i) = s_i$, we have $\delta(\alpha' \circ e_\iota, \pi_i) = e_\iota$. Then, we have $\alpha' \circ (e_\iota)^2 \in runs(\mathcal{A})$. We also have $st(\alpha' \circ (e_\iota)^2, \ell) = v^*$, and $st(\alpha' \circ (e_\iota)^2, \pi_i) = s_i$, by Equation 4.7. Thus, $\delta(\alpha' \circ (e_\iota)^2, \pi_i) = e_\iota$, and so $\alpha' \circ (e_\iota)^3 \in runs(\mathcal{A})$. Following this pattern, we see that for any $r \in \mathbb{N}$, we have $\alpha' \circ (e_\iota)^r \in runs(\mathcal{A})$.

By part 2 of the inductive hypothesis, we have that for all $h \in [i-1]$, rem_{π_h} appears in α'. Also, since π_h performs try_{π_h} only once, π_h is in its remainder section at the end of $\alpha' \circ (e_\iota)^r$, for every $r \in \mathbb{N}$. Thus, by the progress property in Definition 4.3.3, there exists a sufficiently large $r^* \in \mathbb{N}$ such that rem_{π_i} occurs in $\alpha' \circ (e_\iota)^{r^*}$. But since e_ι is a read step by π_i, this is a contradiction. Thus, we conclude that $type(\check{m}_\iota) = \mathtt{W}$, and ι is a write modify iteration.

4. *Part 4,* Down E.

   We first describe the main idea of the proof. Parts a through c follow easily from earlier lemmas or from induction. Part d of the sublemma follows because e_ι is a write step, and so π_i always transitions to the same state after e_ι, as long as e_ι is placed somewhere after e_{ι^-} in $\check{\alpha}$. Similarly, part e follows because π_i always transitions to the same state after e_ι, as long as e_ι is placed after $e_{\iota'}$ in $\check{\alpha}$, and e_ι reads value v in ℓ. Lastly, to see part f, note that since $\check{m}_\iota$ is a read metastep, then by part 3 of the lemma, there are no write steps to ℓ after e_{ι^-} in $\check{\alpha}$. Thus, e_ι reads the same value in ℓ, no matter where we place e_ι after e_{ι^-} in $\check{\alpha}$, and so, part f follows.

   We now present the formal proof of the sublemma. Part a of the sublemma follows from Lemma 4.6.10, and part 1 of the inductive hypothesis. For the other parts, consider two cases, either $\check{m}_\iota \notin N$, or $\check{m}_\iota \in N$.

If $\check{m}_\iota \notin N$, then since e_ι is contained in $steps((\check{m}_\iota)^\iota)$, we have $\alpha = \check{\alpha}$. Thus, for any $k \in [i]$, we have $st(\alpha, \pi_k) = st(\check{\alpha}, \pi_k)$, and so Part 4 of the lemma holds.

If $\check{m}_\iota \in N$, then consider two cases, either $\iota = (i,0)$, or $\iota \neq (i,0)$. If $\iota = (i,0)$, then $e_\iota = \mathsf{try}_{\pi_i}$. Since e_ι does not change the state of any registers, part b of the sublemma holds. Also, parts c through f of the lemma do not apply. Thus, the sublemma holds.

Next, suppose $\iota \neq (i,0)$. Since $\iota \neq (i,0)$, then e_{ι^-} contains a step by π_i. Suppose $\check{m}_\iota$ is linearized as β in α, and let $\check{\beta}$ be β with step e_ι removed. Write $\alpha = \alpha^- \circ \beta \circ \alpha^+$, and $\check{\alpha} = \alpha^- \circ \check{\beta} \circ \alpha^+$. Also, write $\alpha^- = \alpha_1^- \circ e_{\iota^-} \circ \alpha_2^-$. Since e_{ι^-} is the step taken by π_i before e_ι, there are no steps by π_i in α_2^-, $\check{\beta}$ or α^+. We prove each part of the sublemma separately. Note that part c has already been proven earlier.

- *Part b.*

  By Lemma 4.6.11, for any $k \in [i-1]$, we have $st(\alpha^- \circ \beta, \pi_k) = st(\alpha^- \circ \check{\beta}, \pi_k)$, and $\forall \ell \in acc(\alpha^+) : st(\alpha^- \circ \beta, \ell) = st(\alpha^- \circ \check{\beta}, \ell)$. Then, since α^+ does not contain any steps by π_i, we have

  $$st(\alpha, \pi_k) = st(\alpha^- \circ \beta \circ \alpha^+, \pi_k) = st(\alpha^- \circ \check{\beta} \circ \alpha^+, \pi_k) = st(\check{\alpha}, \pi_k).$$

- *Part d.*

  We have

  $$\begin{aligned} st(\alpha, \pi_i) &= st(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ \beta \circ \alpha^+, \pi_i) \\ &= st(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ \check{\beta} \circ \alpha^+ \circ e_\iota, \pi_i) \\ &= st(\check{\alpha} \circ e_\iota, \pi_i). \end{aligned}$$

  The second equality follows because there are no steps by π_i in α^+, and because e_ι is a write step. The third equality follows by the definition of $\check{\alpha}$.

- *Part e.*

  Since there are no steps by π_i in α_2^-, $\check{\beta}$ or α^+, we have

  $$\begin{aligned} st(\check{\alpha}, \pi_i) &= st(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ \check{\beta} \circ \alpha^+, \pi_i) \\ &= st(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ \check{\beta}, \pi_i) \\ &= st(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^-, \pi_i) \\ &= st(\alpha_1^- \circ e_{\iota^-}, \pi_i). \end{aligned}$$

Thus, we get

$$\delta(\check{\alpha}, \pi_i) = \delta(\alpha_1^- \circ e_{\iota^-}, \pi_i) = e_\iota.$$

Here the second equality follows because e_ι is the next step by π_i in α after e_{ι^-}. Since $\mathit{type}(\check{m}_\iota) = \mathtt{W}$, then e_ι reads $v = \mathit{val}(\check{m}_\iota)$ in α. Thus, if $s \in S$ is any system state such that $\mathit{st}(s, \pi_i) = \mathit{st}(\check{\alpha}, \pi_i)$ and $\mathit{st}(s, \ell) = v$, then we have $\mathit{st}(\alpha, \pi_i) = \Delta(s, e_\iota, \pi_i)$.

- *Part f.*

  Since $\mathit{type}(\check{m}_\iota) = \mathtt{R}$, then by part 3 of the lemma, we have $W_\iota = \emptyset$. Thus, there are no write steps to ℓ in α_2^- or in α^+, since e_{ι^-} is contained in $\mathit{steps}((\check{m}_{\iota^-})^\iota)$, and e_{ι^-} comes before α_2^- and α^+ in α. Also, since $\mathit{type}(\check{m}_\iota) = \mathtt{R}$, we have $\mathit{steps}((\check{m}_\iota)^\iota) = \{e_\iota\}$, and so $\beta = e_\iota$, and $\check{\beta} = \varepsilon$. Thus, we have

$$\begin{aligned} \mathit{st}(\alpha, \pi_i) &= \mathit{st}(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ e_\iota \circ \alpha^+, \pi_i) \\ &= \mathit{st}(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ \alpha^+ \circ e_\iota, \pi_i) \\ &= \mathit{st}(\alpha_1^- \circ e_{\iota^-} \circ \alpha_2^- \circ \check{\beta} \circ \alpha^+ \circ e_\iota, \pi_i) \\ &= \mathit{st}(\check{\alpha} \circ e_\iota, \pi_i). \end{aligned}$$

  The second equality follows because e_ι is a read on ℓ, and there are no writes to ℓ in α^+. The third equality follows because $\check{\beta} = \varepsilon$.

5. *Part 5,* Consistency A.

   We first describe the main idea of the proof. Consider two cases, either $\iota = \iota_1$, or $\iota > \iota_1$. In the first case, let $\check{N} = N \cap M_{\iota^-}$ and $\check{N}_1 = N_1 \cap M_{\iota^-}$, and let $\check{\alpha}$ and $\check{\alpha}_1$ be the (version ι^-) linearizations of $\check{N}$ and $\check{N}_1$. Since $\Phi(\iota, N, k) = \Phi(\iota, N_1, k)$, then we also have $\Phi(\iota^-, \check{N}, k) = \Phi(\iota^-, \check{N}_1, k)$, and so $\mathit{st}(\check{\alpha}, \pi_k) = \mathit{st}(\check{\alpha}_1, \pi_k)$ by induction. Then, to conclude that $\mathit{st}(\alpha, \pi_k) = \mathit{st}(\alpha_1, \pi_k)$, we apply part 4 of the lemma. In the case that $\iota > \iota_1$, we first show that $\Phi(\iota, N, k) = \Phi(\iota_1, N_1, k)$ implies that $\Phi(\iota_1, N, k) = \Phi(\iota_1, N_1, k)$, and then apply part 5 of the inductive hypothesis for iteration ι_1.

   We now present the formal proof. Consider two cases, either $\iota = \iota_1$, or $\iota > \iota_1$.

   - *Case $\iota = \iota_1$.*

     Let $\check{N} = N \cap M_{\iota^-}$ and $\check{N}_1 = N_1 \cap M_{\iota^-}$. Also, let $\check{\alpha}$ be α with step e_ι removed, and let $\check{\alpha}_1$ be α_1 with step e_ι removed. By Lemma 4.6.9, both $\check{N}$ and $\check{N}_1$ are prefixes of $(M_{\iota^-}, \preceq_{\iota^-})$. By Lemma 4.6.10, $\check{\alpha}$ and $\check{\alpha}_1$ are outputs of $\textsc{Lin}((\check{N})^{\iota^-}, \preceq_{\iota^-})$ and $\textsc{Lin}((\check{N}_1)^{\iota^-}, \preceq_{\iota^-})$, respectively.

     We first show that if $k \in [i-1]$ and $\Phi(\iota, N, k) = \Phi(\iota, N_1, k)$, then we have $\mathit{st}(\alpha, \pi_k) = \mathit{st}(\alpha_1, \pi_k)$. By Lemma 4.6.9, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$, and $\Phi(\iota, N_1, k) = \Phi(\iota^-, \check{N}_1, k)$.

Thus, since $\Phi(\iota, N, k) = \Phi(\iota, N_1, k)$, we have $\Phi(\iota^-, \check{N}, k) = \Phi(\iota^-, \check{N}_1, k)$. Then by part 5 of the inductive hypothesis, we have

$$st(\check{\alpha}, \pi_k) = st(\check{\alpha}_1, \pi_k).$$

By part 4.b of the lemma, we have

$$st(\alpha, \pi_k) = st(\check{\alpha}, \pi_k), \qquad st(\alpha_1, \pi_k) = st(\check{\alpha}_1, \pi_k).$$

Thus, we conclude that $st(\alpha, \pi_k) = st(\alpha_1, \pi_k)$, for all $k \in [i-1]$.

Next, suppose $k = i$, and $\Phi(\iota, N, i) = \Phi(\iota, N_1, i)$. Consider two cases, either $\check{m}_\iota \notin N$, or $\check{m}_\iota \in N$.

- $\check{m}_\iota \notin N$.

  Since $\check{m}_\iota \notin N$ and $\Phi(\iota, N, i) = \Phi(\iota, N_1, i)$, we have $\check{m}_\iota \notin N_1$. Then, by Lemma 4.6.9, we have $\Phi(\iota, N, i) = \Phi(\iota^-, \check{N}, i)$ and $\Phi(\iota, N_1, i) = \Phi(\iota^-, \check{N}_1, i)$, and so $\Phi(\iota^-, \check{N}, i) = \Phi(\iota^-, \check{N}_1, i)$. Then, by part 5 of the inductive hypothesis, we have $st(\check{\alpha}, \pi_i) = st(\check{\alpha}_1, \pi_i)$. Since $\check{m}_\iota \notin N$, then by part 4.c of the lemma, we have $st(\alpha, \pi_i) = st(\check{\alpha}, \pi_i)$, and $st(\alpha_1, \pi_i) = st(\check{\alpha}_1, \pi_i)$. Thus, we have $st(\alpha, \pi_i) = st(\alpha_1, \pi_i)$.

- $\check{m}_\iota \in N$.

  Since $\check{m}_\iota \in N$ and $\Phi(\iota, N, i) = \Phi(\iota, N_1, i)$, we have $\check{m}_\iota \in N_1$. Then, by Lemma 4.6.9, we have $\Phi(\iota, N, i) = \Phi(\iota^-, \check{N}, i) \cup \{\check{m}_\iota\}$ and $\Phi(\iota, N_1, i) = \Phi(\iota^-, \check{N}_1, i) \cup \{\check{m}_\iota\}$, and so $\Phi(\iota^-, \check{N}, i) = \Phi(\iota^-, \check{N}_1, i)$. Then, by part 5 of the inductive hypothesis, we have $st(\check{\alpha}, \pi_i) = st(\check{\alpha}_1, \pi_i)$. To complete the proof, consider the following cases.

  Suppose first that $type(\check{m}_\iota) = \mathtt{W}$ and $type(e_\iota) = \mathtt{R}$. Let $\ell = reg(\check{m}_\iota)$, $v = val(\check{m}_\iota)$. Let $s \in S$ be any system state such that $st(s, \pi_i) = st(\check{\alpha}, \pi_i) = st(\check{\alpha}_1, \pi_i)$, and $st(s, \ell) = v$. Then by part 4.e of the lemma, we have

  $$st(\alpha, \pi_i) = \Delta(s, e_\iota, \pi_i) \qquad st(\alpha_1, \pi_i) = \Delta(s, e_\iota, \pi_i).$$

  Thus, we have $st(\alpha, \pi_i) = st(\alpha_1, \pi_i)$.

  Next, suppose that either $type(\check{m}_\iota) = \mathtt{W}$ and $type(e_\iota) = \mathtt{W}$, or $type(\check{m}_\iota) = \mathtt{R}$. Then by parts 4.$d$ and 4.f of the lemma, we have $st(\alpha, \pi_i) = \Delta(\check{\alpha}, e_\iota, \pi_i)$, and $st(\alpha_1, \pi_i) = \Delta(\check{\alpha}_1, e_\iota, \pi_i)$. Thus, again we have $st(\alpha, \pi_i) = st(\alpha_1, \pi_i)$.

- *Case* $\iota > \iota_1$

  Let $\varsigma = \iota - \iota_1$ be the number of iterations between ι and ι_1. Define $N^0 = N$ and $\alpha^0 = \alpha$. For $r \in [1, \varsigma]$, inductively let $N^r = N^{r-1} \cap M_{\iota \ominus r}$, and let α^r be α^{r-1} with step $e_{\iota \ominus (r-1)}$ removed. The following lemma states properties about the "versions" of N and α in iteration $\iota \ominus r$, for any $r \in [\varsigma]$.

Claim 4.6.18 *For any $r \in [\varsigma]$, we have the following.*

(a) N^r is a prefix of $(M_{\iota\ominus r}, \preceq_{\iota\ominus r})$.

(b) $\Phi(\iota \ominus r, N^r, k) = \Phi(\iota \ominus (r-1), N^{r-1}, k)$.

(c) α^r is an output of Lin$((N^r)^{\iota\ominus r}, \preceq_{\iota\ominus r})$.

(d) $st(\alpha^r, \pi_k) = st(\alpha^{r-1}, \pi_k)$.

Proof. We use induction, and prove the claim for $r = 1$. The proof for other values of r uses the inductive hypothesis for $r-1$, and is otherwise the same.

– *Part a.*

  Since $N^0 = N$ is a prefix of $(M_\iota, \preceq_\iota)$, then by Lemma 4.6.9, N^1 is a prefix of $(M_{\iota\ominus 1}, \preceq_{\iota\ominus 1})$.

– *Part b.*

  We claim that if $k = i$, then $\check{m}_\iota \notin N^0$. Indeed, if $k = i$ and $\check{m}_\iota \in N^0$, then since we have $\pi_k \in procs((\check{m}_\iota)^\iota)$, $\pi_k \notin procs((\check{m}_\iota)^{\iota\ominus 1})$, and $\iota_1 \leq \iota \ominus 1$, we get that $\Phi(\iota, N^0, k) \neq \Phi(\iota_1, N_1, k)$, a contradiction. Thus, we either have $k \in [i-1]$, or $k = i$ and $\check{m}_\iota \notin N^0$. In both cases, by Lemma 4.6.9, we have $\Phi(\iota, N^0, k) = \Phi(\iota \ominus 1, N^1, k)$.

– *Part c.*

  We have $N^1 = N^0 \cap M_{\iota\ominus 1}$, and α^1 equals α^0 with step e_ι removed. Thus, since α is an output of Lin$(N^\iota, \preceq_\iota)$, then by Lemma 4.6.10, α^1 is an output of Lin$((N^1)^{\iota\ominus 1}, \preceq_{\iota\ominus 1})$.

– *Part d.*

  As in the proof for part 2, we have that if $k = i$, then $\check{m}_\iota \notin N^0$. Thus, by parts 4.*b* and 4.*c* of the lemma, we have that $st(\alpha^1, \pi_k) = st(\alpha^0, \pi_k)$.

□

We now complete the proof of part 5 of the lemma. From part 1 of Claim 4.6.18, we have that N^ς is a prefix of $(M_{\iota\ominus\varsigma}, \preceq_{\iota\ominus\varsigma}) = (M_{\iota_1}, \preceq_{\iota_1})$. By inductively applying Claim 4.6.18, starting from $r = 1$ up to $r = \varsigma$, we get from part 2 of Claim 4.6.18 that

$$\Phi(\iota_1, N^\varsigma, k) = \Phi(\iota, N, k).$$

By inductively applying part 3 of Claim 4.6.18, we get that there exists an execution of Lin$((N^\varsigma)^{\iota_1}, \preceq_\iota)$ with output α^ς.

Since $\Phi(\iota, N, k) = \Phi(\iota_1, N_1, k)$ by assumption, then by part 2 of Claim 4.6.18, we have $\Phi(\iota_1, N^\varsigma, k) = \Phi(\iota_1, N_1, k)$. Thus, by part 5 of the inductive hypothesis, we have

$$st(\alpha^\varsigma, \pi_k) = st(\alpha_1, \pi_k).$$

Finally, by inductively applying part 4 of Claim 4.6.18, we get that

$$st(\alpha, \pi_k) = st(\alpha^{\varsigma}, \pi_k).$$

Thus, we have $st(\alpha, \pi_k) = st(\alpha_1, \pi_k)$. □

6. *Part 6,* Order B.

The main idea is the following. If ι is a modify iteration, then $M_\iota = M_{\iota^-}$, and also, all metasteps are ordered the same way in $\preceq_\iota$ and $\preceq_{\iota^-}$. Thus, the sublemma follows by induction. If ι is a create iteration, then we can show that $W_\iota = \emptyset$, either using part 3 of the lemma (if ι is a read create iteration), or by direct inspection of Construct (if ι is a write create iteration). Thus, for any write metastep $m_2 \in M_{\iota^-}$ on ℓ, we have $m_2 \preceq_{\iota^-} \check{m}_{\iota^-} \prec_\iota \check{m}_\iota$. From this, the lemma follows.

We now present the formal proof. Choose an $\ell \in L$, $m_1 \in \Psi(\iota, \ell)$ and $m_2 \in \Psi^w(\iota, \ell)$, and consider the following cases.

- *ι is a critical create iteration.*

  By Lemma 4.6.3, we either have $\preceq_\iota = \preceq_{\iota^-}$, or $\preceq_\iota = \preceq_{\iota^-} \cup \{(\check{m}_{\iota^-}, \check{m}_\iota)\}$. Also, $\check{m}_\iota \notin M_{\iota^-}$, and $\check{m}_\iota$ contains a critical step that does not access any registers. Thus, since m_1 and m_2 are ordered in $\preceq_{\iota^-}$ by part 6 of the inductive hypothesis, they are ordered in the same way in $\preceq_\iota$, by Lemma 4.6.4.

- *ι is a read create iteration.*

  By Lemma 4.6.3, we have $\preceq_\iota = \preceq_{\iota^-} \cup \{(\check{m}_{\iota^-}, \check{m}_\iota)\}$. If $reg(e_\iota) \neq \ell$, then the sublemma clearly holds in ι.

  If $reg(e_\iota) = reg(\check{m}_\iota) = \ell$, then since ι is a read create iteration, by part 3 of the lemma, we have $W_\iota = \emptyset$. Since m_1 and m_2 are ordered in $\preceq_{\iota^-}$ by induction, they are ordered the same way in $\preceq_\iota$. Also, since $W_\iota = \emptyset$ and $m_2 \in \Psi^w(\iota, \ell)$, then we have $m_2 \preceq_{\iota^-} \check{m}_{\iota^-}$. Finally, we have $\check{m}_{\iota^-} \prec_\iota \check{m}_\iota$, by ⟨37⟩ of ι. Thus, the sublemma holds for ι.

- *ι is a write create iteration.*

  If $reg(e_\iota) \neq \ell$, then the sublemma holds in ι. If $reg(e_\iota) = \ell$, then since ι is a write create iteration, then the test on ⟨19⟩ in iteration ι succeeded, and so $W_\iota = \emptyset$. Thus, $m_2 \preceq_{\iota^-} \check{m}_{\iota^-}$, and so by ⟨27⟩ of iteration ι, we have $m_2 \prec_\iota \check{m}_\iota$. Also, if $m_1 \in R_\iota$, then it follows from ⟨26⟩ of ι that $m_1 \prec_\iota \check{m}_\iota$. Lastly, m_1 and m_2 are ordered the same way in ι^- and ι. Thus, the sublemma holds for ι.

- *ι is a modify iteration.*

  By Lemma 4.6.3, we have $M_{\iota^-} = M_\iota$. Thus, for any $m_1, m_2 \in M_\iota$, we have $m_1, m_2 \in M_{\iota^-}$, and so by Lemma 4.6.4, m_1 and m_2 are ordered the same way in ι as in ι^-.

7. *Part 7,* Order C.

