ALL OF US ARE SMARTER THAN ANY OF US: NONDETERMINISTIC WAIT-FREE HIERARCHIES ARE NOT ROBUST*

WAI-KAU LO† AND VASSOS HADZILACOS‡

Abstract. A wait-free hierarchy [Herlihy91, Jayanti93] classifies object types on the basis of their strength in supporting wait-free implementations of other types. Such a hierarchy is robust if it is impossible to implement objects of types that it classifies as "strong" by combining objects of types that it classifies as "weak". We prove that, if nondeterministic types are allowed, the only wait-free hierarchy that is robust is the trivial one, which lumps all types into a single level. In particular, the Consensus hierarchy (the most closely studied wait-free hierarchy) is not robust. Our result implies that, in general, it is not possible to determine the power of a concurrent system that supports a given set of primitive object types by reasoning about the power of each primitive type in isolation.

Key words. wait-free, consensus number, robustness

AMS subject classifications. 68Q10, 68Q22

1. Introduction.

1.1. Background and overview of the result. Interest in asynchronous shared-memory distributed computing dates back at least to Dijkstra's work on mutual exclusion [4]. In this model of computation, a set of processes cooperate towards some common task (such as excluding each other from a critical section) by asynchronously taking steps, some of which access shared registers. The operations that the processes can apply to the shared registers may be simple read and write operations, or more complex ones such as test-and-set, fetch-and-add, and so forth.

In a more general version of this model of computation, processes communicate by accessing shared objects. An object is accessed by applying an operation to it; the operation changes the state of the object and returns a value to the process that invoked it. Each object belongs to a type. A type consists of a set of states, a set of operations, a set of responses to operations, and a state transition relation that describes the effect of applying each operation to an object of that type. More specifically, the state transition relation specifies the responses the object may return and the states it may enter if an operation is applied when the object is in a particular state. Examples of types include register (with read and write operations only), test-and-set-register (with test-and-set operation), stack, queue and priority-queue, among many others. (We use the sans serif font to indicate object types.)

A distributed system supports, in hardware or software, a certain fixed set of "base" object types. To enhance the usability of such a system, a programmer may wish to enrich the set of available object types by implementing new types, not directly supported by the system. To implement a type T from a set of base types $\mathcal{B}$ in a system of n processes, the programmer must provide n procedures for each operation

*A preliminary version of the results in this paper appeared in the Proceedings of the 29th ACM SIGACT Symposium on Theory of Computing, El Paso, Texas, May 1997.

†Supported by a Canadian Commonwealth Scholarship. Email: wklo@cs.utoronto.ca

‡Supported by a grant from the Natural Sciences and Engineering Research Council of Canada and a fellowship from the Engineering and Physical Sciences Research Council of the U.K. Address address: Department of Computer Science, University of Toronto, 6 King's College Rd., Toronto, Ontario, Canada, M5S 3H5. Email: vassos@cs.utoronto.ca

op of T; these are the procedures that may be invoked by the n processes to apply *op* to an object of type T. These procedures may use any number of objects, all belonging to types in $\mathcal{B} \cup \{\mathsf{register}\}$. Note that this notion of implementation takes registers for granted. (For convenience, in the rest of the paper we say simply "registers" instead of "objects of type register".) This is reasonable since any realistic shared-memory system will provide at least this much.[1]

An implementation is required to be *linearisable* [7]: Any concurrent execution E consisting of invocations of the procedures written by the programmer must be equivalent to a *sequential* execution E_s consisting of the same procedure invocations, where each invocation returns a response and changes the state of the object it accesses in accordance with the state transition relation of the implemented type. Moreover, the sequential execution E_s must respect the order of procedure invocations that did not overlap in E; it may serialise in any way procedure invocations that were active concurrently in E.

Of special interest are *wait-free* implementations. In such implementations, a process is guaranteed to complete any procedure it invokes within a finite number of its own steps, regardless of whether other processes are fast, slow, or even crashed (i.e., halted before completing). The requirement of wait freedom precludes the use of critical sections and busy waiting. In this paper, we consider only wait-free and linearisable implementations.

A fundamental question is whether, in a system of n processes, it is possible to implement a type T from a given set of base types $\mathcal{B}$. In a seminal paper [6], Herlihy established an important link between a type's ability to support implementations of other types and its ability to solve the well-known Consensus problem [5], in which processes must reach agreement on one of their input values (this problem is formally defined in Section 3.1). More specifically, he showed that if objects that belong to a set of types $\mathcal{B}$ (together with registers) can be used to solve Consensus among n processes, then objects that belong to types in $\mathcal{B}$ (together with registers) can be used to implement *any* type in a system of n processes. This fact is referred to as "the universality of Consensus".

Define the *Consensus number* of a type T, as the maximum number n such that there is an algorithm that solves Consensus among n processes using only objects of type T (and registers) — or ∞, if no such maximum exists. We can now place each type in the *Consensus hierarchy* according to its Consensus number. By the universality of Consensus, the relative strength of two types T and T' is captured by their levels, say n and n' respectively, in the Consensus hierarchy. If $n \geq n'$, then T is at least as strong as T': in a system of n processes, T can be used to implement T'. If, further, $n > n'$, then T is strictly stronger than T': T' cannot be used to implement T in a system of n processes.

Although Consensus numbers allow us to make *pairwise* comparisons on the strength of types, it is not clear whether they can also be used to answer the more general implementability question: Can we implement T from a *set* of base types $\mathcal{B}$, in a system of n processes? Jayanti [8] observed that Consensus numbers could be used to answer this question if the Consensus hierarchy is *robust*: i.e., no type at level n can be implemented from types at levels strictly below n, in a system of n processes.

[1] The notion of implementation that we use in this paper is the most prevalent in the literature, although more restricted ones have also been studied. For instance, in some definitions registers are not taken for granted, while in others only one object of each base type may be used in the implementation. For further discussion on such more restricted definitions, see [8] and [10].

If the Consensus hierarchy is robust, a type T of Consensus number n can be implemented from the set of base types $\mathcal{B}$ in a system of n processes if and only if $\mathcal{B}$ contains a type with Consensus number n or more. Thus, we can answer the implementability question by looking at the Consensus numbers of the base types and the desired target type. Another way of thinking about robustness is that, if the Consensus hierarchy has this property, a set of types $\mathcal{B}$ is as strong as the strongest of the types it contains. Thus, we can determine the power of a given set of types by reasoning about the power of each type in isolation. If, on the other hand, the Consensus hierarchy is not robust, then to determine the power of a set of types we must, in general, take into account the interactions between the types in the set — a more formidable task. In view of the importance of the Consensus hierarchy and the salutary consequences of robustness it is useful to know whether the Consensus hierarchy is robust.

Borowsky *et al.* [1], and Peterson *et al.* [15] proved that if we restrict our attention to *deterministic* types, the Consensus hierarchy is robust. A type is deterministic, if whenever we apply an operation to the type, the current state uniquely determines the response of the operation and the new state of the type. Although most of the common types are deterministic, some useful types are most naturally specified as nondeterministic ones. For example, consider a priority queue where keys are not unique. If several elements are in the queue with the same maximum priority, it is natural to allow the priority queue to remove and return any one of these elements. It is, of course, always possible to turn a nondeterministic type into a deterministic one by restricting the allowable behaviour of the type. This, however, results in unnecessary restrictions for the implementor of the type. As pointed out in [11], by arbitrarily and artificially excluding legitimate behaviour, we may be inadvertently ruling out efficient implementations of the type. Thus, the investigation of properties of nondeterministic types is of practical relevance.

In this paper we prove that if we allow nondeterministic types, then the Consensus hierarchy is *not* robust. In view of this, a natural question is whether there is some other hierarchy of types which, like the Consensus hierarchy, classifies types according to their strength in supporting wait-free implementations, but, unlike the Consensus hierarchy, is robust. Jayanti proved that only a coarsening of the Consensus hierarchy can have these properties [8]. In this paper we show that, in fact, the only hierarchy that has these properties is the trivial one in which all types are lumped into level one!

1.2. Overview of the proof. To prove our result we demonstrate two nondeterministic (but finite) object types, negation and booster; each of these is at level one of the Consensus hierarchy, but, when combined, they can implement any type in a system of n processes, for any n. The central idea behind our proof is the formulation of *Equality Negation*, a weaker variant of the Consensus problem which is formally defined in Section 3.2. An object of type negation enables processes to directly solve this problem. An object of type booster allows processes to solve Consensus provided they can prove that they can solve Equality Negation amongst them using initial values specified by the booster object.

We give a simple algorithm that allows n processes to solve Consensus using a negation object, a booster object and registers. (This algorithm suggests itself almost immediately from the definition of the booster type.) By the universality of Consensus, this implies that the two types together can implement any type in a system of n processes. The difficult part of our result is in proving that each of negation and

booster has Consensus number one. The usual technique for proving upper bounds on the Consensus number of a type, known as the "bivalence argument" [5], works well for deterministic types but runs into difficulties in the context of nondeterministic types. The bulk of the technical work in the paper (Sections 7 and 8) seeks ways to circumvent these difficulties.

1.3. Related work. Jayanti was the first to raise the question of robustness of the Consensus hierarchy [8]. Before his work, it had been widely, though tacitly, assumed that the hierarchy is robust. Jayanti showed that various natural variants of the Consensus hierarchy (including the one used in [6]) are not robust. He also proved that, if a robust hierarchy that classifies types according to their ability to implement other types exists, it must be either the Consensus hierarchy or a coarsening of it — i.e., a hierarchy obtained by collapsing together adjacent levels of the Consensus hierarchy.

As mentioned earlier, Borowsky *et al.* [1] and Peterson *et al.* [15] proved that the Consensus hierarchy, restricted to deterministic types, is robust. In recent work, Ruppert [18] gave simple proofs of robustness if we restrict our attention to certain interesting classes of deterministic types.

In the next paragraph we describe some results concerning the (non)robustness of the Consensus hierarchy in a model of objects that differs from the "standard" model introduced in [6] (and used in the present paper). To explain the different models recall that, in a system of n processes, each object of type T has n procedures for each operation *op* of T. Processes trying to apply *op* to an object concurrently must invoke different procedures. In the "standard" model of objects a process that wishes to apply *op* to the object invokes one of these n procedures, and remains bound to it only for the duration of the invocation. Other processes may later use that procedure. In the alternative model, each process permanently binds itself to one procedure. Objects in this model are sometimes called *hardwired*, since the binding of processes to procedures is fixed.

Chandra *et al.* [2] prove that, in the model of hardwired objects, the Consensus hierarchy is not robust if nondeterministic types are allowed. Moran and Rappoport [14] strengthen this result by proving that it applies even if only deterministic types are allowed. We borrowed from [2] the idea of a booster type, which "boosts" the power of a weak type T, if it is given satisfactory evidence that processes have access to T. The booster of [2] promotes a type that solves Consensus between two processes to a type that solves Consensus among three processes. In our case, the booster promotes a type that solves Equality Negation for n processes to a type that solves Consensus for n processes.

Closest to our results is the work of Schenk [19, 20], which precedes ours. He also exhibits two nondeterministic object types each of which (together with registers) is individually too weak to solve Consensus between two processes, but which taken together can solve Consensus for any number of processes. Our result strengthens this in two related ways: First, Schenk's result requires a type that exhibits infinite nondeterminism, while our types exhibit only bounded nondeterminism. Second, and perhaps more importantly, his result requires a definition of wait freedom that is different from the usual one. Specifically, he defines an algorithm to be wait-free if, for any input, there is a bound on the number of steps a process may take before it halts. In the context of objects with infinite nondeterminism this definition is strictly stronger than the usual definition of wait freedom, which requires only that all executions of the algorithm be finite.

In [17], Rachman proved that for each n, there is a (nondeterministic) type with Consensus number n that cannot implement some type with Consensus number smaller than n, in a system of $2n + 1$ processes. This result addresses a point of implementability between types, but it has no bearing on the nonrobustness of the Consensus hierarchy. Informally, nonrobustness asserts that some "strong" type (i.e., a type at level n) *can be implemented by* a combination of "weak" types (i.e., types at levels below n), *in a system of n processes*. In contrast, Rachman's result says that some "strong" type (i.e., a type at level n) *cannot implement* some "weak" type (i.e., a type at a level below n), *in a system with sufficiently many processes*.

1.4. Organisation of the paper. In Section 2 we define the model of computation used in this paper. In Section 3 we review the Consensus problem and introduce the Equality Negation problem. In Section 4 we define the types negation and booster. In Section 5 we show how to solve Consensus for any number of processes using registers along with a negation and a booster object. In Section 6 we define some concepts and derive technical results needed in the next two sections. In Sections 7 and 8 we prove that registers and negation objects alone, as well as registers and booster objects alone, cannot be used to solve Consensus for two processes. Section 9 concludes the paper.

2. The model of computation. In this section we describe the model of computation used in this paper.

2.1. Types and objects. An *object type* T is a tuple (Q, OP, RES, δ), where Q is a set of *states*, OP is a set of *operations*, RES is a set of *responses* to operations, and $\delta \subseteq Q \times OP \times RES \times Q$ is a *state transition relation*, which describes the possible behaviours of objects of type T, and is sometimes called the type's *sequential specification*. More specifically, $(q, op, res, q') \in \delta$ means that if the current state of an object of type T is q and operation op is applied to it, then it is possible that response res will be returned to the operation and the object will enter state q'. We say that T is *deterministic* if for any $q \in Q$ and $op \in OP$, there is at most one $res \in RES$ and $q' \in Q$ so that $(q, op, res, q') \in \delta$; otherwise, we say that T is *nondeterministic*.

For convenience, in this paper we require each type (Q, OP, RES, δ) to be *total*: For any $q \in Q$ and $op \in OP$, there is at least one $res \in RES$ and $q' \in Q$ such that $(q, op, res, q') \in \delta$. That is, the type can do *something* in response to any operation in any state. Further, we assume that, for any $q \in Q$, $op \in OP$ and $res \in RES$, there is at most one $q' \in Q$ so that $(q, op, res, q') \in \delta$. In other words, if an operation is applied in a given state, the type does not lead to different states unless it returns different responses. We remark that this assumption is not necessary for the results in this paper, but it simplifies the presentation of their proofs. All types considered in this paper have this property, so we can make this assumption without loss of generality.

An *object* is an instance of an object type. For the purposes of this paper, we can think of an object O of type T as an automaton whose states and state transition relation are as in T — except that states are labeled with O, to distinguish them from the states of other objects of the same type.

2.2. Processes and algorithms. A *process* is a deterministic automaton that interacts with objects. More precisely, let $\mathcal{O}$ be a set of objects, and let OP and RES be the set of all operations and responses, respectively, of the types to which the objects in $\mathcal{O}$ belong. We define a *process that uses* $\mathcal{O}$ as a tuple $P = (\Sigma, \Sigma_0, \nu, \tau)$, where Σ is a set of *states*, $\Sigma_0 \subseteq \Sigma$ is a set of *initial states*, and $\nu : \Sigma \to OP \times \mathcal{O}$ and $\tau : \Sigma \times RES \to \Sigma$ are functions that describe the interaction of the process with the

objects. Intuitively, if P is in a state $\sigma \in \Sigma$ and $\nu(\sigma) = \langle op, O \rangle$, then in its next step P will apply operation op to object O. (If $\nu(\sigma) = \langle op, O \rangle$, we require that op be an operation of O's type.) Based on its current state, O will return a response res to P and will enter a new state, in accordance with the state transition relation of the type to which O belongs. Finally, P will enter state $\tau(\sigma, res)$, as a result of the response it received from O. For convenience, we assume that the function ν is total. In other words, it is always possible for a process to apply another operation. Note that we can easily accommodate processes that terminate in this framework by imagining that a process that reaches a final state applies infinitely many "DO-NOTHING" operations to a local dummy object which always returns *ack* to such operations.

An *algorithm* $\mathbf{A}$ consists of a set of processes Π, a set of objects $\mathcal{O}$ so that each $P \in \Pi$ uses a subset of $\mathcal{O}$, and an initial state for each object in $\mathcal{O}$. The designated initial state of an object is one of the states of the type to which the object belongs. If $\mathcal{S}$ is a set of types, we say that $\mathbf{A}$ *uses* $\mathcal{S}$ if every object that $\mathbf{A}$ uses belongs to a type in $\mathcal{S}$ and $\mathbf{A}$ uses at least one object of every type in $\mathcal{S}$. A *configuration* C of $\mathbf{A}$ is a tuple consisting of a state for each process in Π and each object in $\mathcal{O}$. C is an *initial* configuration of $\mathbf{A}$ if each process is in one of its initial states, and each object is in the state designated as the initial state by $\mathbf{A}$.

A *step* of process P is a tuple (P, op, O, res); this indicates that P has applied operation op to object O and received response res. Let $P = (\Sigma, \Sigma_0, \nu, \tau)$ and let C be a configuration, where the state of P in C is σ. If $\nu(\sigma) = \langle op, O \rangle$, we say that P has operation op to O *pending* in C. If, in addition, the state of O in C is q, and res is a legitimate response of O to op in state q (i.e., for some state q' of O, (q, op, res, q') is in the state transition relation of O's type), then we say that the step $e = (P, op, O, res)$ is *applicable* to C. If $e = (P, op, O, res)$ is applicable to C, $e(C)$ denotes the configuration resulting from C after step e. More precisely, if in C the state of P is σ and the state of O is q, then $e(C)$ is the configuration in which P has state $\tau(\sigma, res)$, O is in state q' such that (q, op, res, q') is in the state transition relation of O's type, and all other processes and objects are in the same state as in C.[2]

A *schedule* S *of algorithm* $\mathbf{A}$ is a (finite or infinite) sequence of steps of $\mathbf{A}$'s processes. $S = e_1, e_2, \ldots, e_i, \ldots$ is *applicable* to a configuration C, if e_1 is applicable to C, and e_{i+1} is applicable to $e_i(e_{i-1}(\ldots(e_1(C))\ldots))$, for all i. If S is finite and has k steps, $S(C)$ denotes $e_k(e_{k-1}(\ldots(e_1(C))\ldots))$, i.e., the configuration that results after applying the steps in S one at a time, starting with configuration C. If S and S' are schedules such that S is finite, then $S \cdot S'$ denotes their concatenation. If every step in schedule S is a step of the same process P, then S is called a *solo* schedule of P. If S is an infinite schedule and process P has infinitely many steps in S, then P is *correct* in S.