   We prove the sublemma in the case when ι and ι_1 differ by one iteration. The proof for a general ι_1 is simply an inductive version of the following argument. Let $\ell \in L$ and $m \in \Psi^w(\iota, \ell)$. Then we show that $\Upsilon(\iota, \ell, m) = \Upsilon(\iota^-, \ell, m)$. Let $m_1 \in \Psi(\iota, \ell)$. Then by part 6 of the inductive hypothesis, either $m \preceq_{\iota^-} m_1$ or $m_1 \preceq_{\iota^-} m$. So by Lemma 4.6.4, either $m \preceq_\iota m_1$ or $m_1 \preceq_\iota m$, and so we have $\Upsilon(\iota^-, \ell, m) \subseteq \Upsilon(\iota, \ell, m)$. So, to show $\Upsilon(\iota, \ell, m) = \Upsilon(\iota^-, \ell, m)$, it suffices to show the following:

   $$\text{If } reg(\check{m}_\iota) = \ell \text{ and } \check{m}_\iota \not\preceq_{\iota^-} m, \text{ then } \check{m}_\iota \not\preceq_\iota m. \qquad (*)$$

   To show $(*)$, suppose first that ι is a modify iteration. Then $\check{m}_\iota \in M_\iota$. It suffices to consider the case when $\check{m}_\iota$ accesses ℓ. Then, since $\check{m}_\iota \not\preceq_{\iota^-} m$ by assumption, we have by part 6 of the inductive hypothesis that $m \preceq_{\iota^-} \check{m}_\iota$. Thus, by Lemma 4.6.4, we have $m \preceq_\iota \check{m}_\iota$, and $(*)$ holds.

   Next, suppose ι is a read or write create iteration. Then we claim that $W_\iota = \emptyset$. Indeed, if ι is a write create iteration, then $W_\iota = \emptyset$, or else the test on $\langle 19 \rangle$ of ι would fail, and ι would not be a write create iteration. If ι is a read create iteration, then part 3 of the lemma implies that $W_\iota = \emptyset$. Now, since m is a write metastep on ℓ, then $m \notin W_\iota$, and so $m \preceq_{\iota^-} \check{m}_{\iota^-} \prec_\iota \check{m}_\iota$. So, the assumption of $(*)$ does not hold. Thus, again we have $\Upsilon(\iota, \ell, m) = \Upsilon(\iota^-, \ell, m)$.

8. *Part 8,* Consistency B.

   The main idea is the following. To show part a of the sublemma, we use the fact that $h \leq k < i$ and Lemma 4.6.13 to show that $\Phi(\iota, N, h) = \Phi(\iota^k, N_1, h)$, and then apply part 8 of the lemma to conclude that $st(\alpha, \pi_h) = st(\alpha_1, \pi_h)$. For part b, suppose that $\check{m}_\iota \in N$; otherwise, part b follows easily by induction. If e_ι is a read step, then part b follows easily. If e_ι is a write step, and π_i is not the winner of $\check{m}_\iota$, then the value that π_i writes is overwritten by the value written by the winner of $\check{m}_\iota$, and part b again follows. If e_ι is a write step and π_i is also the winner of $\check{m}_\iota$, then ι is a write create iteration. Let $\ell_1 = reg(\check{m}_\iota)$. We claim that ℓ_1 is not accessed by any metastep in $M_k \backslash N_1$. Indeed, if there is a write metastep $m \in M_k \backslash N_1$ on ℓ_1, then $m \in W_\iota \neq \emptyset$, and so ι is a write modify iteration, a contradiction. Otherwise, if there is a read metastep $m \in M_k \backslash N_1$ on ℓ, then $m \in R_\iota \neq \emptyset$, and we have $m \prec_\iota \check{m}_\iota$. Then, since $\check{m}_\iota \in N$, we have $m \in N$, and $m \notin M_k \backslash N_1$, which is again a contradiction. Thus, $\ell_1 \notin acc(M_k \backslash N_1)$, and part b of the sublemma follows.

   We now present the formal proof. Since $k \in [i]$ and $h \in [k]$, then by Lemma 4.6.13, $\check{N}$ is a prefix of $(M_k, \preceq_k)$, and $\Phi(\iota, N, h) = \Phi(\iota^k, N_1, h)$. Thus, by part 8 of the lemma, we have $st(\alpha, \pi_h) = st(\alpha_1, \pi_h)$, and part 1 of the sublemma holds.

   For part 2 of the sublemma, we consider two cases, either $\check{m}_\iota \notin N$, or $\check{m}_\iota \in N$.

If $\check{m}_\iota \notin N$, then $N = N \cap M_{\iota^-}$, and so by Lemmas 4.6.9 and 4.6.10, N is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$, and α is the output of an execution of $\text{LIN}((N)^{\iota^-}, \preceq_{\iota^-})$. Then by part 8 of the inductive hypothesis, we have $st(\alpha, \ell) = st(\alpha_1, \ell)$, for all $\ell \in acc(M_k \backslash N_1)$.

If $\check{m}_\iota \in N$, then suppose $\check{m}_\iota$ is linearized as β in α, and let $\check{\beta}$ be β with step e_ι removed. Write $\alpha = \alpha^- \circ \beta \circ \alpha^+$, and let $\check{\alpha} = \alpha^- \circ \check{\beta} \circ \alpha^+$, and $\check{N} = N \cap M_{\iota^-}$. By Lemmas 4.6.9 and 4.6.10, $\check{N}$ is a prefix of $(M_{\iota^-}, \preceq_{\iota^-})$, and $\check{\alpha}$ is the output of some execution of $\text{LIN}((\check{N})^{\iota^-}, \preceq_{\iota^-})$. Then by the inductive hypothesis, we have

$$\forall \ell \in acc(M_k \backslash N_1) : st(\check{\alpha}, \ell) = st(\alpha_1, \ell). \tag{4.8}$$

Let $\ell \in acc(M_k \backslash N_1)$. To show that $st(\alpha, \ell) = st(\alpha_1, \ell)$, consider the following cases.

- *$type(e_\iota) =$ R.*

  e_ι does not change the state of any register, and so $st(\alpha^- \circ \beta, \ell) = st(\alpha^- \circ \check{\beta}, \ell)$. Also, e_ι is the last step by process π_i in α, and so there are no steps by π_i in α^+. Thus, we have

  $$\begin{aligned} st(\alpha, \ell) &= st(\alpha^- \circ \beta \circ \alpha^+, \ell) \\ &= st(\alpha^- \circ \check{\beta} \circ \alpha^+, \ell) \\ &= st(\check{\alpha}, \ell) \\ &= st(\alpha_1, \ell). \end{aligned}$$

  Here, the third equation follows by the definition of $\check{\alpha}$, and the last equation follows by Equation 4.8.

- *$type(e_\iota) =$ W, and $\diamond(winner(\check{m}_\iota)) \neq \pi_i$.*

  The value written by step e_ι is overwritten by the value written by step $\diamond(win(\check{m}_\iota))$ before it is read by any process. Thus, $st(\alpha^- \circ \beta, \ell) = st(\alpha^- \circ \check{\beta}, \ell)$. Since there are no steps by π_i in α^+, we have $st(\alpha, \ell) = st(\alpha_1, \ell)$.

- *$type(e_\iota) =$ W, and $\diamond(winner(\check{m}_\iota)) = \pi_i$.*

  Since $\diamond(winner(\check{m}_\iota)) = \pi_i$, then the test on $\langle 19 \rangle$ in iteration ι must have succeeded. Thus, ι is a write create iteration, and we have $\beta = e_\iota$, and $\check{\beta} = \varepsilon$. Also, we have $\iota \neq (i, 0)$. Let $\ell_1 = reg(\check{m}_\iota)$. We claim that for any $m \in M_k \backslash N_1$, $reg(m) \neq \ell_1$. Suppose for contradiction there exists $m \in M_k \backslash N_1$ such that $reg(m) = \ell_1$. Since $m \in M_k \backslash N_1$ and $N_1 = N \cap M_k$, then $m \notin N$. Thus, since N is a prefix and $\check{m}_\iota \in N$, we have $m \not\preceq_\iota \check{m}_\iota$. Then, since $\iota \neq (i, 0)$, we also have $m \not\preceq_{\iota^-} \check{m}_{\iota^-}$. Suppose first that $type(m) =$ R. Since $M_k \subseteq M_\iota$, m is a write metastep on ℓ_1, and $m \not\preceq_{\iota^-} \check{m}_{\iota^-}$, then in $\langle 23 \rangle$ of iteration ι, we have $m \in R_\iota$. But then in $\langle 26 \rangle$ of iteration ι, we set $m \prec_\iota \check{m}_\iota$, a contradiction. Next, suppose that $type(m) = \mathtt{W}$. Then in $\langle 14 \rangle$ in iteration ι, we have $W_\iota \neq \emptyset$, m is a write metastep on ℓ_1, and $m \not\preceq_{\iota^-} \check{m}_{\iota^-}$. So, the test on $\langle 19 \rangle$ of iteration ι fails, which is again a contradiction.

For any $m \in M_k \backslash N_1$, we have shown that $reg(m) \neq \ell_1 = reg(e_\iota)$. Thus, since $\ell \in acc(M_k \backslash N_1)$, we have $st(\alpha^- \circ \beta, \ell) = st(\alpha^- \circ \check{\beta}, \ell)$, and so $st(\alpha, \ell) = st(\alpha_1, \ell)$.

9. *Part 9,* EXTENSION.

   The main idea is to set $\check{\alpha}$ to be a linearization of $N \cap M_k$, then apply part 8 of the lemma. Formally, let $\check{N} = N \cap M_k$, let $\check{\gamma}$ be an execution of $\textsc{Lin}(\check{N}^{\iota^k}, \preceq_k)$, and let $\check{\alpha}$ be the output of $\check{\gamma}$. Let $<_{\check{\gamma}}$ be the $\check{\gamma}$ order of $\check{N}$, and for $m \in \check{N}$, let $<_m$ be the $\check{\gamma}$ order of m^{ι^k}.

   By Lemma 4.6.9, $\check{N}$ is a prefix of $(M_k, \preceq_k)$. Thus, there is a total order $<_k$ on M_k, that extends the total order $<_{\check{\gamma}}$ on $\check{N}$. That is, $<_k$ is a total order on M_k, such that for any $m_1, m_2 \in \check{N}$, we have $m_1 <_k m_2$ if and only if $m_1 <_{\check{\gamma}} m_2$. Choose any such $<_k$, and create the following execution γ_k of $\textsc{Lin}((M_k)^{\iota^k}, \preceq_k)$. γ_k orders M_k using $<_k$. For any $m \in \check{N}$, γ_k linearizes m using $<_m$. For $m \in M_k \backslash \check{N}$, γ_k linearizes m using any output of $\textsc{Seq}(m^{\iota^k})$. Let α_k be the output of γ_k.

   By the definition of γ_k, $\check{\alpha}$ is a prefix of α_k. Write $\alpha_k = \check{\alpha} \circ \beta$. Now, by part 8 of the lemma, for all $h \in [k]$, we have $st(\check{\alpha}, \pi_h) = st(\alpha, \pi_h)$, and for all $\ell \in acc(M_k \backslash \check{N})$, we have $st(\check{\alpha}, \ell) = st(\alpha, \ell)$. Also, by part 1 of the inductive hypothesis, we have $\alpha, \check{\alpha}, \alpha_k \in runs(\mathcal{A})$. Thus, by Theorem 4.3.1, we have $\alpha \circ \beta \in runs(\mathcal{A})$.

   To show the last part of the lemma, let $m \in M_k \backslash N$. Then, since $\check{\alpha} \circ \beta$ contains the linearization of every metastep in M_k, the linearization of m appears somewhere in $\check{\alpha} \circ \beta$. Since $\check{\alpha}$ contains only linearizations of metasteps in $\check{N} \subseteq N$, then the linearization of m must appear in β. $\square$

In the remainder of this chapter, we will refer to different parts of Lemma 4.6.17 using the "dot" notation. For example, we write Lemma 4.6.17.1 for part 1 of Lemma 4.6.17.

Lemma 4.6.17 shows that each iteration of CONSTRUCT satisfies certain safety properties. For example, it shows that a linearization of a prefix from any iteration is a run of $\mathcal{A}$. However, it does not show that CONSTRUCT eventually terminates. In particular, it does not show, for any $i \in [n]$, that there exists an iteration ι such that $e_\iota = \mathsf{rem}_{\pi_i}$, so that the i'th call to GENERATE from CONSTRUCT returns. The following lemma shows that each call to GENERATE does return, from which it follows immediately that CONSTRUCT terminates.

Lemma 4.6.19 (Termination Lemma) *Let $i \in [n]$. Then there exists $j_i \geq 0$ such that $e_{(i,j_i)} = \mathsf{rem}_{\pi_i}$.*

Proof. We use induction on i. Consider $i = 1$, and suppose for contradiction that $e_{(1,j)} \neq \mathsf{rem}_{\pi_1}$, for every $j \geq 0$. Then, since the only process that takes steps in $\alpha_{(1,j)}$ is π_i, $\mathcal{A}$ violates the progress property in Definition 4.3.3, a contradiction. Thus, there exist some j_1 such that $e_{(1,j_1)} = \mathsf{rem}_{\pi_1}$.

Next, assume that the lemma holds up to $i-1$; then we show it also holds for i. Suppose for contradiction that $e_{(i,j)} \neq \mathsf{rem}_{\pi_1}$, for every $j \geq 0$. For every $j \geq 0$, let α_j be an output of $\textsc{Lin}((M_{(i,j)})^{(i,j)}, \preceq_{(i,j)})$. Since $M_{(i,j)}$ is a prefix of $(M_{(i,j)}, \preceq_{(i,j)})$, then by Lemma 4.6.17.1 , we have $\alpha_j \in \mathit{runs}(\mathcal{A})$, for all $j \geq 0$. Since $M_{i-1} \subseteq M_{(i,j)}$ for all $j \geq 0$, then by Lemma 4.6.17.2, we have that $\mathsf{try}_{\pi_k}, \mathsf{enter}_{\pi_k}, \mathsf{exit}_{\pi_k}$ and rem_{π_k} occur in α, for all $k \in [i-1]$. Thus, for every $j \geq 0$, every process π_k, $k \in [i-1]$, is in its remainder region after α_j, except π_i. But this violates the progress property in Definition 4.3.3, a contradiction. Thus, there exist some j_i such that $e_{(i,j_i)} = \mathsf{rem}_{\pi_i}$. $\square$

4.6.5 Main Theorems for Construct

Finally, we show the key property of $\textsc{Construct}$, namely, that in any linearization of $(M, \preceq)$ produced by $\textsc{Construct}(\pi)$, all processes $p_1, p_2, \ldots, p_n$ enter the critical section, and they enter in the order $p_{\pi_1}, p_{\pi_2}, \ldots, p_{\pi_n}$.

Theorem 4.6.20 (Construction Theorem A) *Let α be an output of $\textsc{Lin}(M_n, \preceq_n)$. Then for any $i, j \in [n]$ such that $i < j$, steps enter_{π_i} and enter_{π_j} occur in α, and enter_{π_i} occurs before enter_{π_j}.*

Proof. Suppose for contradiction that there exists $i < j$ such that enter_{π_j} occurs before enter_{π_i} in α. Then the basic idea of the proof is to consider the prefix α_1 of α up to and including the occurrence of enter_{π_j}. Since $i < j$, we can use Lemma 4.6.17.9 to show there exists an extension $\alpha_1 \circ \beta$ of α_1, such that only processes $p_{\pi_1}, \ldots, p_{\pi_i}$ take steps in β. Furthermore, enter_{π_i} occurs in β. But this means there is a prefix of $\alpha_1 \circ \beta$ in which enter_{π_i} and enter_{π_j} have both occurred, but neither exit_{π_i} nor exit_{π_j} has occurred, contradicting the mutual exclusion property in Definition 4.3.3.

We now present the formal proof. First, note that enter_{π_i} and enter_{π_j} both occur in α, by Lemma 4.6.17.2. To show that enter_{π_i} occurs before enter_{π_j}, assume for contradiction otherwise. Let γ be the execution of $\textsc{Lin}(M_n, \preceq_n)$ that produced α, let $<_\gamma$ be the γ order of M, and for each $m \in M$, let $<_m$ be the γ order of m. Let α_1 be the prefix of α up to and including event enter_{π_j}. Let $m_j \in M$ be the critical metastep containing enter_{π_j}, and let $N = \{\mu \,|\, (\mu \in M) \wedge (\mu \leq_\gamma m_j)\}$. N is a prefix of $(M, \preceq)$, since $\leq_\gamma$ is consistent with $\preceq_n$. Let γ_1 be an execution of $\textsc{Lin}(N^{\iota^n}, \preceq)$ defined as follows. γ_1 orders N using $<_\gamma$, and for each $m \in N$, γ_1 orders m using $<_m$. Then, by construction, α_1 is the output of γ_1.

Let $\check{N} = N \cap M_i$, and let $\check{\alpha}$ be an output of $\textsc{Lin}(\check{N}^{\iota^i}, \preceq_i)$. Then by Lemma 4.6.17.9, there exists a run α_i that is an output of $\textsc{Lin}((M_i)^{\iota^i}, \preceq_i)$, such that $\alpha_i = \check{\alpha} \circ \beta$ and $\alpha_1 \circ \beta \in \mathit{runs}(\mathcal{A})$. Since enter_{π_j} occurs before enter_{π_i} in α, and since N consists of all the metasteps that are $\leq_\gamma m_j$, then for all $m \in N$, m does not contain enter_{π_i}. Thus, since enter_{π_i} occurs in α_i by Lemma 4.6.17.2, we have by Lemma 4.6.17.9 that enter_{π_i} occurs in β.

Let α_2 be the prefix of $\alpha_1 \circ \beta$ up to and including enter_{π_i}. Then exit_{π_j} does not occur in α_2, since α_1 only contains the events of π_j up through enter_{π_j}, and β does not contain any events by π_j. Also, exit_{π_i} does not occur in α_2, since $\alpha_1 \circ \beta$ is well formed, and so exit_{π_i} can only occur after enter_{π_i} in $\alpha_1 \circ \beta$. Thus, α_2 contains enter_{π_i} and enter_{π_j}, but does not contain exit_{π_i} or exit_{π_j}. Hence, α_2 violates the mutual exclusion property of $\mathcal{A}$, a contradiction. Thus, we must have that enter_{π_i} occurs before enter_{π_j} in α, for all $i < j$. $\Box$

Finally, since our lower bound deals with the cost of canonical runs, we show that every linearization of $(M_n, \preceq_n)$ is canonical.

Theorem 4.6.21 (Construction Theorem B) *Let α be the output of an execution of* $\textsc{Lin}((M_n)^{\iota^n}, \preceq_n)$. *Then $\alpha \in \mathcal{C}$.*

Proof. Let $i \in [n]$ be arbitrary. Then by Lemma 4.6.17.2, try_i,enter_i, exit_i and rem_i each occur once in α. Also, from the discussion at the end of Section 4.3.2, $\delta(\cdot, i)$ is defined so that after p_i performs enter_i, it performs exit_i in its next step. Since i was arbitrary, then α is a canonical execution. $\Box$

4.7 Additional Properties of Construct

In this section, we prove some additional properties of the Construct algorithm. These properties are used in subsequent sections to prove the correctness of the Encode and Decode algorithms. We begin by introducing some notation.

4.7.1 Notation

Definition 4.7.1 (Function G) *Let ι be any iteration. Define $G((M_\iota)^\iota) = \sum_{m \in M_\iota} |steps(m^\iota)|$ to be the total number of steps contained in all the metasteps in M_ι after iteration ι. Also, let $G = G((M_n)^{\iota^n})$ be the total number of steps contained in all the metasteps in M_n after iteration ι^n.*

Let ι be any iteration, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Recall that the function $\textsc{Lin}(N^\iota, \preceq_\iota)$ is nondeterministic, and may return any run that is a linearization of $(N^\iota, \preceq_\iota)$. The following function $\mathcal{L}(\iota, N)$ is the set of all such linearizations.

Definition 4.7.2 (Function $\mathcal{L}$) *Let ι be any iteration, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Define* $\mathcal{L}(\iota, N) = \{\alpha \mid \alpha \text{ is an output of } \textsc{Lin}(N^\iota, \preceq_\iota)\}$.

Let $\iota = (i, j)$ be any iteration, let $k \in [i]$, and let N be a prefix of $(M_\iota, \preceq_\iota)$. We define $\lambda(\iota, N, k)$ to be the minimum metastep in M_ι (with respect to $\preceq_\iota$) not contained in N, that contains process p_{π_k}. We define $\lambda(\iota, N)$ to be the set of minimal metasteps in M_ι that are not contained in N.

Definition 4.7.3 (Function λ) *Let $\iota = (i, j)$ be any iteration, let $k \in [i]$, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Define the following.*

1. *$\lambda(\iota, N, k) = \min_{\preceq_\iota}\{\mu \,|\, (\mu \in M_\iota \backslash N) \wedge (\pi_k \in procs(\mu^\iota))\}$. We say $\lambda(\iota, N, k)$ is the* next p_{π_k} metastep *after (ι, N).*

2. *$\lambda(\iota, N) = \min_{\preceq_\iota}(M_\iota \backslash N)$. We say $\lambda(\iota, N)$ is the set of* minimal metasteps *after (ι, N).*

Recall that the set of metasteps containing any process is totally ordered by $\preceq_\iota$, by Lemma 4.6.8, and so $\lambda(\iota, N, k)$ is either a metastep, or $\emptyset$.

We define the following. An explanation of the definition follows its formal statement.