2.3. Wait-free hierarchies and robustness. Recall from Section 1.1 that the Consensus number of a type T is the maximum number n so that there is an algorithm that solves Consensus among n processes and uses only objects of type T and registers; if there is no such maximum, the Consensus number of T is ∞. By the universality of Consensus, the Consensus number of a type is a measure of its strength, i.e., of its ability to support (wait-free and linearisable) implementations of other types.

[2] It is here we use the assumption that, to go to different states, the object must return different responses. Without this assumption, we have to define $e(C)$ as a *set* of configurations, rather than as a single one.

A *hierarchy* (of types) is a function that assigns to each type a positive integer or ∞, called the type's *level*. The intention is that a type's level reflects its strength: the higher the level, the stronger the type. A hierarchy is *wait-free* if no type is assigned to a level higher than the type's Consensus number. Intuitively, a wait-free hierarchy does not overstate any type's strength. Thus, a hierarchy must be wait-free in order to comply with the intention that its levels somehow reflect the strength of types. The *Consensus hierarchy* is the hierarchy that maps each type exactly to its Consensus number. Clearly, the Consensus hierarchy is wait-free.

A hierarchy h is *robust* if for any $n \geq 1$ such that there is a type at level n and any set of types $\mathcal{B}$, if there is an algorithm that solves Consensus among n processes and uses only objects that belong to types in $\mathcal{B}$ and registers, then $\mathcal{B}$ contains a type of level n or higher. Thus, if a wait-free hierarchy is robust, a type that is classified as "strong" by the hierarchy cannot be implemented by types that are classified as "weak". To see this assume the contrary: a type T at level n can be implemented using only objects of types below n (and registers). Since the hierarchy is wait-free and T is at level n, we can solve Consensus among n processes using only objects of type T (and registers). Since T can be implemented from types in levels below n and registers, we can solve Consensus among n processes using only objects of types in levels below n and registers, which contradicts the definition of robustness.

A hierarchy is *nontrivial* if it assigns types to at least two different levels. Jayanti and Toueg [9], showed that the Consensus hierarchy assigns types to all levels, and so the Consensus hierarchy is nontrivial.

3. The Consensus and Equality Negation problems. In this section we review the Consensus problem and define Equality Negation, a problem that plays a central role in our results.

3.1. The Consensus problem. In the Consensus problem, each of n processes starts with a private initial value, drawn from the set $\{0, 1\}$. Each correct process must eventually irrevocably decide one of these initial values, so that no two processes decide different values. Thus, a *Consensus algorithm* $\mathbf{A}_c$ *for* n *processes* satisfies the following properties. Each process in $\mathbf{A}_c$ has two initial states associated with input values 0 and 1, respectively, and two disjoint sets of states associated with decisions 0 and 1, respectively. Since a decision is irrevocable, we require that once a process has entered a state associated with decision $d \in \{0, 1\}$, all states that it may subsequently enter are also associated with d. In addition, for any schedule S that is applicable to an initial configuration I of $\mathbf{A}_c$, the following three properties hold:

Termination: If S is infinite, then for every correct process in S there is a prefix S' of S such that the process has decided in $S'(I)$.

Validity: If a process has decided in $S(I)$, then its decision must be the initial value of some process in I.

Agreement: If two processes have decided in $S(I)$, then their decisions are the same.

The Termination property requires correct processes to eventually decide in all executions, regardless of the number processes that are correct. Thus, this formulation of this property commits us to so-called *wait-free* algorithms, where correct processes satisfy their liveness properties (in this case, to eventually decide) regardless of the progress made by other processes.

3.2. The Equality Negation problem. We now define the Equality Negation problem for $n \geq 2$ processes, $P_0, \ldots, P_{n-1}$. To understand the definition it is best to first consider the special case of two processes ($n = 2$). In this special case, each of

P_0 and P_1 has a private initial value, drawn from the set $\{0,1,2\}$ (instead of $\{0,1\}$ in Consensus). Each correct process must irrevocably decide either 0 or 1 so that the decisions of the processes are the same if and only if the initial values of the processes are different.

We now generalise the problem for arbitrary $n \geq 2$; the idea is to replace the role of P_1 by a *group* of $n-1$ processes $\mathcal{P}_1 = \{P_1, \ldots, P_{n-1}\}$. More specifically, each process has an initial value drawn from the set $\{0,1,2\}$. Additionally, we assume that all processes in $\mathcal{P}_1$ have the same initial value; if not, the problem imposes no constraints whatsoever on the behaviour of processes. Every process must eventually decide irrevocably either 0 or 1, so that no two processes in $\mathcal{P}_1$ decide different values; and if P_0 and any process in $\mathcal{P}_1$ decide, then their decisions are the same if and only if their initial values are different. Thus, an *Equality Negation algorithm* $\mathbf{A}_{en}$ *for processes* $P_0, \ldots, P_{n-1}$ satisfies the following properties. Each process in $\mathbf{A}_{en}$ has three initial states associated with input values 0, 1 and 2, respectively. As in a Consensus algorithm, each process has two disjoint sets of states associated with decisions 0 and 1, respectively. Since a decision for Equality Negation is also irrevocable, once a process has entered a state associated with decision $d \in \{0,1\}$, all states that it may subsequently enter must be associated with d as well. In addition, for any schedule S applicable to an initial configuration I of $\mathbf{A}_{en}$ in which all processes in $\mathcal{P}_1$ have the same initial value, the following three properties hold.

Termination: If S is infinite, then for every correct process in S, there is a prefix S' of S such that the process has decided in $S'(I)$.

Negation: If process P_0 and any process in $\mathcal{P}_1$ have decided in $S(I)$, then their decisions are the same if and only if their initial values are different.

Agreement: If any two processes in $\mathcal{P}_1$ have decided in $S(I)$, then their decisions are the same.

It is well-known that there is no Consensus algorithm for even two processes that uses only registers [3, 13]. In Appendix A we prove that the same is true for Equality Negation:

THEOREM 3.1. *There is no algorithm that solves Equality Negation for two processes and uses only registers.*

4. Specification of types negation and booster. We define two nondeterministic object types, named negation and booster, that will be used later to establish the result that no nontrivial wait-free hierarchy is robust. Both of these types have bounded nondeterminism; indeed, each has only finitely many states.

4.1. Type negation. The specification of negation appears in Figure 4.1.[3] To understand the specification, it is useful to keep in mind the following intuitive explanation of the type. This type supports only one kind of operation, called NEGATE, with two parameters $i \in \{0,1\}$ and $v \in \{0,1,2\}$. For reasons that will be clear later, we shall refer to i as the *process* parameter and to v as the *initial-value* parameter. A negation object will become "upset" if and only if two NEGATE operations with process parameter 0 are applied or two NEGATE operations with process parameter 1, but with different initial-value parameters, are applied. If the negation object is upset, it will arbitrarily return either 0 or 1 to any operation, and will remain upset forever. As long as the negation object is not upset, it responds to invocations

[3] In this figure, and throughout this paper, for any variable $v \in \{0,1\}$, $\overline{v}$ denotes the complement of v; i.e., $\overline{v} = 1 - v$.

Let $\mathcal{A}$ be the set of four-tuples of the form $\langle v_0, v_1, u_0, u_1\rangle$, where $v_0, v_1 \in \{\perp, 0, 1, 2\}$, $u_0, u_1 \in \{\perp, 0, 1\}$, and the following conditions apply:

- $v_0 = \perp$ if and only if $u_0 = \perp$;
- $v_1 = \perp$ if and only if $u_1 = \perp$;
- at least one of v_0, v_1 is not $\perp$; and
- if $v_0, v_1 \neq \perp$ then $v_0 = v_1$ if and only if $u_0 \neq u_1$.

Set of states: $\{\text{☺},\text{☹}\} \cup \mathcal{A}$.

Set of operations: $\{\ \textsc{negate}(i, v) : i \in \{0, 1\}, v \in \{0, 1, 2\}\ \}$.

Set of responses: $\{0, 1\}$.

```
NEGATE(i, v), i ∈ {0,1}, v ∈ {0,1,2}, return either 0 or 1
  case current_state
    ☹: return Choose({0,1})
    ☺: u ← Choose({0,1})
       if i = 0 then current_state ← ⟨v, ⊥, u, ⊥⟩
       else current_state ← ⟨⊥, v, ⊥, u⟩
       return u
    ⟨v0, v1, u0, u1⟩:
       if i = 0 then
          if v0 ≠ ⊥ then current_state ← ☹; return Choose({0,1})
          else (* v0 = ⊥ and, thus, v1, u1 ≠ ⊥ *)
             if v ≠ v1 then u ← u1
             else u ← ū1
             current_state ← ⟨v, v1, u, u1⟩; return u
       else (* i = 1 *)
          if v1 ≠ ⊥ then
             if v ≠ v1 then current_state ← ☹; return Choose({0,1})
             else return u1
          else (* v1 = ⊥ and, thus, v0, u0 ≠ ⊥ *)
             if v ≠ v0 then u ← u0
             else u ← ū0
             current_state ← ⟨v0, v, u0, u⟩; return u
```

FIG. 4.1. *Specification of type* negation.

with values that satisfy the following: All NEGATE operations with process parameter 1 receive the same response; any two NEGATE operations with different process parameters receive the same response if and only if their initial-value parameters are different. Thus, an object O of type negation can be used to solve Equality Negation for processes $P_0, \ldots, P_{n-1}$ in a direct way: P_0 applies $\textsc{negate}(0, v)$ to O, where v is P_0's initial value; each process in $\{P_1, \ldots, P_{n-1}\}$ applies $\textsc{negate}(1, v')$ to O, where v' is the process' initial value; each process decides the value returned by object O.