Definition 4.7.4 (Next Steps) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $\ell \in L$ and $v \in V$ be arbitrary. For any $k \in [i]$, let $m_k = \lambda(\iota, N, k)$, $s_k = st(\alpha, \pi_k)$, and $e_k = \delta(\alpha, \pi_k)$[28]. Also, let $S_{k,\ell,v} = \{s \,|\, (s \in S) \wedge (st(s, \pi_k) = st(s_k, \pi_k)) \wedge (st(s, \ell) = v)$. We define the following.*

1. *We say e_k is the* next π_k step *after (ι, N).*

2. *If $type(e_k) =$ `R`, then we say π_k* reads ℓ *after (ι, N). If $type(e_k) =$ `W`, then we say π_k* writes to ℓ *after (ι, N).*

3. *Suppose that $type(e_k) =$ `R`, $type(m_k) =$ `W`, and $\ell = reg(e_k)$. Also, suppose that $\exists s \in S_{k,\ell,v} : \Delta(s, e_k, \pi_k) \neq s$. Then we say that π_k* v-reads ℓ *after (ι, N).*

4. *Define $readers(\iota, N, \ell, v)$ to be the set of processes that v-read ℓ after (ι, N).*

5. *Let $wwriters(\iota, N, \ell)$ to be the set of processes that write to ℓ after (ι, N).*

In the above definition, $\ell \in L, v \in V$ and $k \in [i]$ are arbitrary. e_k is the step that π_k performs after α, where α is a linearization of $(N^\iota, \preceq_\iota)$. Depending on whether e_k is a read or write step, we say π_k reads or writes to ℓ after (ι, N). Now, if e_k is a read step, and if m_k, the next π_k metastep after (ι, N), is a write metastep, and if π_k changes its state after reading value v in ℓ, then we say that π_k v-reads ℓ after (ι, N). Note that we do not require that $v = val(m_k)$[29]. We let $readers(\iota, N, \ell, v)$ be the set of processes that v-read ℓ after (ι, N), and we let $wwriters(\iota, N, \ell)$ be the set of processes that write to ℓ after (ι, N). Note that the two w's in the name is intentional[30].

Let ι be any iteration, and let N be a prefix of $(M_\iota, \preceq_\iota)$. In the following definition, $preads(\iota, N, \ell)$ is the set of read metasteps m on ℓ that are contained in N, and such that m is contained in the

[28]Note that s_k and e_k are well defined, because by Lemma 4.6.17.5, we have $st(\alpha_1, \pi_k) = st(\alpha_2, \pi_k)$, for any $\alpha_1, \alpha_2 \in \mathcal{L}(\iota, N)$.

[29]However, we show in Lemma 4.7.13 that π_k does $val(m_k)$-read ℓ after (ι, N). π_k could also v-read ℓ after (ι, N), for some $v \neq val(m_k)$.

[30]We use two w's because $wwriters(\iota, N, \ell)$ may contain both the winning and non-winning write steps in some write metastep on ℓ not in N.

preread set of some (write) metastep that is *not* contained in N. We say any such m is *unmatched*[31]. Formally, we have the following.

Definition 4.7.5 (Unmatched Prereads) *Let ι be an any iteration, $\ell \in L$, and let N be a prefix of $(M_\iota, \preceq_\iota)$. We define*

$$preads(\iota, N, \ell) = \{\mu_1 \,|\, (\mu_1 \in N) \wedge (type(\mu_1) = \texttt{R}) \wedge (reg(\mu_1) = \ell) \wedge (\exists \mu_2 : (\mu_2 \notin N) \wedge (\mu_1 \in preads((\mu_2)^\iota)))\}.$$

For any $m \in preads(\iota, N, \ell)$, we say that m is an unmatched *preread metastep on ℓ after (ι, N).*

4.7.2 Properties for the Encoding

In this section, we prove some properties of Construct that are used in Section 4.9 to show the efficiency of the encoding algorithm. The key lemma in this section is Lemma 4.7.9, which essentially shows that every step in a linearization of $((M_\iota)^\iota, \preceq_\iota)$ causes some process to change its state.

The following lemma states that any (read) metastep is in the preread set of at most one (write) metastep. This is used later to show that Encode does not expend too many bits encoding preread metasteps.

Lemma 4.7.6 (Preread Lemma A) *Let ι be any iteration, and let $m_1, m_2 \in M_\iota$ be such that $m_1 \in preads((m_2)^\iota)$. Then for all $\mu \in M_\iota$ such that $\mu \neq m_2$, we have $m_1 \notin preads(\mu^\iota)$.*

Proof. We use induction on ι. The lemma holds for $\iota = (1, 0)$. We show that if the lemma holds up to $\iota \ominus 1$, then it also holds for ι. Fix $m_1, m_2 \in M_\iota$, and assume that $m_1 \in preads((m_2)^\iota)$. We show that for all $\mu \in M \backslash \{m_2\} : m_1 \notin preads(\mu^\iota)$. Consider two cases, either $\nexists \mu \in M_{\iota^-} : m_1 \in preads(\mu^{\iota^-})$, or $\exists \mu \in M_{\iota^-} : m_1 \in preads(\mu^{\iota^-})$.

1. *Case $\nexists \mu \in M_{\iota^-} : m_1 \in preads(\mu^{\iota^-})$.*

   By Lemma 4.6.3, or by direct inspection of Construct, we can see that only metastep whose *pread* attribute can change during iteration ι is $\check{m}_\iota$. Thus, since $\nexists \mu \in M_{\iota^-} : m_1 \in preads(\mu^{\iota^-})$, we have $\nexists \mu \in M_\iota \backslash \{\check{m}_\iota\} : m_1 \in preads(\mu^\iota)$. Then, since $m_1 \in preads((m_2)^\iota)$, we have $m_2 = \check{m}_\iota$. Thus, the lemma holds.

2. *Case $\exists \mu \in M_{\iota^-} : m_1 \in preads(\mu^{\iota^-})$.*

   Let $m_3 \in M_{\iota^-}$ be such that $m_1 \in preads((m_3)^{\iota^-})$. By the inductive hypothesis, we have that $\forall \mu \in M_{\iota^-} \backslash \{m_3\} : m_1 \notin preads(\mu^{\iota^-})$. We now show that $\forall \mu \in M_\iota \backslash \{m_3\} : m_1 \notin preads(\mu^\iota)$. Let $m \in M_\iota \backslash \{m_3\}$. If $m \in M_{\iota^-}$, then we see by inspection that the *pread* attribute of m does not change during iteration ι, and so $m_1 \notin preads(m^\iota)$.

[31]The reason that we focus on unmatched read metasteps is that one of the necessary conditions for a write metastep m to be a minimal metastep after (ι, N) is that $m \notin N$, and for every read metastep $\mu \in preads(m^\iota)$, we have $\mu \in N$. Thus, a necessary condition for m to be minimal is that all its prereads are unmatched. Please see Lemma 4.7.31.

Next, if $m \notin M_{\iota^-}$, then by Lemma 4.6.3, we have $m = \check{m}_\iota$, and ι is a create iteration. If $\check{m}_\iota$ is not a write metastep on ℓ, where $\ell = reg(m_1)$, then we see from Lemma 4.6.3 that $m_1 \notin preads((\check{m}_\iota)^\iota)$. So, suppose that $\check{m}_\iota$ is a write metastep on ℓ, so that ι is a write create iteration.

We claim that $m_3 \preceq_{\iota^-} \check{m}_{\iota^-}$. Indeed, suppose that $m_3 \npreceq_{\iota^-} \check{m}_{\iota^-}$. Since $m_1 \in preads((m_3)^{\iota^-})$, then we have $reg(m_3) = \ell$, and $type(m_3) = \mathtt{W}$. Thus, in $\langle 15 \rangle$ of iteration ι, we have $m_w \neq \emptyset$, and so ι is a modify iteration, which is a contradiction. Thus, we have $m_3 \preceq_{\iota^-} \check{m}_{\iota^-}$. Now, since $m_1 \in preads((m_3)^{\iota^-})$, we have $m_1 \prec_{\iota^-} m_3$, and so $m_1 \prec_{\iota^-} \check{m}_{\iota^-}$. Thus, from $\langle 23 \rangle$ of ι, we see that $m_1 \notin R_\iota$, since for all $\mu \in R_\iota$, we have $\mu \npreceq_{\iota^-} \check{m}_{\iota^-}$. So, by $\langle 24 \rangle$ of ι, we have $m_1 \notin preads((\check{m}_\iota)^\iota)$. Thus, we have shown that $\forall \mu \in M_\iota \backslash \{m_3\} : m_1 \notin preads(\mu^\iota)$. Finally, since $m_1 \in preads((m_3)^{\iota^-})$, then $m_1 \in preads((m_3)^\iota)$. Since we also have $m_1 \in preads((m_2)^\iota)$, then $m_3 = m_2$, and so the lemma holds.

□

Lemma 4.7.7 (Cost Lemma A) *Let ι be any iteration, and let α be an output of* $\textsc{Lin}((M_\iota)^\iota, \preceq_\iota)$. *Then we have $|\alpha| = G((M_\iota)^\iota)$.*

Proof. This follows by inspection of $\textsc{Lin}((M_\iota)^\iota, \preceq_\iota)$. Indeed, α consists of exactly the set of steps contained in the metasteps contained in M_ι after ι, and so $|\alpha| = G((M_\iota)^\iota)$.

□

The next lemma says that in any linearization of $((M_\iota)^\iota, \preceq_\iota)$, process π_i changes its state after performing its last step e_ι. This lemma is used in Lemma 4.7.9 to show that every step in a run α produced by linearizing $((M_n)^{\iota^n}, \preceq_n)$ incurs unit cost, in the state change model. This fact in turn is used in Section 4.9 to show that the number of bits used to encode $((M_n)^{\iota^n}, \preceq_n)$ is proportional to the cost of α.

Lemma 4.7.8 (State Change Lemma) *Let $\iota = (i, j)$ be any iteration, let α be an output of* $\textsc{Lin}((M_\iota)^\iota, \preceq_\iota)$*, and write $\alpha = \alpha^- \circ e_\iota \circ \alpha^+$, for some step sequences α^- and α^+. Then we have $st(\alpha^- \circ e_\iota, \pi_i) \neq st(\alpha^-, \pi_i)$.*

Proof. The basic idea is the following. Since processes $\pi_1, \ldots, \pi_{i-1}$ do not "see" process π_i, then we have $\alpha^- \circ \alpha^+ \in runs(\mathcal{A})$. At the end of $\alpha^- \circ \alpha^+$, all processes $\pi_1, \ldots, \pi_{i-1}$ are in their remainder sections, and π_i is about to perform step e_ι. Then, if π_i does not change its state after performing e_ι, it will stay in the same state, even after performing an arbitrarily large number of steps, violating the progress property in Definition 4.3.3.

We now present the formal proof. The lemma holds for $\iota = (1, 0)$. Indeed, let $e = \mathsf{try}_{\pi_1}$. Then $e = e_{(1,0)} = \alpha$. We must have $st(\varepsilon, \pi_1) \neq st(\alpha, \pi_1)$, because otherwise, we would have

$e_{(1,1)} = \delta(\alpha, \pi_1) = \delta(\varepsilon, \pi_1) = \mathsf{try}_{\pi_1}$, which violates the well formedness property in Definition 4.3.3. Suppose for induction that the lemma holds up to iteration $\iota \ominus 1$. Then we show it also holds for ι. Consider the following cases, based on the type of e_ι.

1. $type(e_\iota) = \mathtt{C}$.

   If $st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$, then by the same argument as for iteration $(1, 0)$, we have $\delta(\alpha^- \circ e_\iota, \pi_i) = e_\iota$, which is a contradiction.

2. $type(e_\iota) = \mathtt{W}$.

   Suppose for contradiction that $st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$. By Lemma 4.6.10 and 4.6.17.1, we have $\alpha^- \circ \alpha^+ \in runs(\mathcal{A})$. Also, since $M_{i-1} \subseteq M_\iota$, then it follows from 4.6.17.2 that that rem_{π_k} occurs in $\alpha^- \circ \alpha^+$, for all $k \in [i-1]$.

   Since there are no steps by π_i in α^+, then we have $st(\alpha^-, \pi_i) = st(\alpha^- \circ \alpha^+, \pi_i)$, and so $\delta(\alpha^- \circ \alpha^+, \pi_i) = e_\iota$. Then, we have $\alpha^- \circ \alpha^+ \circ e_\iota \in runs(\mathcal{A})$. Since $st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$ and e_ι is a write step, then we have $st(\alpha^- \circ \alpha^+ \circ e_\iota, \pi_i) = st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$, and so $\delta(\alpha^- \circ \alpha^+ \circ e_\iota, \pi_i) = e_\iota$. Thus, we have $\alpha^- \circ \alpha^+ \circ (e_\iota)^2 \in runs(\mathcal{A})$[32], $st(\alpha^- \circ \alpha^+ \circ (e_\iota)^2, \pi_i) = st(\alpha^-, e_\iota)$, and $\delta(\alpha^- \circ \alpha^+ \circ (e_\iota)^2, \pi_i) = e_\iota$, etc. From this, we see that π_i stays in the same state in all extensions of $\alpha^- \circ \alpha^+$. All these extensions are fair, since $\pi_1, \ldots, \pi_{i-1}$ are in their remainder regions following $\alpha^- \circ \alpha^+$. Thus, this violates the progress property in Definition 4.3.3, a contradiction. So, we must have $st(\alpha^- \circ e_\iota, \pi_i) \neq st(\alpha^-, \pi_i)$.

3. $type(e_\iota) = \mathtt{R}$.

   Consider two cases, either $type(\check{m}_\iota) = \mathtt{W}$, or $type(\check{m}_\iota) = \mathtt{R}$. Let $\ell = reg(e_\iota)$.

   If $type(\check{m}_\iota) = \mathtt{W}$, then let $v = val(\check{m}_\iota)$. In α, e_ι reads the value v in ℓ. Let $s \in S$ be any system state such that $st(s, \pi_i) = st(\alpha_\iota, \pi_i)$, and $st(s, \ell) = v$. From $\langle 29 \rangle$ of Construct, we have that $\Delta(s, e_\iota, \pi_i) \neq st(s, \pi_i)$. Let $N \subseteq M_\iota$ be the set of metasteps that are linearized before $\check{m}_\iota$ in α. We can see that $\Phi(\iota^-, N, i) = \Phi(\iota^-, N_\iota, i)$. So, since $\alpha_\iota \in \mathcal{L}(\iota^-, N_\iota)$, then by Lemma 4.6.17.5, we have $st(\alpha^-, \pi_i) = st(s, \pi_i)$. Thus, we have $st(\alpha^- \circ e_\iota, \pi_i) \neq st(\alpha^-, \pi_i)$.

   Next, consider the case when $type(\check{m}_\iota) = \mathtt{R}$, and suppose for contradiction that $st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$. We have $\alpha^- \circ \alpha^+ \in runs(\mathcal{A})$, and rem_{π_k} occurs in $\alpha^- \circ \alpha^+$, for all $k \in [i-1]$. Since there are no steps by π_i in α^+, we have $st(\alpha^-, \pi_i) = st(\alpha^- \circ \alpha^+, \pi_i)$, and so $\delta(\alpha^- \circ \alpha^+, \pi_i) = e_\iota$, and $\alpha^- \circ \alpha^+ \circ e_\iota \in runs(\mathcal{A})$.

   Since $type(\check{m}_\iota) = \mathtt{R}$, then by Lemma 4.6.17.3, we have $W_\iota = \emptyset$. Thus, there are no write steps on ℓ in α^+. Thus, since $st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$, we have $st(\alpha^- \circ \alpha^+ \circ e_\iota, \pi_i) = st(\alpha^- \circ e_\iota, \pi_i) = st(\alpha^-, \pi_i)$, and so $\delta(\alpha^- \circ \alpha^+ \circ e_\iota, \pi_i) = e_\iota$. Then, we have $\alpha^- \circ \alpha^+ \circ (e_\iota)^2 \in runs(\mathcal{A})$,

[32]For any $r \in \mathbb{N}$, we let $(e_\iota)^r$ denote $e_\iota \circ \ldots \circ e_\iota$, where there are r occurrences of e_ι.

$st(\alpha^- \circ \alpha^+ \circ (e_\iota)^2, \pi_i) = st(\alpha^-, e_\iota)$, and $\delta(\alpha^- \circ \alpha^+ \circ (e_\iota)^2, \pi_i) = e_\iota$, etc. Thus, π_i stays in the same state in all extensions of $\alpha^- \circ \alpha^+$. All extensions of $\alpha^- \circ \alpha^+$ are fair, since $\pi_1, \ldots, \pi_{i-1}$ are in their remainder regions following $\alpha^- \circ \alpha^+$. But this contradicts the progress property in Definition 4.3.3. So, we must have $st(\alpha^- \circ e_\iota, \pi_i) \neq st(\alpha^-, \pi_i)$.

□

The next lemma says that the state change cost of any execution α is equal to the length of α. It uses Lemma 4.7.8, which showed that every step in α causes some process to change its state.

Lemma 4.7.9 (Cost Lemma B) *Let $\iota = (i,j)$ be any iteration, and let α be an output of* $\text{Lin}((M_\iota)^\iota, \preceq_\iota)$*. Then we have $C(\alpha) = |\alpha|$.*

Proof. We use induction on ι. The lemma holds for $\iota = (1,0)$, by Lemma 4.7.8. Suppose for induction that the lemma holds up to iteration $\iota \ominus 1$. Then we show that it also holds for ι. Write $\alpha = \alpha^- \circ e_\iota \circ \alpha^+$, and let $\breve{\alpha} = \alpha^- \circ \alpha^+$. By Lemma 4.6.10, $\breve{\alpha}$ is an output of $\text{Lin}((M_{\iota^-})^{\iota^-}, \preceq_{\iota^-})$,and so by the inductive hypothesis, we have $C(\breve{\alpha}) = |\breve{\alpha}|$. Also, we have $|\alpha| = |\breve{\alpha}| + 1$. By Lemma 4.7.8, we have $st(\alpha^- \circ e_\iota, \pi_i) \neq st(\alpha^-, \pi_i)$. Thus, from Definition 4.3.6, we have $C(\alpha^- \circ e_\iota) = C(\alpha^-) + 1$. By Lemma 4.6.11, we have $\forall k \in [i-1] : st(\alpha^-, \pi_k) = st(\alpha^- \circ e_\iota, \pi_k)$ and $\forall \ell \in acc(\alpha^+) : st(\alpha^-, \ell) = st(\alpha^- \circ e_\iota, \ell)$. Also, there are no steps by π_i in α^+. Thus, we have

$$\begin{aligned}
C(\alpha) &= C(\alpha^- \circ e_\iota \circ \alpha^+) \\
&= C(\alpha^- \circ \alpha^+) + 1 \\
&= C(\breve{\alpha}) + 1 \\
&= |\breve{\alpha}| + 1 \\
&= |\alpha|.
\end{aligned}$$

□

Lemma 4.7.10 (Cost Lemma C) *Let α be an output of* $\text{Lin}((M_n)^{\iota^n}, \preceq_n)$*. Then we have $C(\alpha) = G$.*

Proof. We have $G = |\alpha| = C(\alpha)$, where the first equality follows by Lemma 4.7.7, and the second equality follows by Lemma 4.7.9. □

4.7.3 Properties for the Decoding

In this section, we prove some properties of Construct that are used in Section 4.11 to show the correctness of the Decode algorithm. At the end of this section, we recap the properties, and describe how they suggest the decoding strategy used by Decode in Section 4.10.

In the exposition in the remainder of this section, let $\iota = (i, j)$ be an arbitrary iteration, let $k \in [i]$, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Also, let $\alpha \in \mathcal{L}(\iota, N)$, and let $\check{N} = N \cap M_{\iota^-}$.

The following lemma says that unless $k = i$ and $\check{m}_{\iota^-} \in N$, then the next π_k metastep after (ι, N) and after $(\iota^-, \check{N})$ are the same.

Lemma 4.7.11 (λ Lemma A) *Let $\iota = (i, j)$ be any iteration, let $k \in [i]$, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\check{N} = N \cap M_{\iota^-}$. Suppose that $k \neq i$, or $\check{m}_{\iota^-} \notin N$. Then we have $\lambda(\iota, N, k) = \lambda(\iota^-, \check{N}, k)$.*

Proof. By assumption, we either have $k \in [i - 1]$, or $\check{m}_{\iota^-} \notin N$. Since N is a prefix of $(M_\iota, \preceq_\iota)$ and $\check{m}_{\iota^-} \preceq_\iota \check{m}_\iota$, we get that either $k \in [i - 1]$, or $\check{m}_\iota \notin N$. Then, by Lemma 4.6.9, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$. From this, and from Lemma 4.6.8, we get that $\lambda(\iota, N, k) = \lambda(\iota^-, \check{N}, k)$. □

The next lemma states a type of consistency condition. It says that the next π_k step after (ι, N) equals the step that π_k takes in the next π_k metastep after (ι, N).

Lemma 4.7.12 (Step Lemma A) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $k \in [i]$, and let $e = \delta(\alpha, \pi_k)$. Let $m = \lambda(\iota, N, k)$, and let $\epsilon = step(m^\iota, \pi_k)$. Then $e = \epsilon$.*

Proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if it true up to iteration $\iota \ominus 1$, then it is true for ι. Let $\check{N} = N \cap M_{\iota^-}$, let $\check{\alpha} \in \mathcal{L}(\iota^-, \check{N})$, and consider three cases, either $k \neq i$ or $\check{m}_{\iota^-} \notin N$, or $k = i$ and $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$, or $k = i$ and $\check{m}_\iota \in N$.

1. *Case $k \neq i$ or $\check{m}_{\iota^-} \notin N$.*

   Let $m_0 = \lambda(\iota^-, \check{N}, k)$, $e' = \delta(\check{\alpha}, \pi_k)$, and $\epsilon' = step((m_0)^{\iota^-}, \pi_k)$. By Lemma 4.6.9, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$, and so by Lemma 4.6.17.5, we have $st(\alpha, \pi_k) = st(\check{\alpha}, \pi_k)$. Thus, we have $e = e'$. Since $k \neq i$ or $\check{m}_{\iota^-} \notin N$, then by Lemma 4.7.11, we have $m = m_0$. Then, since $m^\iota = (m_0)^\iota = (m_0)^{\iota^-}$ by Lemma 4.6.3, we have $\epsilon = \epsilon'$. By the inductive hypothesis, we have $e' = \epsilon'$. Thus, we have $e = \epsilon$.

2. *Case $k = i$ and $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$.*

   By Lemma 4.6.3, we have $e_\iota = \delta(\alpha_\iota, \pi_i)$. By definition, e_ι is the step of π_i contained in $(\check{m}_\iota)^\iota$, and so $e_\iota = \epsilon$.

   Since $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$, then we have $\Phi(\iota, N, i) = \Phi(\iota^-, N_\iota, i)$. So, by Lemma 4.6.17.5, we have $st(\alpha, \pi_i) = st(\alpha_\iota, \pi_i)$. Thus, we have

$$e = \delta(\alpha, \pi_i) = \delta(\alpha_\iota, \pi_i) = e_\iota = \epsilon.$$

3. *Case $k = i$ and $\check{m}_\iota \in N$.*

   Since $\check{m}_\iota$ is the maximum metastep containing π_i, with respect to $\preceq_\iota$, by Lemma 4.6.8, then we have $m = \lambda(\iota, N, i) = \emptyset$. Thus, there is nothing to prove.

□

The following lemma states another consistency condition. It says that if the next π_k metastep after (ι, N) is a write metastep writing value v to a register ℓ, and if the next π_k step after (ι, N) is a read, then π_k v-reads ℓ after (ι, N).

Lemma 4.7.13 (Step Lemma B) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $e = \delta(\alpha, \pi_k)$, and let $m = \lambda(\iota, N, k)$. Suppose $type(e) = \mathtt{R}$ and $type(m) = \mathtt{W}$. Let $\ell = reg(m)$, $v = val(m)$, and let $s \in S$ be such that $st(s, \pi_k) = st(\alpha, \pi_k)$ and $st(s, \ell) = v$. Then we have $\Delta(s, e, \pi_k) \neq st(\alpha, \pi_k)$.*

Proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if it true up to iteration $\iota \ominus 1$, then it is true for ι. Let $\check{N} = N \cap M_{\iota^-}$, let $\check{\alpha} \in \mathcal{L}(\iota^-, \check{N})$, and consider three cases, either $k \neq i$ or $\check{m}_{\iota^-} \notin N$, or $k = i$ and $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$, or $k = i$ and $\check{m}_\iota \in N$.

1. *Case $k \neq i$ or $\check{m}_{\iota^-} \notin N$.*

   Let $m_0 = \lambda(\iota^-, \check{N}, k)$, $\ell' = val(m_0)$, $v' = val(m_0)$, and $e' = \delta(\check{\alpha}, \pi_k)$. By Lemma 4.7.11, we have $m = m_0$, and so $\ell = \ell'$, and $v = v'$. Let $s' \in S$ be such that $st(s', \pi_k) = st(\check{\alpha}, \pi_k)$ and $st(\ell, \pi_k) = v$. Then by the inductive hypothesis, we have $\Delta(s', e', \pi_k) \neq st(\check{\alpha}, \pi_k)$. We have $\Phi(\iota, N, i) = \Phi(\iota, \check{N}, i)$, and so by Lemma 4.6.17.5, we have $st(\alpha, \pi_i) = st(\check{\alpha}, \pi_i)$. Thus, we have $\Delta(s, e, \pi_k) \neq st(\alpha, \pi_k)$.

2. *Case $k = i$ and $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$.*

   By Lemma 4.6.3, we have $e_\iota = \delta(\alpha_\iota, \pi_i)$, and $e_\iota = step((\check{m}_\iota)^\iota, \pi_k)$. Since $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$, we have $m = \lambda(\iota, N, i) = \check{m}_\iota$, and so by Lemma 4.7.12, we have $e = e_\iota$. Since $type(m) = \mathtt{W}$ and $type(e) = \mathtt{R}$, we have that ι is a read modify iteration, and so $\check{m}_\iota \in M_{\iota^-}$. Then, we have $\ell = reg((\check{m}_\iota)^\iota) = reg((\check{m}_\iota)^{\iota^-})$, and $v = val((\check{m}_\iota)^\iota) = val((\check{m}_\iota)^{\iota^-})$. From $\langle 30 \rangle$ of iteration ι, we see that $\check{m}_\iota$ was chosen so that

$$\exists s : (s \in S) \wedge (st(s, \pi_k) = st(\alpha_\iota, \pi_k)) \wedge (st(s, \ell) = v) \wedge (\Delta(s, e_\iota, \pi_k) \neq st(s, \pi_k)).$$

   We have $\Phi(\iota, N, i) = \Phi(\iota^-, N_\iota, i)$, and so $st(\alpha, \pi_i) = st(\alpha_\iota, \pi_i)$, by Lemma 4.6.17.5. Thus, since $e = e_\iota$, we have $\Delta(s, e, \pi_k) \neq st(\alpha, \pi_k)$.

3. *Case $k = i$ and $\check{m}_\iota \in N$.*

   We have $m = \lambda(\iota, N, i) = \emptyset$, and so there is nothing to prove.

□

The next lemma says that, roughly speaking, if the next steps after a prefix for two processes access the same register, then the next metasteps for the processes after the prefix is the same. More precisely, let $h \in [i]$. Let m_k and m_h be the next π_k and π_h metastep after (ι, N), respectively (assume that both m_k and m_h exist). Suppose that both m_h and m_k are write metasteps, and that m_k writes a value v to a register ℓ. Also, suppose that next π_k step after (ι, N) is a write step. Then, the lemma says that if π_h either writes to ℓ, or v-reads ℓ after (ι, N), then we have $m_h = m_k$. Also, in both cases, we have $\pi_h \in procs((m_k)^\iota)$.

Lemma 4.7.14 (λ Lemma B) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $k \in [i]$, $m_k = \lambda(\iota, N, k)$, $\ell = reg(m_k)$, $v = val(m_k)$, and $e_k = \delta(\alpha, \pi_k)$. Let $h \in [i]$, $m_h = \lambda(\iota, N, h)$, and $e_h = \delta(\alpha, \pi_h)$. Suppose that the following hold.*

1. *$m_k, m_h \neq \emptyset$.*

2. *$type(e_k) = type(m_k) = \mathtt{W}$.*

3. *$reg(m_h) = \ell$.*

Then we have the following.

1. *If $type(e_h) = \mathtt{W}$, then $m_h = m_k$, and $\pi_h \in writers((m_k)^\iota) \cup winner((m_k)^\iota)$.*

2. *If $type(e_h) = \mathtt{R}$ and $type(m_h) = \mathtt{W}$, then let $s \in S$ be such that $st(s, \pi_h) = st(\alpha, \pi_h)$ and $st(s, \ell) = v$. If $\Delta(s, e_h, \pi_h) \neq st(\alpha, \pi_h)$, then $m_h = m_k$, and $\pi_h \in readers((m_k)^\iota)$.*

Proof. The proof is by induction on ι. The main idea is the following. Let $\check{N} = N \cap M_{\iota^-}$, and let $m'_h = \lambda(\iota^-, \check{N}, h)$ and $m'_k = \lambda(\iota^-, \check{N}, k)$ be the next π_h and π_k metasteps after $(\iota^-, \check{N})$, respectively. We can show using Lemma 4.7.11 that either $m'_h = m_h$, or $h = i$ and $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$. Similarly, we can show that either $m'_k = m_k$, or $k = i$ and $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$.

If we have $m'_h = m_h$ and $m'_k = m_k$, then we can prove the lemma using the inductive hypothesis. This is case $1a$ in the formal proof below. Case $1b$ considers when $m'_h = m_h$ and $m'_k \neq m_k$. Here, as stated earlier, we have $k = i$, and so $m_k = \check{m}_\iota$. If e_h is a write step, then we will switch the names of k and h, to get $m'_k = m_k$, $h = i$ and $m'_h \neq m_h$. We describe how to deal with this case in the following paragraph. If instead, e_h is a read step, then we show that there exists a $g \neq k = i$ such that π_g is the winner of $\check{m}_\iota$. We will create a prefix N_2 of $(M_\iota, \preceq_\iota)$ such that $\check{m}_\iota$ is the next π_g metastep after (ι, N_2). In addition, m_h is the next π_h metastep after (ι, N_2). Now, since $g, h < i$, we can apply case $1a$ of the lemma to conclude that $m_k = m_h$.

Finally, we describe the case when $m'_k = m_k$, $h = i$ and $m'_h \neq m_h$. This is case 2 in the formal proof. We show in Claim 4.7.20 that m_k is the minumum write metastep on ℓ not in N. Since $h = i$

and $m'_h \neq m_h$, we have $m_h = \check{m}_\iota$. Now, if e_h is a write step, we show in Claim 4.7.21 that e_h is added to the write steps of m_k in iteration ι. Basically, the reason for this is that, since m_k is the minimum write metastep on ℓ not in N, and since $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$, then m_k is in fact the minimum write metastep on ℓ not in N_ι. Then, it follows from $\langle 15 \rangle$ of Construct that e_h is added to the writes of m_k. Thus, we have $m_h = \check{m}_\iota = m_k$, and $\pi_h \in \mathit{writers}((m_k)^\iota) \cup \mathit{winner}((m_k)^\iota)$. If e_h is a read step, and π_h changes its state after reading the value of m_k, then using similar reasoning as above, we show in Claim 4.7.22 that e_h is added to the read steps of m_k. So again, we have $m_h = \check{m}_\iota = m_k$, and $\pi_h \in \mathit{readers}((m_k)^\iota)$.

We now present the formal proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if the lemma true up to iteration $\iota \ominus 1$, then it is also true for ι. Let $\check{N} = N \cap M_{\iota^-}$, and let $\check{\alpha} \in \mathcal{L}(\iota^-, \check{N})$. It suffices to assume that $k \neq h$. Also, we claim that it suffices to consider two cases, either $(h \neq i) \vee (\check{m}_{\iota^-} \notin N)$, or $(h = i) \wedge (\check{m}_{\iota^-} \in N) \wedge (\check{m}_\iota \notin N)$. In particular, we do not need to consider the case $(h = i) \wedge (\check{m}_{\iota^-} \in N) \wedge (\check{m}_\iota \in N)$, because here, we have $m_h = \lambda(\iota, N, h) = \emptyset$, since by Lemma 4.6.8, $\check{m}_\iota$ is the maximum (with respect to $\preceq$) metastep containing $\pi_h = \pi_i$.

1. *Case $h \neq i$ or $\check{m}_{\iota^-} \notin N$.*

   Let $m'_k = \lambda(\iota^-, \check{N}, k)$, and $m'_h = \lambda(\iota^-, \check{N}, h)$. Since $(h \neq i) \vee (\check{m}_{\iota^-} \notin N)$, we have $m_h = m'_h$, by Lemma 4.7.11. By Lemma 4.6.9, we have $\Phi(\iota, N, h) = \Phi(\iota^-, \check{N}, h)$, and so by Lemma 4.6.17.5, we have $st(\check{\alpha}, \pi_h) = st(\alpha, \pi_h)$, and $e'_h = \delta(\check{\alpha}, \pi_h) = \delta(\alpha, \pi_h) = e_h$. Consider the two following cases.

   (a) *Case $k \neq i$ or $\check{m}_{\iota^-} \notin N$.*

   In this case, we have $m_k = m'_k$, by Lemma 4.7.11. Consider the following cases.

   Suppose first that $\mathit{type}(e_h) = \mathtt{W}$. Then $\mathit{type}(e'_h) = \mathtt{W}$, and so by the inductive hypothesis, we have $m'_h = m'_k$, and $\pi_h \in \mathit{writers}((m'_k)^{\iota^-}) \cup \mathit{winner}((m'_k)^{\iota^-})$. Thus, we have $m_h = m'_h = m'_k = m_k$, and $\pi_h \in \mathit{writers}((m_k)^\iota) \cup \mathit{winner}((m_k)^\iota)$.

   Suppose next that $\mathit{type}(e_h) = \mathtt{R}$, $\mathit{type}(m_h) = \mathtt{W}$, and $\Delta(s, e_h, \pi_h) \neq st(\alpha, \pi_h)$. Let $s' \in S$ be such that $st(s', \pi_h) = st(\check{\alpha}, \pi_h)$, and $st(s', \ell) = v$. Then we have $\mathit{type}(e'_h) = \mathtt{R}$, $\mathit{type}(m'_h) = \mathtt{W}$, and $\Delta(s', e'_h, \pi_h) \neq st(\check{\alpha}, \pi_h)$, and so by the inductive hypothesis we have $m'_h = m'_k$, and $\pi_h \in \mathit{readers}((m'_k)^{\iota^-})$. Thus, we have $m_h = m'_h = m'_k = m_k$, and $\pi_h \in \mathit{readers}((m_k)^\iota)$.

   (b) *Case $k = i$, $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$.*

   In this case, we have $m_k = \check{m}_\iota$. Also, since $\check{m}_\iota$ is a write metastep, then $\iota \neq (i, 0)$. Consider the following.

   Suppose first that $\mathit{type}(e_h) = \mathtt{W}$. Then we have $k = i$ and $h < i$. We will switch the names of k and h, so that $h = i$ and $k < i$. This then becomes case 2 of the proof, which is presented later.

Next, suppose that $type(e_h) = \mathtt{R}$. We have already shown that the lemma holds in case $1a$, after iteration ι. Our goal is to apply this fact to show that the lemma also holds after iteration ι in case $1b$, and when $type(e_h) = \mathtt{R}$. We have the following.

Claim 4.7.15 $m_h \not\preceq_{\iota^-} \check{m}_{\iota^-}$.

Proof. Suppose instead that $m_h \preceq_{\iota^-} \check{m}_{\iota^-}$. Then $m_h \preceq_\iota \check{m}_{\iota^-}$, by Lemma 4.6.4. Since $\lambda(\iota, N, k) = \check{m}_\iota$ and $\iota \neq (i, 0)$, we have $\check{m}_{\iota^-} \in N$. Thus, since N is a prefix of $(M_\iota, \preceq_\iota)$, we have $m_h \in N$, which is a contradiction. □

Claim 4.7.16 *ι is a write modify iteration.*

Proof. Since m_h is a write metastep on ℓ, and $m_h \not\preceq_{\iota^-} \check{m}_{\iota^-}$ by Claim 4.7.15, we have $m_h \in W_\iota$, and so $W_\iota \neq \emptyset$. Then, from $\langle 15 \rangle$ of iteration ι, we see that $m_w \neq \emptyset$. So, the test on $\langle 16 \rangle$ of ι succeeds, and ι is a write modify iteration. □

Claim 4.7.17 $m_h \not\preceq_\iota \check{m}_\iota$.

Proof. From $\langle 15 \rangle$ of iteration ι, we have $\check{m}_\iota = \min_{\preceq_{\iota^-}} W_\iota$. Then, it follows from Lemma 4.6.3 that $\check{m}_\iota = \min_{\preceq_\iota} W_\iota$. By Claim 4.7.15, we have $m_h \not\preceq_{\iota^-} \check{m}_{\iota^-}$, and so since m_h is a write metastep on ℓ, we have $m_h \in W_\iota$. Thus, since $\check{m}_\iota = \min_{\preceq_\iota} W_\iota$, we have $m_h \not\preceq_\iota \check{m}_\iota$. □

Finally, we show that $\check{m}_\iota = m_h$, and $m_h \in readers((\check{m}_\iota)^\iota)$. Let $\pi_g = \diamond(winner(\check{m}_\iota))$. Then $g < k = i$, since ι is a write modify iteration by Claim 4.7.16. Now, let

$$N_1 = \{\mu \,|\, (\mu \in M_\iota) \wedge (\mu \prec_\iota \check{m}_\iota)\}, \qquad N_2 = N_1 \cup N.$$

N_1 is a prefix of $(M_\iota, \preceq_\iota)$, and N_2 is also a prefix of $(M_\iota, \preceq_\iota)$, since the union of two prefixes is a prefix. We have $\check{m}_\iota \notin N$ and $\check{m}_\iota \notin N_1$, and so $\check{m}_\iota \notin N_2$. Thus, since the set of π_g metasteps is totally ordered by $\preceq_\iota$, by Lemma 4.6.8, and since N_1 contains all the metasteps in $\mu \in M_\iota$ that $\mu \prec_\iota \check{m}_\iota$, we have $\check{m}_\iota = \lambda(\iota, N_2, g)$. Let $\alpha_2 \in \mathcal{L}(\iota, N_2)$, and let $e_g = \delta(\alpha_2, \pi_g)$. Then since $\pi_g = \diamond(winner(\check{m}_\iota))$, we have $type(e_g) = \mathtt{W}$.

Next, we have $m_h \notin N$, and $m_h \not\preceq_\iota \check{m}_\iota$ by Claim 4.7.17, and so $m_h \notin N_2$. Thus, since $m_h = \lambda(\iota, N, h)$ and $N \subseteq N_2$, we have $m_h = \lambda(\iota, N_2, h)$. Then, we have $\Phi(\iota, N, h) = \Phi(\iota, N_2, h)$, and so by Lemma 4.6.17.5, we have $st(\alpha_2, \pi_h) = st(\alpha, \pi_h)$. Let $e''_h = \delta(\alpha_2, \pi_h)$. Then, we have $e''_h = e_h$. Let $s_2 \in S$ be such that $st(s_2, \pi_h) = st(\alpha_2, \pi_h)$ and $st(s_2, \ell) = v$. Together with the earlier statements and assumptions, we get the following.

$$g, h < i, \qquad \check{m}_\iota = \lambda(\iota, N_2, g), \qquad m_h = \lambda(\iota, N_2, h), \qquad \ell = reg(\check{m}_\iota),\ v = val(\check{m}_\iota),$$

$$type(\check{m}_\iota) = type(e_g) = \mathtt{W}, \qquad type(m_h) = \mathtt{W},\ type(e''_h) = \mathtt{R},$$

$$\Delta(s_2, e''_h, \pi_h) \neq st(\alpha_2, \pi_h).$$

Now, since we have already proved case $1a$ of the lemma for iteration ι, then we see that by setting "k" in the assumptions of the lemma to "g", we get that $m_h = \check{m}_\iota = m_k$, and $\pi_h \in readers((\check{m}_\iota)^\iota)$. Thus, the lemma is proved.

2. *Case $h = i$, $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$.*

In this case, we have $k < i$, and

$$m_h = \lambda(\iota, N, i) = \check{m}_\iota. \tag{4.9}$$

Since $\check{m}_\iota$ is a write metastep, then $\iota \neq (i, 0)$. Let

$$W = \{\mu \,|\, (\mu \in M_{\iota^-} \backslash N) \wedge (type(\mu) = \mathtt{W}) \wedge (reg(\mu) = \ell)\}.$$

We have $m_k \in W$, and so $W \neq \emptyset$. By Lemma 4.6.17.6, all metasteps in W are totally ordered. Let $m_1 = \min_{\preceq_\iota} W$.

We denote the two cases in the conclusions of the lemma as follows. Let $(C1)$ denote the event that $type(e_h) = \mathtt{W}$, and let $(C2)$ denote the event that $type(e_h) = \mathtt{R}$ and $type(m_h) = \mathtt{W}$ and $\Delta(s, e_h, \pi_h) \neq st(\alpha, \pi_h)$. We have the following.

Claim 4.7.18 $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$.

Proof. Suppose instead that $m_k \preceq_{\iota^-} \check{m}_{\iota^-}$. Since $m_k = \lambda(\iota, N, k)$, we have $m_k \notin N$. But since $\check{m}_{\iota^-} \in N$ and N is a prefix, we also have $m_k \in N$, a contradiction. Thus, $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$. □

Claim 4.7.19 *Suppose $(C1)$ or $(C2)$ hold. Then ι is a modify iteration.*

Proof. Suppose first that $(C1)$ holds. Then since $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$ by Claim 4.7.18, and m_k is a write metastep on ℓ, we have $W_\iota \neq \emptyset$. Then, from $\langle 15 \rangle$ of iteration ι, we have $m_w \neq \emptyset$, and so ι is a write modify iteration. If $(C2)$ holds, then since $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$, m_k is a write metastep on ℓ, and $\Delta(s, e_h, \pi_h) \neq st(\alpha, \pi_h)$, we have $W^s_\iota \neq \emptyset$. Then in $\langle 30 \rangle$ of ι, we have $m_{ws} \neq \emptyset$, and ι is a read modify iteration. □

Claim 4.7.20 *Suppose $(C1)$ or $(C2)$ hold. Then $m_k = m_1$.*

Proof. Let $\pi_g = \diamond(winner((m_1)^\iota))$. Since m_1 is a write metastep, then π_g exists.

$$N_1 = \{\mu \,|\, (\mu \in M_\iota) \wedge (\mu \prec_\iota m_1)\}, \qquad N_2 = N_1 \cup N.$$

Then N_1 and N_2 are both prefixes of $(M_\iota, \preceq_\iota)$. We have $m_1 = \lambda(\iota, N_2, g)$. Let $\alpha_2 \in \mathcal{L}(\iota, N_2)$, let $e_g = \delta(\alpha_2, \pi_g)$, and let $\epsilon_g = step((m_1)^\iota, \pi_g)$. Since $\pi_g = \diamond(winner((m_1)^\iota))$, then $type(\epsilon_g) = \mathtt{W}$. Also, we have $\epsilon_g = e_g$, by Lemma 4.7.12. Thus, we have $type(e_g) = \mathtt{W}$.

To show $m_k = m_1$, we first claim that $g \neq i$. Indeed, if $g = i$, then $\pi_i = \diamond(winner((m_1)^\iota))$, and so from $\langle 20\rangle$ of ι, we have that ι is a write create iteration, contradicting Claim 4.7.19. Next, we claim that $m_k = \lambda(\iota, N_2, k)$. This follows because m_k is a write metastep not in N, and so $m_k \not\prec_\iota m_1 = \min_{\preceq_\iota} W$. Let $e''_k = \delta(\alpha_2, \pi_k)$. We have $\Phi(\iota, N, k) = \Phi(\iota, N_2, k)$, and so $e''_k = e_k$ using 4.6.17.5.