To achieve this behaviour, type negation has two distinguished states: the *fresh* state, denoted by ☺; and the *upset* state, denoted by ☹. The remaining states are of the form $\langle v_0, v_1, u_0, u_1\rangle$, where $v_0, v_1 \in \{\perp, 0, 1, 2\}$ and $u_0, u_1 \in \{\perp, 0, 1\}$. The

intended meaning of state $\langle v_0, v_1, u_0, u_1 \rangle$, assuming the object is initialised to the fresh state, is as follows: v_0 is the initial-value parameter of the first NEGATE operation with process parameter 0; u_0 is the value returned to that operation; v_1 and u_1 have corresponding interpretations for the first NEGATE operation with process parameter 1. By maintaining this information, the object knows all it needs in order to respond appropriately to the NEGATE operations applied to it.

The pseudocode in Figure 4.1 describes the state transition relation of negation. In addition to standard programming constructs, we use the function "Choose" which takes as a parameter a set V of integers and returns an arbitrary value in V. We also assume that the present state of the type is recorded in variable *current_state*.

4.2. Type booster. The specification of type booster is shown in Figure 4.2. To understand the specification, it is useful to keep in mind the following intuitive explanation of the type. There are two kinds of operations that can be applied to a booster object: ENROLL and REVEAL. An ENROLL operation requires a parameter $i \in \{0,1\}$ and arbitrarily returns a value in $\{0,1,2\}$. A REVEAL operation requires three parameters $i \in \{0,1\}$, $v \in \{0,1,2\}$ and $u \in \{0,1\}$, and returns a value in $\{0,1\}$; we shall refer to these three parameters as the *process*, *challenge* and *decision* parameters, respectively. A booster object will be "upset" if (and only if) any one of the following holds:

- Two ENROLL(0) operations are applied.
- For some $i \in \{0,1\}$ and $v \in \{0,1,2\}$, a REVEAL(i,v,u) operation is applied such that no ENROLL(i) operation has previously been applied or v is not the response of any such ENROLL(i) operation.
- A REVEAL(i_0, v_0, u_0) and a REVEAL(i_1, v_1, u_1) operation are applied such that one of the following is true: (a) $i_0 = i_1 = 0$, or (b) $i_0 = i_1 = 1$ and either $v_0 \neq v_1$ or $u_0 \neq u_1$, or (c) $i_0 \neq i_1$ and $v_0 = v_1 \Leftrightarrow u_0 = u_1$.

Once upset, the booster object remains upset forever. If not upset, the object maintains a secret: the parameter of the first ENROLL operation applied. The object will reveal this secret to every process that subsequently accesses it with a REVEAL operation. If upset, however, the object arbitrarily returns either 0 or 1 to every REVEAL operation.

The intended use of a booster object by processes $P_0, \ldots, P_{n-1}$ is as follows: The first thing process P_0 does to a booster object is to apply ENROLL(0) to "enroll" itself in the object. Similarly, the first thing any process in $\mathcal{P}_1 = \{P_1, \ldots, P_{n-1}\}$ does to a booster object is to apply ENROLL(1) to enroll its group in the object. In response to the ENROLL operation it applies, every process receives a value in $\{0,1,2\}$ from the booster object. This value is a "challenge" given by the object with which the process is required to solve Equality Negation. More specifically, process P_0 uses the challenge, say v_0, received from the booster object as its initial value to solve Equality Negation. The different processes in $\mathcal{P}_1$, however, may receive different challenges from the booster object. Thus, before embarking upon solving Equality Negation, processes in $\mathcal{P}_1$ must first choose a unique initial value, say v_1, from the set of challenges they received. After solving Equality Negation and deciding a value $u \in \{0,1\}$, process P_0 (respectively, every process in $\mathcal{P}_1$) applies the operation REVEAL$(0, v_0, u)$ (respectively, REVEAL$(1, v_1, u)$) to the booster object to obtain the secret.

To achieve the behaviour described above, type booster (like type negation) has two distinguished states: the *fresh* state, denoted ☺, and the *upset* state, denoted ☹. The remaining states are of the form $\langle V, v_0, v_1, u_0, u_1, d \rangle$, where $V \subseteq \{0,1,2\}$, $v_0, v_1 \in \{\perp, 0, 1, 2\}$, $u_0, u_1 \in \{\perp, 0, 1\}$, and $d \in \{0,1\}$. The intended meaning of state

Let $\mathcal{A}$ be the set of six-tuples of the form $\langle V, v_0, v_1, u_0, u_1, d\rangle$, where $V \subseteq \{0,1,2\}$, $v_0, v_1 \in \{\perp, 0, 1, 2\}$, $u_0, u_1 \in \{\perp, 0, 1\}$, $d \in \{0,1\}$, and the following conditions apply:

- if $d = 0$ then $v_0 \neq \perp$; if $d = 1$ then $V \neq \emptyset$;
- if $v_0 = \perp$ then $u_0 = \perp$;
- $v_1 = \perp$ if and only if $u_1 = \perp$, and if $v_1 \neq \perp$ then $v_1 \in V$; and
- if $v_0, v_1 \neq \perp$ then $v_0 = v_1$ if and only if $u_0 \neq u_1$.

Set of states: $\{\text{☺},\text{☹}\} \cup \mathcal{A}$.
Set of operations: $\{\,\textsc{enroll}(i) : i \in \{0,1\}\,\} \cup \{\,\textsc{reveal}(i,v,u) : i,u \in \{0,1\}, v \in \{0,1,2\}\,\}$.
Set of responses: $\{0,1,2\}$.

```
ENROLL(i), i ∈ {0,1}, return a value in {0,1,2}
    v ← Choose({0,1,2})
    case current_state
        ☹: no state change
        ☺: if i = 0 then current_state ← ⟨∅, v, ⊥, ⊥, ⊥, 0⟩
           else current_state ← ⟨{v}, ⊥, ⊥, ⊥, ⊥, 1⟩
        ⟨V, v0, v1, u0, u1, d⟩:
           if i = 0 then
              if v0 ≠ ⊥ then current_state ← ☹
              else current_state ← ⟨V, v, v1, u0, u1, d⟩
           else current_state ← ⟨V ∪ {v}, v0, v1, u0, u1, d⟩
    return v

REVEAL(i, v, u), i, u ∈ {0,1}, v ∈ {0,1,2}, return a value in {0,1}
    case current_state
        ☺: current_state ← ☹; return Choose({0,1})
        ☹: return Choose({0,1})
        ⟨V, v0, v1, u0, u1, d⟩:
           if i = 0 then
              if v = v0 and u0 = ⊥ and (u1 = ⊥ or (v = v1 ⇔ u ≠ u1)) then
                 current_state ← ⟨V, v0, v1, u, u1, d⟩; return d
              else current_state ← ☹; return Choose({0,1})
           else (* i = 1 *)
              if v ∈ V and ((u1 = ⊥ and (u0 = ⊥ or (v = v0 ⇔ u ≠ u0)))
                                or (v = v1 and u = u1)) then
                 current_state ← ⟨V, v0, v, u0, u, d⟩; return d
              else current_state ← ☹; return Choose({0,1})
```

FIG. 4.2. *Specification of type* booster.

$\langle V, v_0, v_1, u_0, u_1, d\rangle$ is as follows (assuming the object is initialised to the fresh state):

- d is the "secret" — the parameter the first ENROLL operation applied; i.e., whether P_0 or some process in $\mathcal{P}_1$ is the process that applied the first ENROLL. More accurately, d is the parameter of the first ENROLL operation.

- V is the set of challenges given to processes in $\mathcal{P}_1$, using one of which these processes must solve Equality Negation. More accurately, V is the set of values returned to ENROLL(1) operations.
- v_0 is the challenge with which P_0 must solve Equality Negation. More accurately, the value returned to the first ENROLL(0) operation.
- v_1 is the challenge chosen from V by processes in $\mathcal{P}_1$, with which these processes solve Equality Negation. More accurately, v_1 is the challenge parameter of the first REVEAL$(1,*,*)$ operation.[4]
- u_0 is the decision reached for Equality Negation by P_0. More accurately, u_0 is the decision parameter of the first REVEAL$(0,*,*)$ operation.
- u_1 is the decision reached for Equality Negation by the first process in $\mathcal{P}_1$ that tries to get the secret. More accurately, u_1 is the decision parameter of the first REVEAL$(1,*,*)$ operation.