Now, we have $g, k \neq i$, $e_g = \delta(\alpha_2, \pi_g)$, $e''_k = \delta(\alpha_2, \pi_k)$, $type(e_g) = type(e''_k) = \mathtt{W}$, and $m_1 = \lambda(\iota, N_2, g)$ and $m_k = \lambda(\iota, N_2, k)$. Then, from the case $1a$ in the proof of the lemma, we have $m_1 = m_k$. □

Claim 4.7.21 *Suppose $(C1)$ holds. Then $m_h = m_k$, and $\pi_h \in writers((m_k)^\iota) \cup winner((m_k)^\iota)$.*

Proof. Since $(C1)$ holds, then from $\langle 15\rangle$ of iteration ι, we get that

$$\check{m}_\iota = \min_{\preceq_{\iota^-}} W_\iota = \min_{\preceq_\iota} W_\iota.$$

The second equality follows because ι is a modify iteration, by Claim 4.7.19. We have $m_k \in W_\iota$, since m_k is a write metastep on ℓ, and $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$ by Claim 4.7.18. Thus, we have $\check{m}_\iota \preceq_\iota m_k$. Also, since $\check{m}_\iota \notin N$, and since $\check{m}_\iota$ is a write metastep on ℓ, we have $\check{m}_\iota \in W$. So, $\min_{\preceq_\iota} W = m_1 \preceq_\iota \check{m}_\iota$. Then, since $m_1 = m_k$ by Claim 4.7.20, and since $m_h = \check{m}_\iota$, we have $m_h = \check{m}_\iota = m_k$. Finally, we have $\pi_h \in writers((\check{m}_\iota)^\iota) \cup winner((\check{m}_\iota)^\iota) = writers((m_k)^\iota) \cup winner((m_k)^\iota)$, where the inclusion follows from $\langle 17\rangle$ of iteration ι. □

Claim 4.7.22 *Suppose $(C2)$ holds. Then $m_h = m_k$, and $\pi_h \in readers((m_k)^\iota)$.*

Proof. Since $\check{m}_{\iota^-} \in N$ and $\check{m}_\iota \notin N$, then we have $\Phi(\iota^-, N_\iota, h) = \Phi(\iota, N, h)$. Thus, it follows from Lemma 4.6.17.5 that

$$\forall \mu \in M_{\iota^-} : \mathrm{SC}(\alpha_\iota, \mu, \pi_h) \Leftrightarrow \mathrm{SC}(\alpha, \mu, \pi_h) \tag{4.10}$$

Here, the function SC (state change) is defined as in $\langle 60\rangle$ of Construct. Since $(C2)$ holds, then from ⟨30⟩ of iteration ι and from Equation 4.10, we get that

$$\check{m}_\iota = \min_{\preceq_{\iota^-}} W_\iota^s = \min_{\preceq_\iota} W_\iota^s.$$

The second equality follows because by Claim 4.7.19, ι is a modify iteration. Since $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$ by Claim 4.7.18, and since $\Delta(s, e_h, \pi_h) \neq st(\alpha, \pi_h)$ by $(C2)$, then $m_k \in W_\iota^s$. Thus, we have $\check{m}_\iota \preceq_\iota m_k$. Also, since $\check{m}_\iota \notin N$, then $\check{m}_\iota \in W$. So, since $m_k = m_1 = \min_{\preceq_\iota} W$ by Claim 4.7.20, we have $m_k \preceq_\iota \check{m}_\iota$. Thus, we have $m_k = \check{m}_\iota = m_h$. Finally, we have $\pi_h \in readers((\check{m}_\iota)^\iota) = readers((m_k)^\iota)$, where the inclusion follows from ⟨32⟩ of iteration ι. □

Combining Claims 4.7.21 and 4.7.22, the lemma is proved.

□

Let e_k and m_k be the next π_k step and metastep after (ι, N), and suppose that e_k and m_k are both writes to a register ℓ. Let $h \in [i]$, and suppose m_h is a read metastep containing π_h. The next lemma says that if m_h is an unmatched preread metastep on ℓ after (ι, N), that is, if $m_h \in N$ and m_h is contained in the preread set of some $m \notin N$, then m_h is contained in the preread set of $(m_k)^\iota$. Thus, the lemma basically says that we can find the write metastep to which an unmatched preread metastep is associated, by matching the registers of the write and preread metasteps.

Lemma 4.7.23 (Preread Lemma B) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $k, h \in [i]$, $m_k = \lambda(\iota, N, k)$, $\ell = reg(m_k)$, and $e_k = \delta(\alpha, \pi_k)$. Suppose the following hold.*

1. *$m_k \neq \emptyset$.*
2. *$type(e_k) = type(m_k) =$ `W`.*
3. *$m_h \in N$, $\pi_h \in procs((m_h)^\iota)$, $type(m_h) =$ `R`, and $reg(m_h) = \ell$.*
4. *There exists $m \notin N$ such that $m_h \in preads(m^\iota)$.*

Then we have $m = m_k$.

Proof. The main idea is the following. If $m_k \neq \check{m}_\iota$ (case 1 of the formal proof), then we can show using Lemma 4.7.12 that $m_k = \lambda(\iota^-, \check{N}, k)$. We then prove a series of claims to show that the assumptions of the inductive hypothesis for iteration ι^- are satisfied, for a particular instantiation of the parameters of the lemma, and then prove the lemma using the inductive hypothesis.

If $m_k = \check{m}_\iota$, then consider two cases. If ι is a write create iteration (case $2a$ of the formal proof), then m_k is the only write metastep on ℓ not in N, and so it follows that $m = m_k$. Otherwise, if ι is a modify iteration (case 2b of the formal proof), then if $k \neq i$, we can again prove the lemma using the inductive hypothesis. If $k = i$, then since ι is a modify iteration, there exist a $g < k$ such that π_g is the winner of $\check{m}_\iota$. We show that there exist a prefix $N_2 \supseteq N$ of $(M_\iota, \preceq_\iota)$, such that $m_h \in N_2$, $m \notin N_2$[33], and $\check{m}_\iota$, a write metastep on ℓ, is the next π_g metastep after (ι, N_2). Finally, we apply the inductive hypothesis, to conclude that $m = \check{m}_\iota = m_k$.

We now present the formal proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if it is true up to $\iota \ominus 1$, then it is true for ι. Let $\check{N} = N \cap M_{\iota^-}$, and let $\check{\alpha} \in \mathcal{L}(\iota^-, \check{N})$. First, note that by Lemma 4.7.6, there is exactly one $m \notin N$ such that $m_h \in \mathit{preads}(m^\iota)$.

Claim 4.7.24 *$m_h \in \check{N}$.*

Proof. Suppose $m_h \notin \check{N}$. Then we must have $m_h = \check{m}_\iota$. But from Lemma 4.6.3, we see that for any $\mu \in M_\iota$, we have $\check{m}_\iota \notin \mathit{preads}(\mu^\iota)$, contradicting assumption 5 of the lemma. Thus, we have $m_h \in \check{N}$. □

Now, consider two cases, either $m_k \neq \check{m}_\iota$, or $m_k = \check{m}_\iota$.

1. *Case $m_k \neq \check{m}_\iota$.*

   In this case, we prove the lemma by applying the inductive hypothesis for iteration ι^-. We prove a series of claims, in order to show that the assumptions of the lemma for iteration ι^- are satisfied.

   Claim 4.7.25 *$m_k = \lambda(\iota^-, \check{N}, k)$.*

   Proof. Since $m_k \neq \check{m}_\iota$, then either $k \neq i$, or $\check{m}_{\iota^-} \notin N$. Thus, the claim follows by Lemma 4.7.11, □

   Claim 4.7.26 *Let $e'_k = \delta(\check{\alpha}, \pi_k)$. Then $e'_k = e_k$.*

   Proof. Since $m_k \neq \check{m}_\iota$ by assumption, then by Lemma 4.6.9, we have $\Phi(\iota, N, k) = \Phi(\iota^-, \check{N}, k)$. Then, we have $\mathit{st}(\check{\alpha}, \pi_k) = \mathit{st}(\alpha, \pi_k)$ by Lemma 4.6.17.5, and so the lemma follows. □

   Claim 4.7.27 *ι is not a write create iteration.*

   Proof. Notice first that $m_k \not\preceq_{\iota^-} \check{m}_{\iota^-}$. Indeed, if $m_k \preceq_{\iota^-} \check{m}_{\iota^-}$, then since $\check{m}_{\iota^-} \in N$ and N is a prefix, we have $m_k \in N$, a contradiction. Also, by assumption 2 of the lemma, m_k is a write metastep on ℓ. Thus, we have $W_\iota \neq \emptyset$, and so ι is not a write create iteration. □

   Claim 4.7.28 *$m_h \in \mathit{preads}(m^{\iota^-})$.*

[33]Note that by Lemma 4.7.6, there is a unique $m \notin N$ such that $m_h \in \mathit{preads}(m^\iota)$.

Proof. We will show that $m \in M_{\iota^-}$. Indeed, if $m \notin M_{\iota^-}$, and from Lemma 4.6.3, we must have $m = \check{m}_\iota$, and ι is a write create iteration, contradicting Claim 4.7.27. Since $m \in M_{\iota^-}$, then from Lemma 4.6.3, we see that $preads(m^{\iota^-}) = preads(m^\iota)$. Thus, since $m_h \in preads(m^\iota)$, we have $m_h \in preads(m^{\iota^-})$. $\square$

Since $m \notin N$, then $m \notin \check{N} \subseteq N$. Now, from Claim 4.7.24, we have $m_h \in \check{N}$. From Claim 4.7.28, we have $m_h \in preads(m^{\iota^-})$, where $m \notin \check{N}$. By Claim 4.7.25, we have $m_k = \lambda(\iota^-, \check{N}, k)$, and m_k is a write metastep on ℓ. Lastly, by Claim 4.7.25, we have that $e'_k = e_k$ is a write step on ℓ. Thus, all the assumptions of the lemma hold, if we instantiate "ι" and "N" in the assumptions by ι^- and $\check{N}$, respectively. Then, by the inductive hypothesis, we conclude that $m = m_k$, and so the lemma holds for ι.

2. *Case* $m_k = \check{m}_\iota$.

Since $\check{m}_\iota$ is a write metastep, then $\iota \neq (i, 0)$. We consider two subcases, either ι is a create iteration, or ι is a modify iteration. Notice that since $type(e_k) = \mathtt{W}$, then ι is either a write create or write modify iteration.

(a) *ι is a write create iteration.*

Since ι is a write create iteration, then from $\langle 14 \rangle$ of ι, we see that $W_\iota = \emptyset$. From this, it follows that

$$\{\mu \mid (\mu \in M_{\iota^-} \backslash N) \wedge (type(\mu) = \mathtt{W}) \wedge (reg(\mu) = \ell)\} = \emptyset.$$

Thus, since $m_h \in preads(m^\iota)$, and $m \notin N$ is a write metastep on ℓ, we must have $m = \check{m}_\iota = m_k$, and so the lemma holds for ι.

(b) *ι is a write modify iteration.*

We have two cases, either $k \neq i$, or $k = i$. If $k \neq i$, then by Lemma 4.7.11, we have $m_k = \lambda(\iota^-, \check{N}, k)$. Since ι is a write modify iteration, we can argue as in the proof of 4.7.28 that $m_h \in preads(m^{\iota^-})$. Also, we can show $e'_k = e_k$, where e'_k is defined as in Claim 4.7.26. Thus, we can apply the inductive hypothesis to conclude that $m = m_k$, and so the lemma holds for ι.

If $k = i$, then we have the following.

Claim 4.7.29 *$\check{m}_\iota \preceq_\iota m$.*

Proof. From $\langle 15 \rangle$ of iteration ι, we have $\check{m}_\iota = \min_{\preceq_{\iota^-}} W_\iota$. Since $\iota \neq (i, 0)$ and $\check{m}_\iota = \lambda(\iota, N, k)$, then $\check{m}_{\iota^-} \in N$. Thus, since $m \notin N$, then we have $m \not\preceq_{\iota^-} \check{m}_{\iota^-}$, and so $m \in W_\iota$. Since m_k and m are both write metasteps on ℓ, then they are ordered by $\preceq_\iota$, by Lemma 4.6.17.6. Thus, we have $\check{m}_\iota \preceq_\iota m$. $\square$

Claim 4.7.30 $m \preceq_\iota \check{m}_\iota$.

Proof. Suppose for contradiction that $\check{m}_\iota \prec_\iota m$. Let $\pi_g = \diamond(winner(\check{m}_\iota))$, and let

$$N_1 = \{\mu \,|\, (\mu \in M_{\iota^-}) \wedge (\mu \prec_{\iota^-} \check{m}_\iota)\}, \qquad N_2 = N_1 \cup N.$$

N_1 is a prefix, and N_2 is a prefix because the union of two prefixes is a prefix. Let $\alpha_2 \in \mathcal{L}(\iota^-, N_2)$, and let $m_g = \lambda(\iota^-, N_2, g)$. Then we have the following.

- Since $\check{m}_\iota \notin N$, and since N_1 contains all metasteps in $\mu \in M_{\iota^-}$ such that $\mu \in \prec_{\iota^-} \check{m}_\iota$, then we have $m_g = \check{m}_\iota$, and so $type(m_g) = \mathtt{W}$.
- Let $e_g = \delta(\alpha_2, \pi_g)$. Then since $\pi_g = \diamond(winner(\check{m}_\iota))$, we have $type(e_g) = \mathtt{W}$.
- We have $m_h \in N_2$, since $m_h \in N$.
- We have $m \notin N_2$, since $m \notin N$ by assumption, and since $\check{m}_\iota \prec_\iota m$ and $\check{m}_\iota \notin N_1$.

Combining the above, we see that all the assumptions of the lemma hold, if we instantiate "ι", "N" and "m" in the assumptions by ι^-, N_2 and $m_g = \check{m}_\iota$, respectively. Then, by the inductive hypothesis, we have that $m_h \in preads((\check{m}_\iota)^{\iota^-})$. But this is a contradiction, because $\check{m}_\iota \prec_\iota m$, and $m_h \in preads(m^\iota)$. Thus, we conclude that $m \preceq_\iota \check{m}_\iota$. □

From Claims 4.7.29 and 4.7.30, we get that $m_k = \check{m}_\iota = m$, and so the lemma holds for ι.

The next lemma gives a characterization of the minimal metasteps after (ι, N). Namely, a metastep m is minimal exactly when the preread set of m is contained in N, and for every process π_k contained in m, the next π_k metastep after (ι, N) is m. This is not the most convenient characterization for decoding purposes, since the decoder does not have direct knowledge of the preread set of m, nor the processes contained in m. In subsequent lemmas (Lemmas 4.7.35 and 4.7.36), we provide other characterizations of the minimal metasteps after a prefix, that are more convenient for the decoder.

Lemma 4.7.31 (λ Lemma C) *Let $\iota = (i, j)$ be any iteration, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Let $m \in M_\iota \backslash N$, and suppose $type(m) = \mathtt{W}$. Then $m \in \lambda(\iota, N)$ if and only if we have the following.*

1. *$preads(m^\iota) \subseteq N$.*
2. *For all $\pi_k \in procs(m^\iota)$, we have $\lambda(\iota, N, k) = m$.*

Proof. The main idea is the following. Consider three cases, $\check{m}_\iota \not\preceq_\iota m$, $m = \check{m}_\iota$, or $\check{m}_\iota \prec_\iota m$. In the first case, we have $m \in \lambda(\iota, N)$ precisely when $m \in \lambda(\iota^-, \check{N})$; thus, the lemma can be shown by applying the inductive hypothesis. If $m = \check{m}_\iota$, then $m \in \lambda(\iota, N)$ precisely when $m \in \lambda(\iota^-, \check{N})$, and $\lambda(\iota, N, i) = m$; then we again apply the inductive hypothesis, noting that $procs(m^\iota) = procs(m^{\iota^-}) \cup \{\pi_i\}$. Finally, if $\check{m}_\iota \prec_\iota m$, then ι is a modify iteration, and so $\check{m}_\iota \prec_{\iota^-} m$.

From this, it once more follows that $m \in \lambda(\iota, N)$ precisely when $m \in \lambda(\iota^-, \check{N})$, and the lemma follows from the inductive hypothesis.

We now present the formal proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if the lemma is true for $\iota \ominus 1$, then it is also true for ι. Let $\check{N} = N \cap M_{\iota^-}$. Consider three cases, either $\check{m}_\iota \not\preceq_\iota m$, $m = \check{m}_\iota$, or $\check{m}_\iota \prec_\iota m$. Recall from Definition 4.6.16 that $\Upsilon(\iota, m)$ is the set of metasteps in M_ι that $\preceq_\iota m$.

1. *Case* $\check{m}_\iota \not\preceq_\iota m$.

Claim 4.7.32 $(m \in \lambda(\iota, N)) \Leftrightarrow (m \in \lambda(\iota^-, \check{N}))$.

Proof. Since $\check{m}_\iota \not\preceq_\iota m$, then it follows from Lemma 4.6.3 that $\Upsilon(\iota, m) = \Upsilon(\iota^-, m)$. We can see that $m \in \lambda(\iota, N) \Leftrightarrow (\Upsilon(\iota, m) \subseteq N) \wedge (m \notin N)$. We claim that

$$(\Upsilon(\iota, m) \subseteq N) \wedge (m \notin N) \Leftrightarrow (\Upsilon(\iota^-, m) \subseteq \check{N}) \wedge (m \notin \check{N}).$$

First, note that $(m \notin N) \Leftrightarrow (m \notin \check{N})$, because $m \neq \check{m}_\iota$, and N and $\check{N}$ are either equal, or differ by $\check{m}_\iota$.

Next, we show that $(\Upsilon(\iota, m) \subseteq N) \Leftrightarrow (\Upsilon(\iota^-, m) \subseteq \check{N})$. Indeed, since $\Upsilon(\iota, m) = \Upsilon(\iota^-, m)$, and $\check{N} \subseteq N$, then $(\Upsilon(\iota^-, m) \subseteq \check{N}) \Rightarrow (\Upsilon(\iota, m) \subseteq N)$. For the other direction, notice that $\check{m}_\iota \notin \Upsilon(\iota, m)$, since if $\check{m}_\iota \in \Upsilon(\iota, m)$, then we must have $\check{m}_\iota \preceq_\iota m$, a contradiction. Thus, since N and $\check{N}$ differ at most by $\check{m}_\iota$, we have $(\Upsilon(\iota, m) \subseteq N) \Rightarrow (\Upsilon(\iota^-, m) \subseteq \check{N})$.

From the above, we have

$$\begin{aligned}(m \in \lambda(\iota, N)) &\Leftrightarrow (\Upsilon(\iota, m) \subseteq N) \wedge (m \notin N) \\ &\Leftrightarrow (\Upsilon(\iota^-, m) \subseteq \check{N}) \wedge (m \notin \check{N}) \\ &\Leftrightarrow (m \in \lambda(\iota^-, \check{N})).\end{aligned}$$

□

Claim 4.7.33

$$(preads(m^{\iota^-}) \subseteq \check{N}) \wedge (\forall \pi_k \in procs(m^{\iota^-}) : \lambda(\iota, \check{N}, k) = m) \quad \Leftrightarrow$$
$$(preads(m^{\iota}) \subseteq N) \wedge (\forall \pi_k \in procs(m^{\iota}) : \lambda(\iota, N, k) = m).$$

Proof. Since $m \not\preceq_\iota \check{m}_\iota$, then by Lemma 4.6.3, we have $m^\iota = m^{\iota^-}$. Now, since $\check{m}_\iota \notin preads(m^{\iota^-})$, we have $(preads(m^{\iota^-}) \subseteq \check{N}) \Leftrightarrow (preads(m^\iota) \subseteq N)$.

Next, we show that

$$(\forall \pi_k \in procs(m^{\iota^-}) : \lambda(\iota, \check{N}, k) = m) \Leftrightarrow (\forall \pi_k \in procs(m^{\iota}) : \lambda(\iota, N, k) = m).$$

We first claim that either we have $\check{m}_{\iota^-} \notin \check{N}$, or $\forall \pi_k \in procs(m^{\iota^-}) : k \neq i$. Indeed, suppose that we have $\check{m}_{\iota^-} \in \check{N}$, and there exists $k \in procs(m^{\iota^-})$ such that $k = i$. Then this means $\lambda(\iota^-, \check{N}, i) = \check{m}_{\iota} = m$, which contradicts the assumption that $\check{m}_{\iota} \not\preceq_{\iota} m$. Now, since we have $\check{m}_{\iota^-} \notin \check{N}$ or $\forall \pi_k \in procs(m^{\iota^-}) : k \neq i$, then by Lemma 4.7.12, we have $\forall \pi_k \in procs(m^{\iota^-})$: $\lambda(\iota^-, \check{N}, k) = \lambda(\iota, N, k)$. Finally, since $procs(m^{\iota}) = procs(m^{\iota^-})$, we have

$$(\forall \pi_k \in procs(m^{\iota^-}) : \lambda(\iota, \check{N}, k) = m) \Leftrightarrow (\forall \pi_k \in procs(m^{\iota}) : \lambda(\iota, N, k) = m).$$

□

Combining the above, we get the following.

$$\begin{aligned} m \in \lambda(\iota, N) \quad &\Leftrightarrow \quad (m \in \lambda(\iota^-, \check{N})) \\ &\Leftrightarrow \quad (preads(m^{\iota^-}) \subseteq \check{N}) \wedge (\forall \pi_k \in procs(m^{\iota^-}) : \lambda(\iota, \check{N}, k) = m) \\ &\Leftrightarrow \quad (preads(m^{\iota}) \subseteq N) \wedge (\forall \pi_k \in procs(m^{\iota}) : \lambda(\iota, N, k) = m). \end{aligned}$$

Here, the first equivalence follows by Claim 4.7.32. The second equivalence follows by the inductive hypothesis. The final equivalence follows by Claim 4.7.33.

2. *Case* $m = \check{m}_{\iota}$.

Let $N_1 = \{\mu \,|\, (\mu \in M_{\iota}) \wedge (\mu \preceq_{\iota} \check{m}_{\iota^-})\}$. By Lemma 4.6.3, we can see that

$$\Upsilon(\iota, m) = \Upsilon(\iota^-, m) \cup N_1.$$

Thus, we have the following.

$$\begin{aligned} m \in \lambda(\iota, N) \quad &\Leftrightarrow \quad (\Upsilon(\iota, m) \subseteq N) \wedge (m \notin N) \\ &\Leftrightarrow \quad (\Upsilon(\iota^-, m) \subseteq \check{N}) \wedge (\check{m}_{\iota^-} \in N) \wedge (\check{m}_{\iota} \notin N) \\ &\Leftrightarrow \quad (m \in \lambda(\iota^-, \check{N})) \wedge (\lambda(\iota, N, i) = \check{m}_{\iota}) \\ &\Leftrightarrow \quad (preads(m^{\iota^-}) \subseteq \check{N}) \wedge (\forall \pi_k \in procs(m^{\iota^-}) : \lambda(\iota, \check{N}, k) = m) \wedge ((\lambda(\iota, N, i) = \check{m}_{\iota}) \\ &\Leftrightarrow \quad (preads(m^{\iota}) \subseteq N) \wedge (\forall \pi_k \in procs(m^{\iota}) : \lambda(\iota, N, k) = m). \end{aligned}$$

The next to last equivalence follows by the inductive hypothesis, and the last equivalence follows from the fact that $procs((\check{m}_\iota)^\iota) = procs((\check{m}_\iota)^{\iota^-}) \cup \{\pi_i\}$, and by using similar arguments as in the proof of Claim 4.7.33.