With this information, the object knows all it needs in order to respond appropriately to operations.

5. The proof of nonrobustness. Later in the paper we shall prove that each of negation and booster has Consensus number one. Taking this for granted, in this section we prove our main result: in the context of nondeterministic types, no nontrivial wait-free hierarchy can be robust. The key to this proof is the following lemma.

LEMMA 5.1. *For any $n \geq 1$, if Consensus among n processes can be solved using only* negation *and* booster *objects and registers, then Consensus among $n+1$ processes can also be solved using only* negation *and* booster *objects and registers.*

Proof. Suppose that Consensus among n processes can be solved using only negation and booster objects and registers. Let $\mathbf{A}_c^1$ and $\mathbf{A}_c^2$ be algorithms that solve Consensus among n processes, $P_1, P_2, \ldots, P_n$, and use only negation and booster objects and registers. Without loss of generality, we may assume that $\mathbf{A}_c^1$ and $\mathbf{A}_c^2$ solve Consensus with initial values drawn from the set {0,1,2}.[5]

We shall show that the algorithm in Figure 5.1 uses $\mathbf{A}_c^1$ and $\mathbf{A}_c^2$ to solve Consensus among $n+1$ processes, $P_0, P_1, \ldots, P_n$. Following are some conventions used in the pseudocode in Figure 5.1 and other algorithms described later. We use capital letters for names of shared objects (including registers), and lower-case letters for names of local variables. Local variables with the same name that belong to different processes are distinct. The notation "Apply(P, op, O)" denotes the procedure by which process P applies operation op to object O; the procedure returns the response of the object to that operation. We use a special (but familiar) notation for the application of operations to an object R of type register: we place R on the left-hand side of an assignment statement to indicate a WRITE operation applied to R, and we use the name R in any other context to indicate a READ operation applied to R. If $\mathbf{A}$ is an Equality Negation or Consensus algorithm and P is a process of $\mathbf{A}$, the procedure Execute$(P, \mathbf{A}, k)$ causes the caller to execute the steps of process P in algorithm $\mathbf{A}$ using k as its initial value, and returns the decision of P in the simulated execution of that algorithm.

Before proving that the algorithm in Figure 5.1 does, in fact, solve Consensus, we informally explain how it works.

[4] Here, and in the rest of the paper, we use "$*$" to indicate a "don't care" entry.

[5] In our definition of Consensus (sometimes called *binary* Consensus for emphasis), the initial values of processes are drawn from the set {0,1}. It is well-known [16] that if a set of types that includes register can be used to solve (binary) Consensus, it can also be used to solve Consensus with initial values drawn from an arbitrary set.

Shared: all booster and negation objects and registers used in $\mathbf{A}_c^1$ and $\mathbf{A}_c^2$, each initialised as specified by $\mathbf{A}_c^1$ or $\mathbf{A}_c^2$

O_b: booster, initialised to ☺

O_n: negation, initialised to ☺

R_0, R_1: register, each initialised to $\perp$

Code for process P_0

1 $R_0 :=$ initial value of P_0
2 $v :=$ Apply$(P_0, \text{ENROLL}(0), O_b)$
3 $u :=$ Apply$(P_0, \text{NEGATE}(0, v), O_n)$
4 $d :=$ Apply$(P_0, \text{REVEAL}(0, v, u), O_b)$
5 **decide** R_d

Code for process P_k, $k \in \{1, \ldots, n\}$

1 $init :=$ initial value of P_k
2 $R_1 :=$ Execute$(P_k, \mathbf{A}_c^1, init)$
3 $w :=$ Apply$(P_k, \text{ENROLL}(1), O_b)$
4 $v :=$ Execute$(P_k, \mathbf{A}_c^2, w)$
5 $u :=$ Apply$(P_k, \text{NEGATE}(1, v), O_n)$
6 $d :=$ Apply$(P_k, \text{REVEAL}(1, v, u), O_b)$
7 **decide** R_d

FIG. 5.1. *A Consensus algorithm for $n+1$ processes using only objects of types* negation *and* booster *and registers.*

- P_0 first writes its initial value into register R_0. It then enrolls in the booster object O_b by applying ENROLL(0) and obtains a challenge, say v, with which it is required to solve Equality Negation. To do so, P_0 applies NEGATE$(0, v)$ to the negation object O_n. Let u be the value returned by O_n. By applying REVEAL$(0, v, u)$ to O_b, P_0 determines whether it or one of $\{P_1, \ldots, P_n\}$ was the first to enroll in O_b.
- Processes $P_1, \ldots, P_n$ first agree on one of their initial values; they use $\mathbf{A}_c^1$ for that purpose. They then write the agreed-upon initial value into register R_1 (multiple processes may write into R_1, but they all write the same value). Each P_k, $k \in \{1, \ldots, n\}$, then enrolls in O_b by applying ENROLL(1), and obtains a challenge. Since, object O_b may hand out different challenges to processes $P_1, \ldots, P_n$, these processes use the Consensus algorithm $\mathbf{A}_c^2$ to agree upon one of these challenges, say v. Each P_k, $k \in \{1, \ldots, n\}$, uses v as its initial value to solve Equality Negation. To do so, P_k applies NEGATE$(1, v)$ to the negation object O_n. Let u be the value returned by O_n. By applying REVEAL$(1, v, u)$ to O_b, P_k determines whether P_0 or one of the other processes was the first to enroll in O_b.

Since all processes agree on whether P_0 or one of the other processes was the first to enroll in O_b, they can agree on the initial value of some process: If P_0 was the first to enroll, all processes agree on the value they read from R_0; otherwise, they all agree on the value they read from R_1.

We now prove that the algorithm in Figure 5.1 solves Consensus. It is clear that the algorithm satisfies Termination, since both $\mathbf{A}_c^1$ and $\mathbf{A}_c^2$ do. Consider any execution of the algorithm. From the specification of booster and negation, it is easy to verify that the algorithm accesses O_b and O_n in such a way that these objects never become upset. Furthermore, if d is the parameter of the first ENROLL operation applied to O_b, then that ENROLL operation will cause O_b to enter a state of the form $\langle *, *, *, *, *, d \rangle$. The last component of the state is never changed, once it is set. Furthermore, any REVEAL operation applied to O_b that does not cause the object to become upset returns the last component of the state. Thus,

every REVEAL operation applied to O_b returns d, where d is the parameter of the first ENROLL operation applied to O_b. (2)

Next we claim that, for each $k \in \{0,1\}$, at most one value is written into register R_k, and that value is an initial value of some process. This is clearly true for $k = 0$, since only P_0 writes into R_0, and the value it writes there is its initial value. For $k = 1$, this follows from the Agreement and Validity properties of the Consensus algorithm $\mathbf{A}_c^1$ used by the processes in $\{P_1, \ldots, P_n\}$ to determine the value they each write into R_1.

Thus, to prove the algorithm in Figure 5.1 satisfies Validity, it suffices to show that if a process decides the value in register R_d, then some process has previously written a value into R_d. This follows from (2), since in the algorithm a process writes into R_k before applying ENROLL(k), for each $k \in \{0,1\}$.

It remains to show that the algorithm satisfies Agreement. Consider any execution in which two distinct processes P and P' have decided. By (2), P and P' decide the value in the same register R_d, where d is the parameter of the first ENROLL operation applied to O_b. By Validity, a process decides the value in R_d only if some process has written a value into R_d. As argued earlier, only one value is written into R_d in any given execution. Therefore, P and P' decide the same value, the one written in R_d. Therefore, the algorithm satisfies Agreement.

Thus the algorithm in Figure 5.1 solves Consensus for $n+1$ processes. It uses only registers, booster and negation objects because, in addition to the (registers, booster and negation) objects used by $\mathbf{A}_c^1$ and $\mathbf{A}_c^2$, the algorithm uses only two registers, one booster object and one negation object. □

THEOREM 5.2. *For any $n \geq 1$, Consensus among n processes can be solved using only registers and objects of types* negation *and* booster.

Proof. By induction on n. The basis, $n = 1$ is trivial (because Consensus for one process is trivial). The induction step is Lemma 5.1. □

COROLLARY 5.3. *There is no nontrivial wait-free hierarchy that is robust.*

Proof. Suppose h is a nontrivial wait-free hierarchy of types. We shall prove that h is not robust. By Theorems 7.10 and 8.15 (see Sections 7 and 8, respectively) the Consensus number of negation and booster is 1. Since h is wait-free, $h(\text{negation}) \leq 1$ and $h(\text{booster}) \leq 1$, and thus $h(\text{negation}) = h(\text{booster}) = 1$. Since h is nontrivial, there is some level $n > 1$ and some type T such that $h(T) = n$. Let $\mathcal{B} = \{\text{negation}, \text{booster}\}$. By Theorem 5.2, there is an algorithm that solves Consensus among n processes and uses only objects that belong to types in $\mathcal{B}$ and registers. Since $h(\text{negation}) = h(\text{booster}) = 1$ and $n > 1$, $\mathcal{B}$ contains no type that is at level n or higher. Thus, h is not robust, as wanted. □

6. Some technical preliminaries. In this section we define some concepts and prove some technical results that are needed in our proofs that each of the types negation and booster has Consensus number one.