3. *Case* $\check{m}_\iota \prec_\iota m$.

   Let $N_2 = \{\mu \,|\, (\mu \in M_\iota) \wedge (\mu \preceq_\iota \check{m}_\iota)\}$. Using Lemma 4.6.3, we get that $\Upsilon(\iota, m) = \Upsilon(\iota^-, m) \cup N_2$. Since $\check{m}_\iota \prec_\iota m$, then we can see from Lemma 4.6.3 that ι is a modify iteration. Thus, since $\check{m}_\iota \prec_\iota m$, we also have $\check{m}_\iota \prec_{\iota^-} m$, and so $(m \in \lambda(\iota, N)) \Leftrightarrow (m \in \lambda(\iota^-, \check{N}))$. Also, we have $M_\iota = M_{\iota^-}$, $N = \check{N}$, and $m^\iota = m^{\iota^-}$. Thus, using the inductive hypothesis, we have the following.

$$\begin{aligned}
m \in \lambda(\iota, N) \quad &\Leftrightarrow \quad m \in \lambda(\iota^-, \check{N}) \\
&\Leftrightarrow \quad (preads(m^{\iota^-}) \subseteq \check{N}) \wedge (\forall \pi_k \in procs(m^{\iota^-}) : \lambda(\iota, \check{N}, k) = m) \\
&\Leftrightarrow \quad (preads(m^\iota) \subseteq N) \wedge (\forall \pi_k \in procs(m^\iota) : \lambda(\iota, N, k) = m).
\end{aligned}$$

□

Let e and m be the next π_k step and metastep after (ι, N), respectively. Suppose that e is a write step, and m is a write metastep writing a value v to a register ℓ. Then the next lemma states that the unmatched preread metasteps on ℓ after (ι, N) are a subset of $preads(m^\iota)$. Also, the processes that v-read ℓ after (ι, N) are a subset of $readers(m^\iota)$, and the processes that write to ℓ after (ι, N) are a subset of $writers(m^\iota) \cup winner(m^\iota)$. In addition, for each process in $readers(\iota, N, \ell, v) \cup wwriters(\iota, N, \ell)$, the next metastep for the process after (ι, N) is m. This lemma is used in the proof of Lemma 4.7.35.

Lemma 4.7.34 (λ Lemma D) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $k \in [i]$, $m = \lambda(\iota, N, k)$, and $e = \delta(\alpha, \pi_k)$. Suppose that $type(e) = type(m) = \mathtt{W}$, and let $\ell = reg(m)$ and $v = val(m)$. Then we have the following.*

1. *$preads(\iota, N, \ell) \subseteq preads(m^\iota)$.*

2. *$readers(\iota, N, \ell, v) \subseteq readers(m^\iota)$, and $\forall \pi_h \in readers(\iota, N, \ell, v) : \lambda(\iota, N, h) = m$.*

3. *$wwriters(\iota, N, \ell) \subseteq writers(m^\iota) \cup winner(m^\iota)$, and $\forall \pi_h \in wwriters(\iota, N, \ell) : \lambda(\iota, N, h) = m$.*

Proof. The proof mainly involves unraveling definitions, then applying Lemmas 4.7.14 and 4.7.23. We show each part of the lemma separately. For the first part, let $m_1 \in preads(\iota, N, \ell)$. Then by the definition of $preads(\iota, N, \ell)$, m_1 is a read metastep on ℓ, $m_1 \in N$, and $\exists m_2 \notin N : m_1 \in$

$preads((m_2)^\iota)$. Then by Lemma 4.7.23, we have $m_2 = m$. Since m_1 was arbitrary, we have $preads(\iota, N, \ell) \subseteq preads(m^\iota)$.

In the rest of the lemma, for any $h \in [i]$, let $m_h = \lambda(\iota, N, h)$, $s_h = st(\alpha, \pi_h)$, $e_h = \delta(\alpha, \pi_h)$, and $S_{h,\ell,v} = \{s \mid (s \in S) \wedge (st(s, \pi_h) = st(s_k, \pi_h)) \wedge (st(s, \ell) = v)\}$.

For the second part of the lemma, let $\pi_h \in readers(\iota, N, \ell, v)$. Then by the definition of $readers(\iota, N, \ell, v)$, m_h is a write metastep on ℓ, e_h is a read step on ℓ, and $\exists s \in S_{h,\ell,v} : \Delta(s, e_h, \pi_h) \neq s_h$. Then by Lemma 4.7.14, we have $m_h = m$, and $\pi_h \in readers(m^\iota)$. Since h was arbitrary, we have $\forall \pi_h \in readers(\iota, N, \ell, v) : \lambda(\iota, N, h) = m$, and $readers(\iota, N, \ell, v) \subseteq readers(m^\iota)$.

By the definition of $wwriters(\iota, N, \ell)$, m_h is a write metastep on ℓ, and e_h is a write step on ℓ. Then, by Lemma 4.7.14, we have $m_h = m$, and $\pi_h \in writers(m^\iota) \cup winner(m^\iota)$. Since h was arbitrary, we have $\forall \pi_h \in wwriters(\iota, N, \ell) : \lambda(\iota, N, h) = m$, and $wwriters(\iota, N, \ell) \subseteq writers(m^\iota) \cup winner(m^\iota)$. $\square$

The next lemma gives a characterization of the minimal metasteps after (ι, N) that is convenient for the decoder. Let e and m be the next p_{π_k} step and metastep after (ι, N), respectively. Suppose that e is a write step, and m is a write metastep writing value v to ℓ. Then the lemma states that m is a minimal metastep after (ι, N) if and only if $|preads(\iota, N, \ell)| = |preads(m^\iota)|$, $|readers(\iota, N, \ell, v)| = |readers(m^\iota)|$, $|wwriters(\iota, N, \ell)| = |writers(m^\iota) \cup winner(m^\iota)|$. To make use of this characterization, the decoder only needs to be able to compute the sets $preads(\iota, N, \ell), readers(\iota, N, \ell, v)$, and $wwriters(\iota, N, \ell)$, and to know the cardinalities of the sets $preads(m^\iota)$, $readers(m^\iota)$, and $writers(m^\iota)$. The cardinalities are stored by the encoder, and can be retrieved from the encoding by the decoder at the appropriate time. The sets $readers(\iota, N, \ell, v)$ and $wwriters(\iota, N, \ell)$ can be computed by the decoder simply by knowing N. $preads(\iota, N, \ell)$ can be computed by knowing N, and additionally, for each read metastep in N, a flag indicating whether the metastep is a preread. These flags are also stored in the encoding.

Lemma 4.7.35 (λ Lemma E) *Let $\iota = (i, j)$ be any iteration, let N be a prefix of $(M_\iota, \preceq_\iota)$, and let $\alpha \in \mathcal{L}(\iota, N)$. Let $k \in [i]$, $m = \lambda(\iota, N, k)$, and $e = \delta(\alpha, \pi_k)$. Let $\ell = reg(m)$ and $v = val(m)$. Suppose that $type(e) = type(m) = \mathtt{W}$. Then $m \in \lambda(\iota, N)$ if and only if we have the following.*

1. *$|preads(\iota, N, \ell)| = |preads(m^\iota)|$.*
2. *$|readers(\iota, N, \ell, v)| = |readers(m^\iota)|$.*
3. *$|wwriters(\iota, N, \ell)| = |writers(m^\iota) \cup winner(m^\iota)|$.*

Proof.

1. $(\Rightarrow)$ *direction.*

Since $m \in \lambda(\iota, N)$, then by Lemma 4.7.31, we have $preads(m^\iota) \subseteq N$, and $\forall \pi_k \in procs(m^\iota)$: $\lambda(\iota, N, k) = m$. We show each part of the lemma separately.

To show $|preads(\iota, N, \ell)| = |preads(m^\iota)|$, let $m_1 \in preads(m^\iota)$. Then we have the following: $m_1 \in N$, m_1 is a read metastep on ℓ, $m \notin N$, and $m_1 \in preads(m^\iota)$. Thus, we have $m_1 \in preads(\iota, N, \ell)$, by the definition of $preads(\iota, N, \ell)$. Since m_1 was arbitrary, we have $preads(m^\iota) \subseteq preads(\iota, N, \ell)$. Since we also have $preads(\iota, N, \ell) \subseteq preads(m^\iota)$ by Lemma 4.7.34, then $|preads(\iota, N, \ell)| = |preads(m^\iota)|$.

In the rest of the lemma, for any $h \in [i]$, let $m_h = \lambda(\iota, N, h)$, $s_h = st(\alpha, \pi_h)$, $e_h = \delta(\alpha, \pi_h)$, and $S_{h,\ell,v} = \{s \,|\, (s \in S) \wedge (st(s, \pi_h) = st(s_k, \pi_h)) \wedge (st(s, \ell) = v)\}$.

To show that $|readers(\iota, N, \ell, v)| = |readers(m^\iota)|$, let $\pi_h \in readers(m^\iota)$. Since $m \in \lambda(\iota, N)$, then by Lemma 4.7.31, we have $m_h = m$. Let ϵ_h denote the step that π_h takes in m. Since $\pi_h \in readers(m^\iota)$, then ϵ_h is a read step on ℓ. By Lemma 4.7.12, we have $e_h = \epsilon_h$, so e_h is also a read step on ℓ. Then, by Lemma 4.7.13, there exists $s \in S_{h,\ell,v}$ such that $\Delta(s, e_h, \pi_h) \neq s_h$. Thus, by the definition of $readers(\iota, N, \ell, v)$, we have $\pi_h \in readers(\iota, N, \ell, v)$. Since π_h was arbitrary, we have $readers(m^\iota) \subseteq readers(\iota, N, \ell, v)$. Since we also have $readers(\iota, N, \ell, v) \subseteq readers(m^\iota)$ by Lemma 4.7.34, then $|readers(\iota, N, \ell, v)| = |readers(m^\iota)|$.

To show that $|wwriters(\iota, N, \ell)| = |writers(m^\iota) \cup winner(m^\iota)|$, let $\pi_h \in writers(m^\iota) \cup winner(m^\iota)$. Since $m \in \lambda(\iota, N)$, then by Lemma 4.7.31, we have $m_h = m$. Let ϵ_h denote the step that π_h takes in m. Since $\pi_h \in writers(m^\iota)$, then ϵ_h is a write step on ℓ. By Lemma 4.7.12, we have $e_h = \epsilon_h$, so e_h is also a write step on ℓ. Thus, by the definition of $wwriters(\iota, N, \ell)$, we have $\pi_h \in wwriters(\iota, N, \ell)$, and so $writers(m^\iota) \cup winner(m^\iota) \subseteq wwriters(\iota, N, \ell)$. Since we also have $wwriters(\iota, N, \ell) \subseteq writers(m^\iota) \cup winner(m^\iota)$ by Lemma 4.7.34, then $|wwriters(\iota, N, \ell)| = |writers(m^\iota) \cup winner(m^\iota)|$.

2. ($\Leftarrow$) *direction.*

By Lemma 4.7.34, we have $preads(\iota, N, \ell) \subseteq preads(m^\iota), readers(\iota, N, \ell, v) \subseteq readers(m^\iota)$, and $wwriters(\iota, N, \ell) \subseteq writers(m^\iota) \cup winner(m^\iota)$. Thus, since $|preads(\iota, N, \ell)| = |preads(m^\iota)|$, $|readers(\iota, N, \ell, v)| = |readers(m^\iota)|$, and $|wwriters(\iota, N, \ell)| = |writers(m^\iota) \cup winner(m^\iota)|$, we have $preads(\iota, N, \ell) = preads(m^\iota), readers(\iota, N, \ell, v) = readers(m^\iota)$, and $wwriters(\iota, N, \ell) = writers(m^\iota) \cup winner(m^\iota)$.

Let $m_1 \in preads(\iota, N, \ell)$. Then by definition, we have $m_1 \in N$. Thus, since $preads(\iota, N, \ell) = preads(m^\iota)$, we have

$$preads(m^\iota) \subseteq N. \tag{4.11}$$

Next, let $\pi_h \in readers(\iota, N, \ell, v)$. Then by Lemma 4.7.34, we have $\lambda(\iota, N, h) = m$. Since

$readers(\iota, N, \ell, v) = readers(m^\iota)$, we have

$$\forall \pi_h \in readers(m^\iota) : \lambda(\iota, N, h) = m. \tag{4.12}$$

Next, let $\pi_g \in wwriters(\iota, N, \ell)$. Then by Lemma 4.7.34, we have $\lambda(\iota, N, g) = m$. Since $wwriters(\iota, N, \ell) = writers(m^\iota) \cup winner(m^\iota)$, then

$$\forall \pi_g \in writers(m^\iota) \cup winner(m^\iota) : \lambda(\iota, N, g) = m. \tag{4.13}$$

Finally, by combining Equations 4.11, 4.12 and 4.13, and applying Lemma 4.7.31, we have that $m \in \lambda(\iota, N)$.

□

Lemma 4.7.35 gave a convenient characterization of the minimal *write* metasteps after (ι, N). The next lemma characterizes the minimal read and critical metasteps after (ι, N). The minimality condition is simple: a read or critical metastep is minimal after (ι, N) exactly when it is the next metastep after (ι, N) for some process.

Lemma 4.7.36 (λ Lemma F) *Let $\iota = (i, j)$ be any iteration, let $m \in M_\iota$, and let N be a prefix of $(M_\iota, \preceq_\iota)$. Suppose that type$(m) \in \{\mathtt{R}, \mathtt{C}\}$. Then $m \in \lambda(\iota, N)$ if and only if there exists $k \in [i]$ such that $m = \lambda(\iota, N, k)$.*

Proof. We use induction on ι. The lemma is true for $\iota = (1, 0)$. We show that if the lemma is true for $\iota \ominus 1$, then it is also true for ι. Let $\check{N} = N \cap M_{\iota^-}$.

1. $(\Rightarrow)$ *direction.*

   Consider two cases, either $m \neq \check{m}_\iota$, or $m = \check{m}_\iota$.

   (a) *Case $m \neq \check{m}_\iota$.*

   We claim that $(m \in \lambda(\iota, N)) \Rightarrow (m \in \lambda(\iota^-, \check{N}))$. Indeed, suppose that $m \in \lambda(\iota, N)$, and let $m_1 \in M_{\iota^-}$ be such that $m_1 \preceq_{\iota^-} m$. We want to show that $m_1 \in \check{N}$. We have $m_1 \preceq_\iota m$ by Lemma 4.6.4, and so $m_1 \in N$, since $m \in \lambda(\iota, N)$. Then, since $m_1 \neq \check{m}_\iota$, we also have $m_1 \in N \cap M_{\iota^-} = \check{N}$, and so the claim holds. Now, since $m \in \lambda(\iota^-, \check{N})$, then by the inductive hypothesis, there exists $k \in [i]$ such that $m = \lambda(\iota^-, \check{N}, k)$. Since $m \neq \check{m}_\iota$, then we have $k \neq i$ or $\check{m}_{\iota^-} \notin N$. Thus, by Lemma 4.7.11, we have $m = \lambda(\iota^-, \check{N}, k) = \lambda(\iota, N, k)$, and so the lemma holds.

   (b) *Case $m = \check{m}_\iota$.*

   Since $m = \check{m}_\iota$, then we see from Lemma 4.6.3 that $m = \lambda(\iota, N, i)$.

2. ($\Leftarrow$) *direction.*

   Consider two cases, either $m \neq \check{m}_\iota$, or $m = \check{m}_\iota$.

   (a) *Case $m \neq \check{m}_\iota$.*

   Since $m \neq \check{m}_\iota$, then we have $k \neq i$ or $\check{m}_{\iota^-} \notin N$, and so $m = \lambda(\iota, N, k) = \lambda(\iota^-, \check{N}, k)$, by Lemma 4.7.11. Then by the inductive hypothesis, we have $m \in \lambda(\iota^-, \check{N})$, and so $\Upsilon(\iota^-, m) \subseteq \check{N}$. Consider two cases, either $\check{m}_\iota \not\prec_\iota m$, or $\check{m}_\iota \prec_\iota m$.

   If $\check{m}_\iota \not\prec_\iota m$, then we have $\Upsilon(\iota, m) = \Upsilon(\iota^-, m)$. Thus, we have $\Upsilon(\iota, m) \subseteq N$, and so $m \in \lambda(\iota, N)$.

   If $\check{m}_\iota \prec_\iota m$, then let $N_1 = \{\mu \,|\, (\mu \in M_\iota) \wedge (\mu \preceq_\iota \check{m}_\iota)\}$. We have $\Upsilon(\iota, m) = \Upsilon(\iota^-, m) \cup N_1$, and ι is a modify iteration. Since m is a read or critical metastep and $m = \lambda(\iota, N, k)$, we have $\check{m}_\iota \in N$, and so $N_1 \subseteq N = \check{N}$. Thus, since $\Upsilon(\iota^-, m) \subseteq \check{N}$, we have $\Upsilon(\iota, m) \subseteq N$, and so $m \in \lambda(\iota, N)$.

   (b) *Case $m = \check{m}_\iota$.*

   Since $\check{m}_\iota = \lambda(\iota, N, k)$ is a read or critical metastep, then we have $k = i$, $\check{m}_{\iota^-} \in N$, and $\check{m}_\iota \notin N$. Thus, we see that $\check{m}_\iota \in \lambda(\iota, N)$.

□

Summary of Properties for Decoding

In the remainder of this section, we describe how the lemmas in Section 4.7.3 motivate the DECODE algorithm presented in Section 4.10. For any metastep m, we always refer to the iteration ι^n version of m. Thus, we omit the ι^n superscript from our notation. For example, we write *steps*(m) to mean *steps*(m^{ι^n}).

Our goal for decoding is to output a linearization of $(M_n, \preceq_n)$[34]. To do this, DECODE maintains an invariant that at any point in its execution, it has output a linearization α of $(N, \preceq_n)$, where N is some prefix of $(M_n, \preceq_n)$. To satisfy the invariant, DECODE ensures that whenever it appends a set of steps to α, those steps are precisely the steps in some minimal metastep not contained in N. That is, if DECODE appends a set of steps β to α, then $\beta =$ *steps*(m), for some $m \in \lambda(N)$[35]. Thus, the main task of the decoder is to identify the minimal metasteps after N.

If m is a read or critical metastep, then it is easy for DECODE to know when $m \in \lambda(N)$. Indeed, Lemma 4.7.36 shows that $m \in N$ precisely when m is the next metastep after N for some process. Next, suppose that m is a write metastep, and let $\ell =$ *reg*(m) and $v =$ *val*(m). To determine when $m \in \lambda(N)$, DECODE uses the property from Lemma 4.7.35, namely, that $m \in \lambda(N)$ if and

[34] Following the notational convention in this section, we write $(M_n, \preceq_n)$ to mean $((M_n)^{\iota^n}, \preceq_n)$.

[35] DECODE also ensures that it appends all the non-winning write steps in β to α first, then appends the winning write, then appends the read steps in β. This condition is easy to satisfy, and will not be discussed further.

only if $|preads(N, \ell)| = |preads(m)|$, $|readers(N, \ell, v)| = |readers(m)|$, and $|wwriters(N, \ell)| = |writers(m) \cup winner(m)|$. To perform these checks, Decode first needs to know the values of $|preads(m)|$, $|readers(m)|$ and $|writers(m) \cup winner(m)|$. These values are the components of the signature for m, and are stored in the string E_π output by Encode operating on input $(M_n, \preceq_n)$. Thus, Decode gets these values by reading E_π. Next, Decode must be able to compute the sets $preads(N, \ell)$, $readers(N, \ell, v)$ and $wwriters(N, \ell)$, based on the current linearization α of $(N, \preceq_n)$ that is has produced. To compute $preads(N, \ell)$, Decode uses Lemma 4.7.23, which shows that if a read metastep on ℓ in N is contained in the preread set of *any* write metastep not in N, then it is contained in the preread set of m. To compute $readers(N, \ell, v)$ and $wwriters(N, \ell)$, Decode keeps track of the processes whose next step after N v-reads ℓ, or writes to ℓ. The algorithm presented in Section 4.10 is an implementation of these ideas.

4.8 The Encoding Step

In this section, we present an algorithm that encodes $((M_n)^{\iota^n}, \preceq_n)$ as a string of length $O(C(\alpha))$, where α is any linearization of $((M_n)^{\iota^n}, \preceq_n)$[36]. In the remainder of this chapter, we will consider only the iteration ι^n version of any metastep. Thus, we write m to mean m^{ι^n}, for any $m \in M_n$. We first define the following.

Definition 4.8.1 (Extended Type) *Define*

$$\mathcal{T} = \{\mathtt{C}, \mathtt{R}, \mathtt{W}, \mathtt{PR}, \mathtt{SR}, \mathtt{\$}\} \cup \bigcup_{pr, r, w \in [n]} \{\mathtt{PR}pr\mathtt{R}r\mathtt{W}w\}.$$

We say that $\mathcal{T}$ is the set of extended types*. Let $m \in M_n$ be a metastep, let $e \in steps(m)$ be a step in m, and let $i = proc(e)$ be the process taking step e. Then we define the following.*

1. *If $type(m) = \mathtt{W}$, then we define the following.*
   (a) *If $type(e) = \mathtt{R}$, then $xtype(e, m) = \mathtt{R}$.*
   (b) *If $type(e) = \mathtt{W}$ and $i \neq winner(m)$, then $xtype(e, m) = \mathtt{W}$.*
   (c) *If $type(e) = \mathtt{W}$ and $i = winner(m)$, then $xtype(e, m) = \mathtt{PR}pr\mathtt{R}r\mathtt{W}w$, where $pr = |preads(m)|$, $r = |reads(m)|$ and $w = |writes(m)| + 1$.*
2. *If $type(m) = \mathtt{R}$, then e is a read step. We define the following.*
   (a) *If $\exists \mu \in M$ such that $m \in preads(\mu)$, then $xtype(e, m) = \mathtt{PR}$.*
   (b) *Otherwise, $xtype(e, m) = \mathtt{SR}$.*

[36] By Lemma 4.6.17.1, we have $\alpha \in runs(\mathcal{A})$. Also, we show in Lemma 4.7.10 that all linearizations of $((M_n)^{\iota^n}, \preceq_n)$ have the same cost in the state change cost model.