6.1. Computation trees. Let $\mathbf{A}$ be an algorithm, and I be an initial configuration of $\mathbf{A}$. Consider a tree $\mathcal{T}$ whose nodes are finite schedules of $\mathbf{A}$ that are applicable to I (not necessarily all of them), so that the root of $\mathcal{T}$ is the empty schedule, denoted $S_\perp$, and there is an edge from node S to node S' only if $S' = S \cdot e$, for some step e. We say that $\mathcal{T}$ is a *computation tree of* $\mathbf{A}$ *from* I if for any node S of $\mathcal{T}$ and any process P of $\mathbf{A}$, S has at least one child $S \cdot e$, where e is a step of P. In the sequel, we shall not distinguish between a node in a computation tree and the (finite) schedule corresponding to the node. Similarly, we shall not distinguish between a (finite or

infinite) path in a computation tree and the (finite or infinite) schedule corresponding to the path.

Informally, a computation tree of **A** from I represents executions of **A** starting from I for all possible interleavings of the processes. A node S represents a point in some execution; at that point the algorithm is in configuration $S(I)$. It is easy to see that if all object types used by **A** are deterministic, there is a unique computation tree of **A** from I. If, however, some object types are nondeterministic, there may be more than one computation tree of **A** from I.

A computation tree of **A** from I is *full* if its set of nodes is the entire set of finite schedules of **A** that are applicable to I. The full computation tree of **A** from I is unique, regardless of whether the object types used by **A** are deterministic. If **A** uses nondeterministic object types, we can think of the different computation trees of **A** from I as obtained from the full computation tree of **A** from I by (repeatedly) applying the following pruning rule: If there are multiple edges out of a node S corresponding to steps of a single process P (because of the nondeterministic responses of the object accessed by P in the operation pending in $S(I)$), then some, but not all, of these edges may be pruned.

We use the following notation and terminology for computation trees. Let $\mathcal{T}$ be a computation tree of **A** from an initial configuration I, let S be a node in $\mathcal{T}$, and let P be a process of **A**. The subtree of $\mathcal{T}$ that is rooted at S is denoted *subtree*$(S, \mathcal{T})$, and we say that it is *full* if it is equal to *subtree*$(S, \mathcal{F})$, where $\mathcal{F}$ is the full computation tree of **A** from I. The set of nodes $\{ S \cdot e : S \cdot e \text{ is a node in } \mathcal{T} \text{ and } e \text{ is a step of } P \}$ is denoted *children*$(P, S, \mathcal{T})$; these are the nodes reached from S by a single step of process P. We say that S_1 is a *P-sibling* of S_2 in $\mathcal{T}$ if there is a node S in $\mathcal{T}$ so that $S_1, S_2 \in$ *children*$(P, S, \mathcal{T})$.

LEMMA 6.1. *Let* **A** *be an algorithm, and let $\mathcal{T}$ and $\mathcal{T}'$ be any computation trees of* **A** *from (not necessarily distinct) initial configurations I and I', respectively. Consider any nodes S_0 and $S_0 \cdot S$ in $\mathcal{T}$ and node S_1 in $\mathcal{T}'$ such that subtree$(S_1, \mathcal{T}')$ is full and every process that takes a step in S and every object is in the same state in $S_0(I)$ as in $S_1(I')$. Then, $S_1 \cdot S$ is a node in $\mathcal{T}'$, and every process that takes a step in S and every object is in the same state in $S_0 \cdot S(I)$ as in $S_1 \cdot S(I')$.*

Proof. Let S^i be the schedule consisting of the first i steps of S, for all $0 \leq i \leq |S|$. Using the fact that *subtree*$(S_1, \mathcal{T}')$ is full and every process that takes a step in S and every object is in the same state in $S_0(I)$ as in $S_1(I')$, a straightforward induction on i proves that S^i is applicable to $S_1(I')$, and every process that takes a step in S and every object is in the same state in $S_0 \cdot S^i(I)$ as in $S_1 \cdot S^i(I')$. The lemma follows from this, since $S = S^\ell$, where $\ell = |S|$. □

6.2. Valence of nodes. Let **A** be a Consensus or Equality Negation algorithm for processes $P_0, P_1, \ldots, P_{n-1}$, where $n \geq 2$. Let I be an initial configuration of **A**, $\mathcal{T}$ be a computation tree of **A** from I, and S be a node in $\mathcal{T}$. For any $v \in \{0, 1\}$, we say that S is *v-valent in $\mathcal{T}$* if there is no descendent S' of S in $\mathcal{T}$ such that process P_0 decides $\overline{v}$ in $S'(I)$. S is *univalent in $\mathcal{T}$* if it is v-valent in $\mathcal{T}$, for some $v \in \{0, 1\}$. S is *bivalent in $\mathcal{T}$* if there are descendents S_0 and S_1 of S in $\mathcal{T}$ so that process P_0 decides 0 in $S_0(I)$ and it decides 1 in $S_1(I)$. (Note that in this definition it is the same process, P_0, whose decisions are 0 and 1 in $S_0(I)$ and $S_1(I)$, respectively.) Informally, a univalent node represents a point in an execution of **A** where the outcome (of Consensus or Equality Negation) has been "sealed" — even if no process has actually decided yet. A bivalent node represents a point in an execution where both outcomes are still possible.

The *valence* of a node S in $\mathcal{T}$ refers to whether S is bivalent or univalent in $\mathcal{T}$ and, in the latter case, whether it is 0-valent or 1-valent. As noted in Section 6.1, if **A** uses nondeterministic types, there may be many computation trees of **A** from I. Therefore, a node S that appears in two different computation trees may have a different valence in one than in the other, because the valence of S in a computation tree $\mathcal{T}$ depends on the subset of S's descendents in the *full* computation tree that are actually present in $\mathcal{T}$.

Let S_0 and S_1 be two univalent nodes in $\mathcal{T}$. We say that S_0 and S_1 *have the same valence* in $\mathcal{T}$ if they are both 0-valent or both 1-valent in $\mathcal{T}$. We say that they *have opposite valence* in $\mathcal{T}$ if one is 0-valent and the other is 1-valent in $\mathcal{T}$.

The notions of v-valence, univalence and bivalence for *configurations* of *Consensus* algorithms were originally defined in [5], and subsequently used extensively in the literature. Although the above definitions are very much in the spirit of those in [5], there are some minor differences. We now explain the reasons for these differences. First, we define valence with respect to nodes of computation trees, rather than configurations. This is because some of our results (in particular, those in Section 7) are more conveniently proved using computation trees. A second, and more important, difference is the fact that we now wish to have a more general definition that encompasses both Consensus and Equality Negation algorithms; in the past, valence was only defined in the context of Consensus algorithms. The need for this greater generality underlies the other difference of our definition relative to the earlier ones: In the "classical" definition, a node is v-valent if there is no descendent of it in which *any* process (not P_0 in particular) decides $\overline{v}$. An analogous comment applies to the definition of bivalence. In the case of Consensus, there is no difference between the two definitions, since the Agreement property requires that any two processes that decide, decide the same value. In the case of Equality Negation, however, if the initial values of all processes are the same, then P_0 is supposed to decide a different value than the other processes. Thus, in this case, we should specify whether it is the decision of P_0 or the decision of the other processes that is taken to be the "outcome" of the computation. It is not important *which* of the two is chosen. In our definition we arbitrarily chose P_0.

6.3. Properties of computation trees. In this section we prove several properties of computation trees of any Consensus or Equality Negation algorithm for two processes. These properties are used in Sections 7 and 8. For the rest of this section, let **A** be a Consensus or Equality Negation algorithm for processes P_0 and P_1, and I be any initial configuration of **A**.

LEMMA 6.2. *Let $\mathcal{T}$ be any computation tree of* **A** *from initial configuration I, and let S be any node in $\mathcal{T}$. For each $k \in \{0,1\}$, there exists a solo schedule S_k of process P_k such that $S \cdot S_k$ is a node in $\mathcal{T}$ and P_k has decided in $S \cdot S_k(I)$.*

Proof. Immediate from the Termination requirement (of Consensus or Equality Negation), and the definition of computation tree. □

By Lemma 6.2, every node S in $\mathcal{T}$ has a descendent S' such that process P_0 has decided in $S'(I)$. From the definition of valence it follows that S has exactly one of the three possible attributes: 0-valent, 1-valent or bivalent. Therefore, the valence of each node in $\mathcal{T}$ is well-defined.

LEMMA 6.3. *Let $\mathcal{T}$ be any computation tree of* **A** *from initial configuration I. Let S_0 and S_1 be any univalent nodes in $\mathcal{T}$ such that some process has decided the same value in $S_0(I)$ as in $S_1(I)$. Then, S_0 and S_1 have the same valence in $\mathcal{T}$.*

Proof. Observe that if a process decides a value, say v, in a node S of $\mathcal{T}$, then

the initial configuration I and v uniquely determine the value w that P_0 can decide in any descendent of S in $\mathcal{T}$. (If $\mathbf{A}$ is a Consensus algorithm, then $w = v$ regardless of the initial configuration. If $\mathbf{A}$ is an Equality Negation algorithm, then $w = v$ (respectively, $w = \overline{v}$) if the initial values of P_0 and P_1 in I are different (respectively, the same).)