3. If type(m) = `C`*, then e is a critical step. We define xtype(e,m) =* `C`*.*

We say xtype(e,m) is the extended type *of e in m.*

Please see Figure 4-5 for the pseudocode of the encoding algorithm. All text in the pseudocode in `typewriter` font represent string literals. For example, `W` is the string "W".

The input to Encode is a pair $(M, \preceq)$, where M is a set of metasteps, and $\preceq$ is a partial order on M. The encoding uses a two dimensional grid of cells, with n columns and an infinite number of rows. The encoder fills some of the cells with strings. The contents of the cell in column i and row j is denoted by $T(i,j)$.

The encoder iterates over all the metasteps in M, in an arbitrary order. For each $m \in M$, it iterates over all the steps in *steps*(m), again in an arbitrary order. Let $e \in$ *steps*(m), and suppose e is performed by process p. Then Encode calls $\mathrm{PC}(p, m, M, \preceq)$, which returns a number q such that m is p's q'th largest metastep in M[37]. Note that q is well defined, since the set of metasteps containing p in M is totally ordered by $\preceq$, by Lemma 4.6.8. The encoder fills cell $T(p,q)$ with (a string representation of) *xtype*(e,m), the extended type of e in m. Note that *xtype*(e,m) contains information about the types of both e and m. For example, if e is a read step, then *xtype*(e,m) can be `R`, `SR` or `PR`; *xtype*(e,m) = `R` indicates that e is a read step in a write metastep, while *xtype*$(e,m) \in \{$`SR`, `PR`$\}$ indicates that e is a read step in a read metastep; also, *xtype*(e,m) = `PR` indicates that the read metastep containing e is a preread metastep. As another example, if e is the winning step in a metastep, then *xtype*(e,m) contains the signature for that metastep, *i.e.*, a count of the number of reads, writes and prereads in the metastep.

The complete encoding E_π is produced by concatenating all nonempty cells $T(1,\cdot)$ (in order), then appending all nonempty cells $T(2,\cdot)$, etc., and finally appending all nonempty cells $T(n,\cdot)$. The encoder uses the helper function *nrows*(T,i), which returns how many nonempty cells there are in column i of T.

4.9 Correctness Properties of the Encoding

In this section, we show that the length of the string E_π output by Encode is proportional to the cost of a linearization of $(M_n, \preceq_n)$. Recall from Definition 4.7.1 that G is the total number of steps contained in all the metasteps in M_n after iteration ι^n.

Theorem 4.9.1 (Encoding Theorem A) *Let α be the output of* $\mathrm{Lin}(M_n, \preceq_n)$*. Then we have $|E_\pi| = O(C(\alpha))$.*

Proof. The main idea for the proof is the following. Given a metastep $m \in M_n$, there are two parts to the cost of encoding m. The first part is the cost to encode the steps of m, and possibly the

[37] "PC" stands for *program counter.*

```
1: procedure ENCODE($M, \preceq$)
2:    for all $m \in M$ do
3:       for all $e \in steps(m)$
4:          $p \leftarrow proc(e)$;    $q \leftarrow$ PC($p, m, M, \preceq$)
5:          $T(p, q) \leftarrow xtype(e, m)$
6:    end for end for

7:    for $i \leftarrow 1, n$ do
8:       for $j \leftarrow 1, nrows(T, i)$ do
9:          $E_\pi \leftarrow E_\pi \circ \texttt{\#} \circ T(i, j)$
10:      end for
11:      $E_\pi \leftarrow E_\pi \circ \texttt{\$}$
12:   end for
13:   return $E_\pi$
14: end procedure

15: procedure PC($p, m, M, \preceq$)
16:   $N \leftarrow \{\mu \,|\, (\mu \in M) \wedge (p \in procs(\mu))\}$
17:   sort $N$ in increasing order of $\preceq$ as $n_1, \ldots, n_{|N|}$
18:   return $q \in 1, \ldots, |N|$ such that $n_q = m$
19: end procedure
```

Figure 4-5: Encoding M and $\preceq$ as a string E_π.

signature of m, if m is a write metastep. The other cost to encoding m is for encoding the preread set of m, if m is a write metastep. If m has t steps, then we show that the cost of the first part of encoding m is $O(t)$. For the second part, we do not compute the cost directly, but rather, charge the cost to the encoding costs of all the read metasteps in *pread*(m). From this, it follows that, summed over all $m \in M_n$, the encoding cost of both parts is bounded by $O(G)$, which is $O(|\alpha|)$ by Lemma 4.7.10.

We now present the formal proof. Let $c \geq 1$ be the smallest constant such that any symbol in E_π, such as SR or #, can be encoded using at most c bits, and any natural number d in E_π can be encoded using at most $c \log d$ bits. Clearly, c is finite. Recall that ENCODE($M_n, \preceq_n$) works by iterating over the metasteps in M_n, and encoding information about each metastep m in several cells of T. Each cell is associated either with a step contained in m, or a read metastep contained in *pread*(m). For any $m \in M_n$, define the following.

1. Let $s(m)$ be the number of bits used in E_π to encode m. More precisely, $s(m)$ is the sum of the number of bits used in all the cells of T associated with m.

2. Let $t(m) = |steps(m)|$ be number of steps in m.

3. Let $r(m) = |reads(m)|$ be number of read steps in m.

4. Let $w(m) = |writes(m) \cup win(m)|$ be number of write and winning steps in m.

5. Let $p(m) = |preads(m)|$ be number of preread metasteps of m.

We have

$$|E_\pi| \leq \sum_{m \in M_n} s(m) + c \sum_{m \in M_n} t(m) + O(n). \tag{4.14}$$

Here, the $c\sum_{m \in M_n} t(m)$ and $O(n)$ terms account for the delimiters, such as #, used in E_π when concatenating the cells of T. We have $c\sum_{m \in M_n} t(m) = cG$. Also, we have $n \leq G$, since each process

p_i takes at least one step, say $\mathtt{try}_i$, in M_n. So, we have that $|E_\pi| \le \sum_{m \in M_n} s(m) + (c+1)G$. Then, to bound $|E_\pi|$, it suffices to bound $\sum_{m \in M_n} s(m)$.

Claim 4.9.2 $\sum_{m \in M_n} s(m) \le 6cG$.

Proof. We first claim that

$$s(m) \le c(t(m) + \log r(m) + \log w(m) + \log p(m) + 3).$$

Indeed, if m is a read or critical metastep, then $t(m) = 1$, and ENCODE writes at most one symbol `R`, `C`, `PR` or `SR` in the cell associated with m, using c bits. If m is a write metastep, then for each of the $t(m) - 1$ nonwinning steps, ENCODE writes either `R` or `W` in the cell associated with the step, using c bits. For the winning step, ENCODE writes the 3 symbols `PR`, `R` and `W`, and also the numbers $r(m), w(m)$ and $p(m)$. Hence, it uses at most $c(\log r(m) + \log w(m) + \log p(m) + 3)$ bits.

Now, we have

$$\begin{aligned}
\sum_{m \in M_n} s(m) &\le c \sum_{m \in M_n} (t(m) + \log r(m) + \log w(m) + \log p(m) + 3) \\
&\le c \sum_{m \in M_n} (t(m) + r(m) + w(m) + 3) + c \sum_{m \in M_n} p(m) \\
&\le c \sum_{m \in M_n} (2t(m) + 3) + c \sum_{m \in M_n} p(m) \\
&\le 5c \sum_{m \in M_n} t(m) + c \sum_{m \in M_n} p(m) \\
&\le 5cG + c \sum_{m \in M_n} p(m)
\end{aligned}$$

Here, the third inequality holds because $\mathit{steps}(m) = \mathit{reads}(m) \cup \mathit{writes}(m) \cup \mathit{win}(m)$, so that $t(m) = r(m) + w(m)$. The fourth inequality holds because $t(m) \ge 1$, since m contains at least one step. The final inequality holds because $\sum_{m \in M_n} t(m)$ is the total number of steps contained in all the metasteps in M_n, which is G. We have the following.

Claim 4.9.3 $\sum_{m \in M_n} p(m) \le G$.

Proof. Let $R = \{\mu \,|\, (\mu \in M_n) \wedge (\mathit{type}(\mu) = \mathtt{R})\}$ be the set of all read metasteps contained in M_n. Let $m_1, m_2 \in M_n$ be any two different write metasteps. Then $\mathit{preads}(m_1) \subseteq R$, and $\mathit{preads}(m_2) \subseteq R$. Also, by Lemma 4.7.6, we have that for any $m \in R$, if $m \in \mathit{preads}(m_1)$, then $m \notin \mathit{preads}(m_2)$. So, $\mathit{preads}(m_1) \cap \mathit{preads}(m_2) = \emptyset$. Thus, we have

$$\sum_{m \in M_n} |\mathit{preads}(m)| = \sum_{m \in M_n} p(m) \le |R| \le G.$$

$\square$

Variable	**Domain of type**	**Meaning**
E_π	An output of Encode	The input to Decode.
α	$runs(\mathcal{A})$	A linearization of a prefix of $(M_n, \preceq_n)$.
$done$	$2^{[n]}$	Processes that have completed their exit sections.
$pc_i, i \in [n]$	$\mathbb{N}$	Number of steps taken by p_i in α, plus 1.
$e_i, i \in [n]$	$E_i \cup \{\perp\}$	The next step of p_i after α.
$wait_i, i \in [n]$	$2^{[n]}$	Processes p_i such that $e_i \neq \perp$.
$\ell_i, i \in [n]$	$L \cup \{\perp\}$	The register accessed by e_i.
$type_i, i \in [n]$	T	The extended type of e_i in the next p_i metastep after α.
$sig_\ell, \ell \in L$	Record with fields $r, w, pr, win \in 0..n$	Signature of min. write metastep on ℓ not lin. in α.
$R_\ell, \ell \in L$	$2^{[n]}$	Processes p_i such that e_i reads ℓ.
		Also, p_i changes state after reading $val(e_{(sig_\ell.win)})$ in ℓ.
$W_\ell, \ell \in L$	$2^{[n]}$	Processes p_i such that e_i writes to ℓ.
$PR_\ell, \ell \in L$	$2^{[n]}$	Processes p_i that have done final read to ℓ.

Figure 4-6: The types and meanings of variables used in Decode.

Combining Claim 4.9.3 with the expression for $\sum_{m \in M_n} s(m)$, we get $\sum_{m \in M_n} s(m) \leq 6cG$. □

Since $C(\alpha) = G$ by Lemma 4.7.10, then by combining Equation 4.14 and Claim 4.9.2, we have

$$|E_\pi| \leq 6cG + (c+1)G = (7c+1)G = O(C(\alpha)).$$

□

4.10 The Decoding Step

In this section, we describe the decoding step. The input to Decode is a string E_π produced by Encode$(M_n, \preceq_n)$ (where $(M_n, \preceq_n)$ is the output of Construct(π)). Decode outputs a run that is a linearization of $(M_n, \preceq_n)$. For ease of notation, we denote the input to Decode by E.

At a high level, the decoding algorithm proceeds in a loop, where at any point in the loop, it has output a run α that is a linearization of some prefix N of $(M_n, \preceq_n)$. We say the metasteps in N have been *executed*, and we say the metasteps in $M_n \backslash N$ are *unexecuted*. By reading E, the decoder finds a *minimal unexecuted metastep* m, with respect to $\preceq_n$. The decoder executes m, by linearizing m and appending the result to α. It then begins the next iteration of the decoding loop.

Please see Figure 4-7 for the pseudocode for Decode. We refer to line numbers in Decode using angle brackets, with a subscript D. For example, $\langle 6 \rangle_{\text{D}}$ refers to line 6 in Figure 4-7. We first describe the variables in Decode. Please also see Table 4-6. α is the run that the decoder builds. $done \subseteq [n]$ is the set of processes that have completed their trying, critical and exit sections. For $i \in [n]$, pc_i is the number of metasteps the decoder has executed that contain p_i, plus one. e_i is the step p_i takes after α, ℓ_i is the register accessed by e_i, and $type_i$ is the extended type of e_i in the next p_i metastep after α[38]. We call e_i process p_i's *next step*. At certain points in the decoding, the decoder may not

[38]Recall that α is supposed to be the linearization of some prefix N of $(M_n, \preceq_n)$. Thus, by the next p_i metastep

```
procedure DECODE($E$)
    $\forall i \in [n] : pc_i \leftarrow 2,\ type_i \leftarrow \varepsilon,\ e_i \leftarrow \perp,\ \ell_i \leftarrow \perp$
    $\forall \ell \in L : sig_\ell \leftarrow \perp,\ R_\ell, PR_\ell, W_\ell \leftarrow \emptyset$
    $\alpha \leftarrow \mathsf{try}_1 \circ \mathsf{try}_2 \circ \ldots \circ \mathsf{try}_n;\quad done \leftarrow \emptyset;\quad wait \leftarrow \emptyset$
    repeat
        for all $(i \notin done \cup wait)$ do
            $e_i \leftarrow \delta(\alpha, i);\quad \ell_i \leftarrow reg(e_i);\quad wait \leftarrow wait \cup \{i\}$
            $type_i \leftarrow$ GETSTEP$(E, i, pc_i)$
            switch
                case $type_i = \mathtt{W}$:
                    if $type_i$ contains a signature $sig$ then
                        $sig_{\ell_i} \leftarrow$ MAKESIG$(sig, i)$
                    end if
                    $W_{\ell_i} \leftarrow W_{\ell_i} \cup \{i\}$
                case $type_i = \mathtt{R}$:
                    choose $s \in S$ s.t. $(st(s,i) = st(\alpha,i)) \wedge (st(s,\ell_i) = val(e_{(sig_{\ell_i}.win)}))$
                    if $(sig_{\ell_i} \neq \varepsilon) \wedge (\Delta(s, e_i, i) \neq st(\alpha, i))$ then
                        $R_{\ell_i} \leftarrow R_{\ell_i} \cup \{i\}$
                    else
                        $wait \leftarrow wait \backslash \{i\}$
                    end if
                case $type_i = \mathtt{PR}$:
                    $PR_{\ell_i} \leftarrow PR_{\ell_i} \cup \{i\}$
                    $\alpha \leftarrow \alpha \circ e_i$
                    $pc_i \leftarrow pc_i + 1;\quad type_i \leftarrow \varepsilon;\quad e_i \leftarrow \perp;\quad wait \leftarrow wait \backslash \{i\}$
                case $(type_i = \mathtt{SR}) \vee (type_i = \mathtt{C})$:
                    $\alpha \leftarrow \alpha \circ e_i$
                    $pc_i \leftarrow pc_i + 1;\quad type_i \leftarrow \varepsilon;\quad e_i \leftarrow \perp;\quad wait \leftarrow wait \backslash \{i\}$
                case $type_i = \mathtt{\$}$:
                    $done \leftarrow done \cup \{i\}$
            end switch
        end for

        for all $\ell \in L$ such that $sig_\ell \neq \perp$ do
            if $(|R_\ell| = sig_\ell.r) \wedge (|PR_\ell| = sig_\ell.pr) \wedge (|W_\ell| = sig_\ell.w)$
                $\beta \leftarrow concat(\bigcup_{i \in W_\ell \backslash \{sig_\ell.win\}} e_i)$
                $\gamma \leftarrow concat(\bigcup_{i \in R_\ell} e_i)$
                $\alpha \leftarrow \alpha \circ \beta \circ e_{(sig_\ell.win)} \circ \gamma$
                for all $i \in R_\ell \cup W_\ell$ do
                    $pc_i \leftarrow pc_i + 1;\quad type_i \leftarrow \varepsilon;\quad e_i \leftarrow \perp$
                end for
                $wait \leftarrow wait \backslash (R_\ell \cup W_\ell)$
                $sig_\ell \leftarrow \perp;\quad R_\ell, PR_\ell, W_\ell \leftarrow \emptyset$
            end if
        end for
    until $done = \{1, \ldots, n\}$
    return $\alpha$
end procedure

procedure GETSTEP($E$,$i$,$pc$)
    read $E$ until we have read $i - 1$ $\mathtt{\$}$ symbols
    read $E$ until we have read $pc$ # symbols
    return the string up to before the next # symbol
end procedure

procedure MAKESIG($s$,$i$)
    suppose $s = \mathtt{PR}pr\mathtt{R}r\mathtt{W}w$
    $sig.pr \leftarrow pr;\quad sig.r \leftarrow r;\quad sig.w \leftarrow w$
    $sig.win \leftarrow i$
    return $sig$
end procedure
```

Figure 4-7: Decoding $E = E_\pi$ to produce a linearization of $(M, \preceq)$.

yet know the next steps of some processes. If the decoder knows the next step of process p_i, then it places i in *wait*; the idea is that the decoder is waiting to group e_i with some other next steps, which together make up the steps of a minimal unexecuted metastep. For every $\ell \in L$, if $sig_\ell \neq \varepsilon$, then sig_ℓ contains the signature of an unexecuted write metastep m on ℓ. sig_ℓ is a record with four fields, r, w, pr and win. r, w and pr represent the sizes of $reads(m), writes(m) \cup win(m)$, and $preads(m)$,

after α, we mean the next p_i metastep after N

respectively. $sig_\ell.win$ is the name of the winner of m. We say $e_{(sig_\ell.win)}$ is the *winning step* on ℓ. R_ℓ is a set of processes such that the next step of each process is a read on ℓ, and the process would change its state if it read the value of the winning step on ℓ. W_ℓ is a set of processes whose next step is a write to ℓ. PR_ℓ is a set of processes that have done their last read step on ℓ in M_n, and such that the read step is contained in a preread metastep, that itself is contained in the preread set of an unexecuted write metastep on ℓ.

Having described the variables of Decode, we now describe the general aim of these variables. Recall that in Section 4.7.3, we proved several characterizations of the minimal metasteps after a prefix. Suppose that at some point in the execution of Decode, α is a linearization of some prefix N of $(M_n, \preceq_n)$. Then for any $\ell \in L$, the sets PR_ℓ, R_ℓ and W_ℓ in Decode represent $preads(N, \ell)$, $readers(N, \ell, v)$ and $wwriters(N, \ell)$, respectively[39][40]. Also, if $sig_\ell \neq \perp$, then $sig_\ell.pr$, $sig_\ell.r$ and $sig_\ell.w$ equal $|preads(m)|, |reads(m)|$ and $|writes(m)|$, respectively, for some write metastep m on ℓ, such that m is the next π_k metastep after N for some $k \in [n]$. In addition, $sig_\ell.win$ equals $\diamond(winner(m))$. The general strategy of Decode is to use Lemmas 4.7.35 and 4.7.36, which are based on comparing the quantities $|preads(N, \ell)|$, $|readers(N, \ell, v)|$ and $|wwriters(N, \ell)|$ against $|preads(m)|, |reads(m)|$ and $|writes(m)|$, to decide when $m \in \lambda(N)$.

We now describe the operation of Decode. Each iteration of the main repeat loop of Decode consists of two sections, from $\langle 6-32\rangle_{\mathrm{D}}$, and from $\langle 33-44\rangle_{\mathrm{D}}$. The purpose of the first section is to find the next step of each process, and also to execute some minimal unexecuted *read* and *critical* metasteps. The purpose of the second section is to divide the next steps computed in the first section into groups, such that each group of steps is exactly the steps contained in some minimal unexecuted *write* metastep. Then, $\langle 33-44\rangle_{\mathrm{D}}$ also executes these metasteps.

Consider any $i \notin done \cup wait$. That is, p_i is has not finished its exit section, and the decoder does not know its next step. In $\langle 7\rangle_{\mathrm{D}}$, the decoder computes e_i, using the run α it has already generated and p_i's transition function $\delta(\cdot, i)$. In $\langle 8\rangle_{\mathrm{D}}$, the decoder calls the helper function GetStep(E, i, pc_i), which returns the extended type of e_i in p_i's next metastep. The decoder then switches based on the value of $type_i$.

First consider the case $type_i = \mathtt{W}$ $\langle 10\rangle_{\mathrm{D}}$, and let ℓ_i be the register e_i writes to $\langle 7\rangle_{\mathrm{D}}$. Then the decoder adds i to W_{ℓ_i}. In addition, if $type_i$ contains a signature sig, the decoder sets sig_{ℓ_i} to MakeSig(sig, i) $\langle 12\rangle_{\mathrm{D}}$. If $sig = \mathtt{PR}pr\mathtt{R}r\mathtt{W}w$, where pr, r and w are numbers, then MakeSig(sig, i) sets $sig_\ell.win \leftarrow i$ (indicating that p_i is the winner of the metastep corresponding to this signature), $sig_\ell.r \leftarrow r$, $sig_\ell.w \leftarrow w$, and $sig_\ell.pr \leftarrow pr$.

Next, consider the case $type_i = \mathtt{R}$, and let ℓ_i be the register e_i reads. The decoder first checks

[39] $preads(N, \ell)$, $readers(N, \ell, v)$ and $wwriters(N, \ell)$ are defined in Definition 4.7.4.

[40] We say that PR_ℓ, R_ℓ and W_ℓ in Decode *represent* $preads(N, \ell)$, $readers(N, \ell, v)$ and $wwriters(N, \ell)$, because they may not equal $preads(N, \ell)$, $readers(N, \ell, v)$ and $wwriters(N, \ell)$ at all points in the execution of Decode. For example, there may be a point in the execution of Decode when $wwriters(N, \ell) \neq \emptyset$, but $W_\ell = \emptyset$, because the decoder has not yet computed the elements of W_ℓ yet.

whether $sig_{\ell_i} \neq \perp$. If $sig_{\ell_i} \neq \perp$, the decoder then checks whether the (value of the) winning write step in the metastep corresponding to this signature, namely, step $e_{(sig_\ell.win)}$, would cause p_i to change its state $\langle 18 \rangle_{\mathrm{D}}$. If so, the decoder adds i to R_{ℓ_i}. If either of the checks fails, the decoder removes i from *wait*, so that on the next iteration of the decoding loop, the decoder will check whether there exists a possibly different winning step on ℓ_i that will cause p_i to change its state.

Next, consider the case $type_i = \mathtt{PR}$, and let ℓ_i be the register e_i reads. e_i is the lone read step in a read metastep m, and so the decoder executes m by appending e_i to α $\langle 24 \rangle_{\mathrm{D}}$. The decoder then increments pc_i, and removes i from *wait* $\langle 25 \rangle_{\mathrm{D}}$, indicating that it needs to compute a new next step for p_i in the next iteration of the decoding loop . In addition, because $type_i = \mathtt{PR}$, then m is the last read metastep containing p_i on ℓ_i in M_n, and so the decoder adds i to PR_{ℓ_i} $\langle 23 \rangle_{\mathrm{D}}$.

Next, consider the cases $type_i = \mathtt{SR}$ or $type_i = \mathtt{C}$ $\langle 26 \rangle_{\mathrm{D}}$. Then e_i is the lone step in a read or critical metastep m, and so the decoder executes m by appending e_i to α. In addition, it removes i from *wait*, and increments pc_i.

Finally, suppose $type_i = \mathtt{\$}$. This indicates that p_i has finished all its steps in M_n. Thus, the decoder adds p_i to *done* $\langle 30 \rangle_{\mathrm{D}}$.

Now, we describe the second section of the decoding loop, between $\langle 33 - 44 \rangle_{\mathrm{D}}$. Recall that the goal of this section is to divide the next steps into groups, with each group corresponding to the steps in some minimal unexecuted write metastep. The grouping is based on the register accessed by the next steps. In particular, the decoder iterates over all the registers ℓ for which it knows the signature $\langle 33 \rangle_{\mathrm{D}}$. For each ℓ, it checks whether the sizes of R_ℓ, W_ℓ and PR_ℓ match the sizes in sig_ℓ $\langle 34 \rangle_{\mathrm{D}}$. If so, it sets β to be the concatenation, in an arbitrary order, of all the write steps e_i, for $i \in W_\ell \backslash \{sig_\ell.win\}$. It sets γ to be the concatenation of all read steps e_i, for $i \in R_\ell$. Then, it appends $\beta \circ e_{sig_\ell.win} \circ \gamma$ to α. We will show in Lemma 4.11.2 that the steps in $\beta \circ e_{sig_\ell.win} \circ \gamma$ are precisely the steps of some minimal unexecuted write metastep. The decoder removes $R_\ell \cup W_\ell$ from *wait* $\langle 41 \rangle_{\mathrm{D}}$, to indicate that it needs to compute next steps for these processes in the next iteration of the decoding loop. It also increments pc_i, for all the processes $i \in R_\ell \cup W_\ell$. Finally, it resets $sig_\ell, R_\ell, PR_\ell$ and W_ℓ.

The decoder performs the decoding loop between $\langle 5 - 45 \rangle_{\mathrm{D}}$ until $done = [n]$, indicating that all processes have entered their remainder sections. Then it returns the step sequence α it has constructed. We show in Theorem 4.11.4 that α is a linearization of $(M_n, \preceq_n)$.

4.11 Correctness Properties of the Decoding

In this section, we use several lemmas proven in Section 4.7.3 to show Theorem 4.11.4, which states that $\textsc{Decode}(E_\pi)$ outputs a run α that is a linearization $(M_n, \preceq_n)$. This section uses some notation defined in Section 4.7.1.

In the remainder of this section, let ϑ denote an arbitrary execution of $\textsc{Decode}(E_\pi)$. Consider any point in ϑ. Then we call a tuple consisting of the values of all the variables of $\textsc{Decode}(E_\pi)$ (such as pc_i, for all $i \in [n]$, and R_ℓ, for all $\ell \in L$) at that point, a *state* of ϑ. If σ is a state of ϑ and x is a variable of $\textsc{Decode}$, then we use $\sigma.x$ to denote the value of x in σ. In the following, when we say that we prove a statement using *induction* on ϑ, we mean that we prove the statement by assuming that it holds in a certain state in ϑ, then showing that it also holds in a state that occurs later in ϑ. Recall that we refer to line x in $\textsc{Decode}$ by the notation $\langle x\rangle_{\mathrm{D}}$. We say that an *iteration* of ϑ is one execution of the loop between $\langle 5 - 45\rangle_{\mathrm{D}}$ in $\textsc{Decode}$. We do not necessarily induct over the iterations of ϑ. Rather, we often induct on ϑ at a finer granularity, by considering multiple points within an iteration.

One of the components of a state σ is the step sequence $\sigma.\alpha$ that $\textsc{Decode}(E_\pi)$ has built up. The following definition says that σ is *N-correct* if $\sigma.\alpha$ is a linearization of a prefix N of $(M_n, \preceq_n)$.

Definition 4.11.1 *Consider any state σ in ϑ, and let N be a prefix of $(M_n, \preceq_n)$. Then we say σ is N-*correct *if $\sigma.\alpha \in \mathcal{L}(N)$.*

The following lemma says that given any state σ of ϑ, σ is N-correct, for some prefix N of $(M_n, \preceq_n)$. Thus, $\textsc{Decode}(E_\pi)$ always satisfies a safety condition: it never outputs a step sequence that is not a linearization of a prefix of $(M_n, \preceq_n)$.

Lemma 4.11.2 (Safety Lemma) *Let σ be any state in ϑ. Then there exists a prefix N of $(M_n, \preceq_n)$ such that σ is N-correct.*

Proof. The main idea of the proof is to use Lemmas 4.7.35 and 4.7.36, to show that each time the decoder appends a set of steps ω to $\sigma.\alpha$, where $\sigma.\alpha$ is a linearization of a prefix N of $(M_n, \preceq_n)$, then ω is exactly the steps in *steps*(m), for some $m \in \lambda(N)$.

Formally, we use induction on ϑ. Let σ_0 be the state in ϑ at the end of $\langle 4\rangle_{\mathrm{D}}$. Then σ_0 is N_0 correct, for $N_0 = \{\mathsf{try}_1, \ldots, \mathsf{try}_n\}$. For the inductive step, suppose that σ is N-correct, for some prefix N of $(M_n, \preceq_n)$, and suppose that after σ, $\textsc{Decode}$ appends a sequence of steps ω to $\sigma.\alpha$. Then we prove that the set of steps in ω equals the set of steps contained in some minimal unexecuted metastep $m \in \lambda(N)$. From this, it follows that σ' is $(N \cup \{m\})$-correct, where σ' is the state of ϑ after appending ω. In the remainder of this proof, we often suppress the "σ dot" notation when referring to the value of a variable at a point in ϑ. Rather, we will simply indicate the location at which we consider the value of a variable.

There are three places where $\textsc{Decode}$ appends steps to α: in $\langle 24\rangle_{\mathrm{D}}$, $\langle 27\rangle_{\mathrm{D}}$, $\langle 37\rangle_{\mathrm{D}}$. First, suppose that $\textsc{Decode}$ appends a step e_i to α in $\langle 24\rangle_{\mathrm{D}}$ or $\langle 27\rangle_{\mathrm{D}}$. Then we have $type_i \in \{\mathtt{C}, \mathtt{PR}, \mathtt{SR}\}$. Let $m = \lambda(N, \pi^{-1}(i))$ be the next p_i metastep after N. Since $type_i \in \{\mathtt{C}, \mathtt{PR}, \mathtt{SR}\}$, we have $type(m) \in \{\mathtt{C}, \mathtt{R}\}$, and so by Lemma 4.7.36, we have $m \in \lambda(N)$. Let ϵ be the step that p_i takes in m. Then we have $e_i = \epsilon$, and so $\alpha \circ e_i$ is N'-correct, for $N' = N \cup \{m\}$.

Next, suppose DECODE appends a sequence of steps ω to α in $\langle 37\rangle_{\mathrm{D}}$. Then from $\langle 34\rangle_{\mathrm{D}}$, there exists some $\ell \in L$, such that $|W_\ell| = sig_\ell.w$, $|R_\ell| = sig_\ell.r$ and $|PR_\ell| = sig_\ell.pr$. For any process $i \in [n]$, let $e_i = \delta(\alpha, i)$. Also, let $k = sig_\ell.win$, and let $m = \lambda(N, \pi^{-1}(k))$. Since sig_ℓ contains the signature for m, we see by inspection of the ENCODE algorithm that the following hold:

1. p_k is the winner of m.

2. e_k is a write step.

3. $sig_\ell.r = |readers(m)|$, $sig_\ell.pr = |preads(m)|$ and $sig_\ell.w = |writers(m) \cup winner(m)|$.

From $\langle 10 - 14\rangle_{\mathrm{D}}$, we see that W_ℓ is the set of processes p_i such that e_i is a write step to ℓ, and e_i belongs to a metastep not contained in N. Thus, we have $W_\ell = writers(N, \ell)$. Then, since $|W_\ell| = sig_\ell.w = |writers(m) \cup winner(m)|$, we get that

$$|wwriters(N, \ell)| = |writers(m) \cup winner(m)|.$$

Next, from $\langle 15 - 21\rangle_{\mathrm{D}}$, we see that R_ℓ is the set of processes p_i such that e_i is a read step on ℓ, e_i belongs to a metastep not contained in N, and reading value $val(m)$ in ℓ causes p_i to change from its current state $st(\alpha, i)$[41]. Thus, we have $R_\ell = readers(N, \ell, val(m))$. Since $|R_\ell| = sig_\ell.r = |readers(m)|$, then we get that

$$|readers(N, \ell, val(m))| = |readers(m)|.$$

Finally, we see from $\langle 23 - 25\rangle_{\mathrm{D}}$ that PR_ℓ is the set of processes p_i that have performed a read metastep contained in N, such that the read metastep is contained in the preread set of some write metastep not contained in N. Thus, $PR_\ell = preads(N, \ell)$. Since $|PR_\ell| = sig_\ell.pr = |preads(m)|$, we get that

$$|preads(N, \ell)| = |preads(m)|.$$

Combining this with the earlier facts that $|readers(N, \ell, val(m))| = |readers(m)|$ and $|wwriters(N, \ell)| = |writers(m) \cup winner(m)|$, and applying Lemma 4.7.35, we get that $m \in \lambda(N)$. Thus, letting ω be $\beta \circ e_{sig_\ell.win} \circ \gamma$, where β and γ are defined as in $\langle 35 - 36\rangle_{\mathrm{D}}$, we get that $\alpha \circ \omega$ is N'-correct, for $N' = N \cup \{m\}$.

From the above, we have that if α is N-correct, then after DECODE appends a sequence of steps to α, the resulting run is N'-correct, for some prefix $N' \supset N$ of $(M_n, \preceq_n)$. Thus, the lemma holds by induction. $\square$

Lemma 4.11.2 showed that if DECODE(E_π) ever appends a sequence of steps to α, then those

[41] Note that $val(m)$ is the value written by step $e_{sig_\ell}.win$.

steps correspond to the steps in some minimal unexecuted metastep. The next lemma shows a liveness property, that in every iteration of ϑ, $\textsc{Decode}(E_\pi)$ does append some steps to α.

Lemma 4.11.3 (Liveness Lemma) *Let σ be the state at $\langle 6\rangle_{\mathrm{D}}$ in some iteration of ϑ, and let σ be the state at $\langle 44\rangle_{\mathrm{D}}$ in the same iteration. Then either $\sigma'.done = [n]$, or $\sigma.\alpha$ is a strict prefix of $\sigma'.\alpha$.*

Proof. By Lemma 4.11.2, σ is N-correct, for some prefix N of $(M_n, \preceq_n)$. Suppose $\sigma'.done \neq [n]$. Then there exists $i \in [n]$ such that $\lambda(N, i) \neq \emptyset$, and so $\lambda(N) \neq \emptyset$. Let $m \in \lambda(N)$, and suppose first that $type(m) \in \{\mathtt{C}, \mathtt{R}\}$. Let $i \in procs(m)$. Then we see that at $\langle 9\rangle_{\mathrm{D}}$ after σ, we have $type_i \in \{\mathtt{C}, \mathtt{PR}, \mathtt{SR}\}$, and so in $\langle 24\rangle_{\mathrm{D}}$ or $\langle 27\rangle_{\mathrm{D}}$, we have $\alpha \leftarrow \alpha \circ e_i$. Thus, the lemma holds.

Next, suppose that $type(m) = \mathtt{W}$, and let $\ell = reg(m)$ and $v = val(m)$. Then, following the arguments in the proof of Lemma 4.11.2, we have at $\langle 34\rangle_{\mathrm{D}}$ after σ that $R_\ell = readers(N, \ell, v), W_\ell = wwriters(N, \ell)$, and $PR_\ell = preads(N, \ell)$. Also, we have at $\langle 34\rangle_{\mathrm{D}}$ that $sig_\ell.r = |readers(m)|$, $sig_\ell.w = |writers(m) \cup winner(m)|$ and $sig_\ell.pr = |preads(m)|$. Since $m \in \lambda(N)$, then by Lemma 4.7.35, we have $|readers(N, \ell, v)| = |readers(m)|$, $|wwriters(N, \ell)| = |writers(m) \cup winner(m)|$ and $|preads(N, \ell)| = |preads(m)|$. Thus, we have $|W_\ell| = sig_\ell.w$, $|R_\ell| = sig_\ell.r$ and $|PR_\ell| = sig_\ell.pr$ at $\langle 34\rangle_{\mathrm{D}}$, and so in $\langle 37\rangle_{\mathrm{D}}$, $\textsc{Decode}$ appends $\beta \circ e_{sig_\ell}.win \circ \gamma$ to α. Thus, the lemma holds. $\square$

Theorem 4.11.4 (Decoding Theorem A) *Let α be the output of* $\textsc{Decode}$. *Then α is a linearization of $(M_n, \preceq_n)$.*

Proof. By Lemma 4.11.2, $\sigma.\alpha$ is a linearization of some prefix N of $(M_n, \preceq_n)$, for any state σ in ϑ. By Lemma 4.11.3, $\textsc{Decode}$ continues to append steps to α until $done = [n]$. We can see that $done = [n]$ precisely when all the metasteps in M_n have been linearized in α. Thus, the final output α of $\textsc{Decode}$ is a linearization of $(M_n, \preceq_n)$. $\square$

4.12 A Lower Bound on the Cost of Canonical Runs

In this section, we use the main theorems shown in Sections 4.6.5, 4.9 and 4.11 to prove that there exists a canonical run α with $\Omega(n \log n)$ cost in the state change cost model. We begin with the following definition.

Definition 4.12.1 *Let $\pi \in S_n$ be an arbitrary permutation. Then we define the following.*

1. *Let $(M_\pi, \preceq_\pi)$ be any output of* $\textsc{Construct}(\pi)$.

2. *Let E_π be any output of* $\textsc{Encode}(M_\pi, \preceq_\pi)$.

3. *Let α_π be any output of* $\textsc{Decode}(E_\pi)$.

Lemma 4.12.2 (Uniqueness Lemma) *Let $\pi_1, \pi_2 \in S_n$, such that $\pi_1 \neq \pi_2$. Then $\alpha_{\pi_1} \neq \alpha_{\pi_2}$.*

Proof. By Theorem 4.11.4, α_{π_1} is a linearization of $(M_{\pi_1}, \preceq_{\pi_1})$, and α_{π_2} is a linearization of $(M_{\pi_2}, \preceq_{\pi_2})$. Thus, by Theorem 4.6.20, processes $p_1, \ldots, p_n$ all enter the critical section in α_{π_1}, and they enter in the order π_1. $p_1, \ldots, p_n$ also all enter the critical section in α_{π_2}, and they enter in the order π_2. Thus, since $\pi_1 \neq \pi_2$, then we have $\alpha_{\pi_1} \neq \alpha_{\pi_2}$. □

Finally, we prove our main lower bound. It states that for any mutual exclusion algorithm $\mathcal{A}$, there is a canonical run α of $\mathcal{A}$, in which each process $p_1, \ldots, p_n$ enters and exits the critical section once, such that the cost of α in the state change cost model is $\Omega(n \log n)$. Recall that $\mathcal{C}$ is the set of canonical runs.

Theorem 4.12.3 (Main Lower Bound) *Let $\mathcal{A}$ be any algorithm solving the mutual exclusion problem. Then there exists a $\pi \in S_n$ such that $\alpha_\pi \in \mathcal{C}$, and $C(\alpha_\pi) = \Omega(n \log n)$.*

Proof. By Theorem 4.6.21, we have $\alpha_\pi \in \mathcal{C}$, for all $\pi \in S_n$. Assume for contradiction that the theorem is false. Then for all $\pi \in S_n$, we have $C(\alpha_\pi) = o(n \log n)$. Since $|E_\pi| = O(C(\alpha_\pi))$ by Theorem 4.9.1, then we have $|E_\pi| = o(n \log n)$, for all $\pi \in S_n$. Since $2^{o(n \log n)} = o(n!)$ and $|S_n| = n!$, we have $|\{E_\pi\}_{\pi \in S_n}| < |S_n|$. Then by the pigeonhole principle, there exists $\pi_1, \pi_2 \in S_n$ with $\pi_1 \neq \pi_2$ such that $E_{\pi_1} = E_{\pi_2}$. Thus, we have

$$\alpha_{\pi_1} = \textsc{Decode}(E_{\pi_1}) = \textsc{Decode}(E_{\pi_2}) = \alpha_{\pi_2}.$$

But by Lemma 4.12.2, we have $\alpha_{\pi_1} \neq \alpha_{\pi_2}$, which is a contradiction. Thus, there must exist a $\pi \in S_n$ such that $C(\alpha_\pi) = \Omega(n \log n)$. □

Bibliography

[1] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 12–21. IEEE, 1992.

[2] James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 90–99, New York, NY, USA, 2001. ACM Press.

[3] James H. Anderson and Yong-Jik Kim. Nonatomic mutual exclusion with local spinning. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 3–12, New York, NY, USA, 2002. ACM Press.

[4] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 2003.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[6] Hagit Attiya and Danny Hendler. Time and space lower bounds for implementations using -cas. In *DISC*, pages 169–183, 2005.

[7] Saâd Biaz and Jennifer L. Welch. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Information Processing Letters*, 80(3):151–157, 2001.

[8] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Compututation*, 107(2):171–184, 1993.

[9] Robert Cypher. The communication requirements of mutual exclusion. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 147–156, New York, NY, USA, 1995. ACM Press.

[10] Danny Dolev, Joe Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 504–511, New York, NY, USA, 1984. ACM Press.

[11] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.

[12] Rui Fan, Indraneel Chakraborty, and Nancy Lynch. Clock synchronization for wireless networks. In *OPODIS 2004: 8th conference on principles of distributed systems*, pages 400–414. Springer, 2004.

[13] Rui Fan and Nancy Lynch. Gradient clock synchronization. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 320–327, New York, NY, USA, 2004. ACM Press.

[14] Rui Fan and Nancy Lynch. Gradient clock synchronization. *Distributed Computing*, 18(4):255–266, 2006.

[15] Rui Fan and Nancy Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 275–284, New York, NY, USA, 2006. ACM.

[16] C. Fetzer and F. Cristian. Integrating external and internal clock synchronization. *Journal of Real-Time Systems*, 12(2):123–172, 1997.

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[18] Seth Gilbert. *Virtual Infrastructure for Wireless Ad Hoc Networks*. PhD thesis, MIT, 2007.

[19] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 1990.

[20] Joseph Y. Halpern, Nimrod Megiddo, and Ashfaq A. Munshi. Optimal precision in the presence of uncertaint. *Journal of Complexity*, 1(2):170–196, 1985.

[21] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 201–210, New York, NY, USA, 1998. ACM Press.

[22] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. *The Theory of Timed I/O Automata*. Morgan and Claypool, 2005.

[23] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 23–32, New York, NY, USA, 1999. ACM Press.

[24] Leslie Lamport and P. Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[25] Errol Lloyd. Broadcast scheduling for tdma in wireless multihop networks. *Handbook of wireless networks and mobile computing*, pages 347–370, 2002.

[26] Thomas Locher and Roger Wattenhofer. Oblivous gradient clock synchronization. In *DISC '06: 20th International Symposium on Distributed Computing*, pages 520–533, 2006.

[27] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[28] Lennart Meier and Lothar Thiele. Gradient clock synchronization in sensor networks. Technical report, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, 2005.

[29] J. Mellor-Crummey and M. Scott. Algorithms for scalable sychronization on shared-memory multicomputers. *ACM Transations on Computer Systems*, 1991.

[30] D. L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Computers*, 39(10):1482–1493, 1991.

[31] Rafail Ostrovsky and Boaz Patt-Shamir. Optimal and efficient clock synchronization under drifting clocks. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 3–12. ACM Press, 1999.

[32] Boaz Patt-Shamir and Sergio Rajsbaum. A theory of clock synchronization. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 810–819. ACM Press, 1994.

[33] Hairong Qi, Xiaoling Wang, S. Sitharama Iyengar, and Krishnendu Chakrabarty. Multisensor data fusion in distributed sensor networks using mobile agents. In *Proceedings of the International Conference on Information Fusion*, pages 11–16, 2001.

[34] Michael Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, Cambridge, Massachusetts, 1986.

[35] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[36] An swol Hu and Sergio D. Servetto. Algorithmic aspects of the time synchronization problem in large-scale sensor networks. *Mob. Netw. Appl.*, 10(4):491–503, 2005.

[37] P. Verissimo, L. Rodrigues, and A. Casimiro. *Cesiumspray: a precise and accurate global time service for large-scale systems.* Technical Report NAV-TR-97-0001, Universidade de Lisboa, 1997.

[38] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S.J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[39] Jennifer Lundelius Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[40] Y.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 1995.