Let P_k, for some $k \in \{0, 1\}$, be the process that has decided the same value, say v, in $S_0(I)$ as in $S_1(I)$. Let S_0' and S_1' be descendents of S_0 and S_1, respectively, such that P_0 has decided in $S_0'(I)$ and $S_1'(I)$ (S_0' and S_1' exist by Lemma 6.2). By the preceding observation, P_0 decides the same value, say w, in $S_0'(I)$ as in $S_1'(I)$. Recalling that S_0 and S_1 are both univalent, from the definition of w-valent node it follows that S_0 and S_1 are both w-valent in $\mathcal{T}$. Thus, S_0 and S_1 have the same valence in $\mathcal{T}$. □

A computation tree $\mathcal{T}$ of $\mathbf{A}$ is *1-full* if every subtree rooted at a 1-valent node in $\mathcal{T}$ is full; in particular, every full computation tree of $\mathbf{A}$ is 1-full.

LEMMA 6.4. *Let $\mathcal{T}$ be any 1-full computation tree of $\mathbf{A}$ from initial configuration I. Let S_0 and S_1 be any univalent nodes in $\mathcal{T}$ such that the state of some process and each object is the same in $S_0(I)$ as in $S_1(I)$. Then, S_0 and S_1 have the same valence in $\mathcal{T}$.*

Proof. Suppose, for contradiction, that S_0 and S_1 have opposite valence. Without loss of generality, assume that S_0 is 0-valent and S_1 is 1-valent in $\mathcal{T}$. Let P be the process whose state in $S_0(I)$ is the same as in $S_1(I)$. By Lemma 6.2 (applied to node S_0 in $\mathcal{T}$), there exists a solo schedule S of P such that $S_0 \cdot S$ is a node in $\mathcal{T}$ and P has decided in $S_0 \cdot S(I)$. Since $\mathcal{T}$ is 1-full and S_1 is 1-valent in $\mathcal{T}$, it follows that $subtree(S_1, \mathcal{T})$ is full. By hypothesis, the state of process P and each object is the same in $S_0(I)$ as in $S_1(I)$. Since S is a solo schedule of P, by Lemma 6.1 (applied for $I' = I$), $S_1 \cdot S$ is a node in $\mathcal{T}$ and P has the same state in $S_0 \cdot S(I)$ as in $S_1 \cdot S(I)$. Thus, P decides the same value in $S_0 \cdot S(I)$ as in $S_1 \cdot S(I)$. By Lemma 6.3, $S_0 \cdot S$ and $S_1 \cdot S$ have the same valence in $\mathcal{T}$. This contradicts our assumption that S_0 and S_1 have opposite valence in $\mathcal{T}$. □

Let $\mathcal{T}$ be any computation tree of $\mathbf{A}$ from initial configuration I, let S be any node in $\mathcal{T}$, and let e_0 and e_1 be any steps of $\mathbf{A}$. We say *e_0 and e_1 commute at S in $\mathcal{T}$* if both $S \cdot e_0 \cdot e_1$ and $S \cdot e_1 \cdot e_0$ are nodes in $\mathcal{T}$ and the state of each process and each object is the same in $S \cdot e_0 \cdot e_1(I)$ as in $S \cdot e_1 \cdot e_0(I)$. We say that *e_k overwrites $e_{\overline{k}}$ at S in $\mathcal{T}$*, for some $k \in \{0, 1\}$, if both $S \cdot e_k$ and $S \cdot e_{\overline{k}} \cdot e_k$ are nodes in $\mathcal{T}$ and the state of P_k and each object is the same in $S \cdot e_k(I)$ as in $S \cdot e_{\overline{k}} \cdot e_k(I)$.

LEMMA 6.5. *Let $\mathcal{T}$ be any 1-full computation tree of $\mathbf{A}$ from initial configuration I. Let S be any node in $\mathcal{T}$, and let e_0 and e_1 be any steps of $\mathbf{A}$ such that $S \cdot e_0$ and $S \cdot e_1$ are univalent nodes of opposite valence in $\mathcal{T}$. Then,*

(a) e_0 and e_1 do not commute at S in $\mathcal{T}$; and

(b) neither one of e_0 and e_1 overwrites the other at S in $\mathcal{T}$.

Proof. Without loss of generality, assume that $S \cdot e_0$ is 0-valent and $S \cdot e_1$ is 1-valent in $\mathcal{T}$.

(a) Suppose, for contradiction, that e_0 and e_1 commute at S in $\mathcal{T}$. Then, by definition, $S \cdot e_0 \cdot e_1$ and $S \cdot e_1 \cdot e_0$ are nodes in $\mathcal{T}$ and the state of each process and object is the same in $S \cdot e_0 \cdot e_1(I)$ as in $S \cdot e_1 \cdot e_0(I)$. By Lemma 6.4, $S \cdot e_0 \cdot e_1$ and $S \cdot e_1 \cdot e_0$ have the same valence in $\mathcal{T}$. This contradicts the fact that $S \cdot e_0 \cdot e_1$ and $S \cdot e_1 \cdot e_0$ have opposite valence in $\mathcal{T}$ (because $S \cdot e_0$ and $S \cdot e_1$ do).

(b) Suppose, for contradiction, that e_k overwrites $e_{\overline{k}}$ at S in $\mathcal{T}$, for some $k \in \{0, 1\}$. Then, by definition, $S \cdot e_k$ and $S \cdot e_{\overline{k}} \cdot e_k$ are nodes in $\mathcal{T}$ and the state of P_k and each

object is the same in $S \cdot e_k(I)$ as in $S \cdot e_{\overline{k}} \cdot e_k(I)$. By Lemma 6.4, $S \cdot e_k$ and $S \cdot e_{\overline{k}} \cdot e_k$ have the same valence in $\mathcal{T}$. This contradicts the fact that $S \cdot e_k$ is k-valent and $S \cdot e_{\overline{k}} \cdot e_k$ is $\overline{k}$-valent in $\mathcal{T}$ (because $S \cdot e_{\overline{k}}$ is). □

LEMMA 6.6. *Let $\mathcal{F}$ be any full computation tree of $\mathbf{A}$. Let S be any node in $\mathcal{F}$, and let e_0 and e_1 be any steps of processes P_0 and P_1, respectively, such that $S \cdot e_0$ and $S \cdot e_1$ are univalent nodes of opposite valence in $\mathcal{F}$. Then, e_0 and e_1 access the same object.*

Proof. Suppose, for contradiction, that e_0 and e_1 access different objects. Then, e_1 is applicable to $S \cdot e_0(I)$ and e_0 is applicable to $S \cdot e_1(I)$. Since $\mathcal{F}$ is a full computation tree, $S \cdot e_0 \cdot e_1$ and $S \cdot e_1 \cdot e_0$ are nodes in $\mathcal{F}$. Furthermore, each process and object has the same state in $S \cdot e_0 \cdot e_1(I)$ as in $S \cdot e_1 \cdot e_0(I)$. Thus, e_0 and e_1 commute at S in $\mathcal{F}$. This contradicts Lemma 6.5(a) (because $\mathcal{F}$ is full and hence 1-full). □

Let $\mathcal{T}$ be any computation tree of $\mathbf{A}$, and S be any node in $\mathcal{T}$. We say that S is a *critical node* in $\mathcal{T}$ if S is bivalent and all of its children are univalent in $\mathcal{T}$.

LEMMA 6.7. *Let $\mathcal{T}$ be any computation tree of $\mathbf{A}$ with a bivalent root. Then, $\mathcal{T}$ has a critical node.*

Proof. Suppose, for contradiction, that $\mathcal{T}$ has no critical node. Then, every bivalent node in $\mathcal{T}$ has at least one child that is also bivalent in $\mathcal{T}$. Thus, there is an *infinite* path S of $\mathcal{T}$ that starts at the (bivalent) root of $\mathcal{T}$ and consists entirely of bivalent nodes. Consider any node S' on this path; since S' is bivalent, no process has decided in $S'(I)$. Thus, there is an infinite schedule S of $\mathbf{A}$ that is applicable to I such that, for any prefix S' of S, no process has decided in $S'(I)$. Since S is infinite, at least one process takes infinitely many steps in S and therefore is correct in S. As we have just shown, however, in every prefix S' of S, no process — and, in particular, no correct process in S — decides in $S'(I)$. This contradicts the fact that $\mathbf{A}$ is a Consensus algorithm or an Equality Negation algorithm (in particular, it contradicts the Termination property of $\mathbf{A}$). □

7. Type negation has Consensus number one. In this section we prove that type negation has Consensus number one — i.e., that Consensus for two processes cannot be solved using only negation objects and registers.

7.1. The bivalence argument and some limitations. There is a standard technique for proving that there is no Consensus algorithm for n processes using a certain set of object types, known as "the bivalence argument". This technique goes back to the ground-breaking result of Fischer *et al.* [5] for the asynchronous message-passing model, and was further adapted by others, e.g., [3, 6, 13], for the asynchronous read/write-memory model. In this subsection, we sketch the main ingredients of this technique for two processes, and call attention to a difficulty that arises when it is applied to nondeterministic object types.

The bivalence argument employs proof by contradiction. It begins by assuming that there is a Consensus algorithm $\mathbf{A}_c$ for processes P_0 and P_1 that uses only objects of some given types. Consider the full computation tree $\mathcal{F}$ of this algorithm from an initial configuration I in which the two processes have different initial values. The root of $\mathcal{F}$ is bivalent because, by Validity, in a solo schedule a process can only decide its own initial value. Thus, by Lemma 6.7, $\mathcal{F}$ has a critical node S. By definition, some of the (univalent) children of the critical node S must be 0-valent and some must be 1-valent in $\mathcal{F}$. If the object types used by $\mathbf{A}_c$ are deterministic, then S has exactly two children in $\mathcal{F}$, say $S \cdot e_0$ and $S \cdot e_1$, where e_k is a step of process P_k, for $k \in \{0, 1\}$. Thus, in this case, we can say that *the outcome of Consensus after S is determined by which one of the two processes takes the next step.* By Lemma 6.6, e_0

For $k, v \in \{0,1\}$, $e_k^v = (P_k, \text{NEGATE}(k,k), O, v)$.

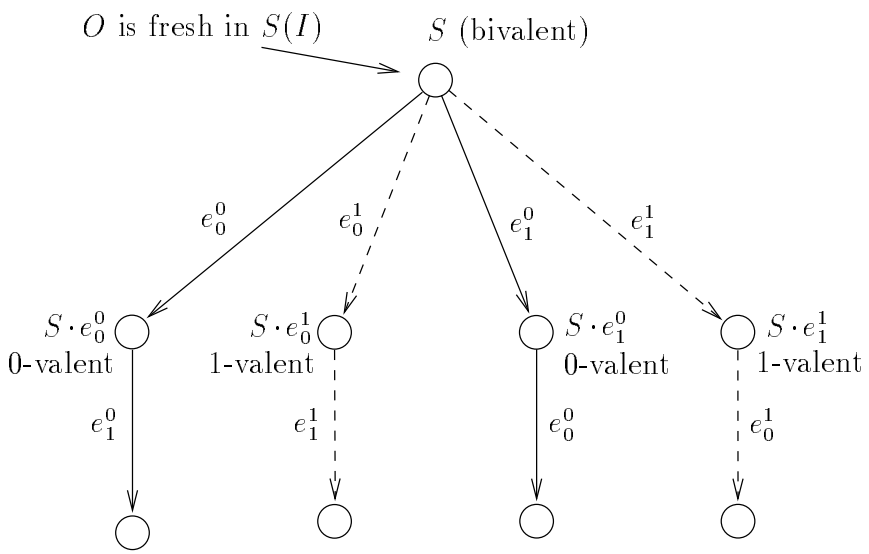

FIG. 7.1. *A critical node in* $\mathcal{F}$.

and e_1 both access the same object. Finally, by a case analysis involving the types of operations applied in steps e_0 and e_1, the following is shown: There exist schedules S_0 and S_1 such that (i) both $S \cdot e_0 \cdot S_0$ and $S \cdot e_1 \cdot S_1$ are nodes in $\mathcal{F}$, and (ii) some process has decided the same value in $S \cdot e_0 \cdot S_0(I)$ and $S \cdot e_1 \cdot S_1(I)$. This contradicts the fact that the outcome of Consensus is determined by which of the two processes takes the next step after S.

This type of argument runs into difficulties when applied to algorithms that use *nondeterministic* types. Specifically, we can no longer assert that at the critical node S, the outcome of Consensus is determined by which of the two processes takes the next step. Since the object types are nondeterministic, it is possible that *each* of the two processes can lead to both a 0-valent node and a 1-valent node, depending on the response of the object accessed.

To appreciate this point in more concrete terms, suppose that the Consensus algorithm $\mathbf{A}_c$ uses negation objects and that the critical node S is such that, in configuration $S(I)$, each process P_k has a NEGATE(k,k) operation pending to a negation object O in the fresh state. Recall that a NEGATE operation applied to a fresh negation object can return either 0 or 1. Suppose, further, that each process P_k decides the value returned by the NEGATE(k,k) operation (see Figure 7.1). Now the step of process P_k that causes it to decide v, namely e_k^v, is not applicable after the other process, $P_{\overline{k}}$, has applied the step that causes it to decide the opposite value, $\overline{v}$, for all $k \in \{0,1\}$ and all $v \in \{0,1\}$. This means that the we cannot derive a contradiction in the style of the bivalence argument.

Despite this, it turns out that the negation type is not powerful enough to solve Consensus between two processes. To prove this, we use a modified version of the

bivalence argument, based on a restricted computation tree. This is described in the following subsection.

7.2. The pruned computation tree. In the remainder of Section 7, we assume, for contradiction, that $\mathbf{A}_c$ is a Consensus algorithm for processes P_0 and P_1 that uses only negation objects and registers. Let I be an initial configuration of $\mathbf{A}_c$ in which P_0 and P_1 have different initial values, $\mathcal{T}$ be any computation tree of $\mathbf{A}_c$ from I, and $\mathcal{F}$ be the full computation tree of $\mathbf{A}_c$ from I. By the definition of type negation and the definition of computation tree, it is easy to see that $1 \leq |children(P, S, \mathcal{T})| \leq 2$, for every node S in $\mathcal{T}$ and each process P.

Consider a nonroot node S in $\mathcal{T}$, and let P be the process that takes the last step in S. We say that S is an *avoidable* 1-*valent node* in $\mathcal{T}$ if S is 1-valent in $\mathcal{F}$ and has a P-sibling S' in $\mathcal{F}$ such that S' is not 1-valent in $\mathcal{F}$. Intuitively, an avoidable 1-valent node S in $\mathcal{F}$ is reached from its (bivalent) parent, $\hat{S}$, by a step of some process P; that step seals the outcome of Consensus to be 1. Furthermore, S has a P-sibling S' that is not 1-valent; this means that the object accessed by the operation of P that is pending in $\hat{S}(I)$, *could* have returned a value that would leave open the possibility of the outcome of Consensus being 0. In other words, starting from configuration $\hat{S}(I)$, outcome 1 could be avoided by a fortuitous choice of the value returned by the object accessed by the operation of process P that is pending in $\hat{S}(I)$. As a concrete illustration, the computation tree shown in Figure 7.1 contains two avoidable 1-valent nodes, namely $S \cdot e_0^1$ and $S \cdot e_1^1$.

The *pruned computation tree of* $\mathbf{A}_c$ *from* I, denoted $\mathcal{L}$, is the tree obtained from the *full* computation tree $\mathcal{F}$ of $\mathbf{A}_c$ from I by pruning all its avoidable 1-valent nodes — i.e., removing them and their descendents. $\mathcal{L}$ is, in fact, a computation tree: If a node S in $children(P, \hat{S}, \mathcal{F})$ is pruned, then some P-sibling of S is not pruned. In other words, $\hat{S}$ has at least one child $\hat{S} \cdot e$ in $\mathcal{L}$ where e is a step of P. Thus, the pruning of S will not inhibit P from taking the next step after $\hat{S}$. Intuitively, pruning the avoidable 1-valent nodes from the full computation tree amounts to restricting the nondeterministic behaviour of objects: Whenever an object has two responses to an operation, one of which forecloses the possibility of outcome 0 while the other leaves the possibility of outcome 0 open, we restrict the behaviour of the object by choosing the latter response.

The following lemmata establish some basic properties of the pruned computation tree $\mathcal{L}$.

LEMMA 7.1. *The root of* $\mathcal{L}$ *is bivalent.*

Proof. Recall that $\mathcal{L}$ is a computation tree of $\mathbf{A}_c$ from initial configuration I in which the two processes have different initial values. Without loss of generality, assume that P_0 has initial value 0 and P_1 has initial value 1 in I. By Lemma 6.2 (applied to the root, $S_\perp$, of $\mathcal{L}$), for each $k \in \{0, 1\}$, there exists a solo schedule S_k of P_k such that $S_\perp \cdot S_k = S_k$ is a node in $\mathcal{L}$ and P_k has decided in $S_k(I)$. Since P_k has initial value k in I and S_k is a solo schedule of P_k, by Validity, P_k must decide k in $S_k(I)$. By Lemma 6.2 (applied to node S_1 of $\mathcal{L}$), there is a solo schedule S_0' of P_0 such that $S_1 \cdot S_0'$ is a node in $\mathcal{L}$ and P_0 has decided in $S_1 \cdot S_0'(I)$. Since P_1 has decided 1 in $S_1(I)$ and the decision is irrevocable, P_1 has decided 1 in $S_1 \cdot S_0'(I)$. By Agreement, P_0 has also decided 1 in $S_1 \cdot S_0'(I)$. Thus, the root of $\mathcal{L}$ has two descendents, S_0 and $S_1 \cdot S_0'$ so that P_0 has decided 0 and 1, respectively, in $S_0(I)$ and $S_1 \cdot S_0'(I)$. Thus, the root of $\mathcal{L}$ is bivalent. □

LEMMA 7.2. *Let* S *be any node in the full computation tree* $\mathcal{F}$*. Then*

(a) S *is not a node in* $\mathcal{L}$ *if and only if* S *is a descendent of an avoidable* 1-*valent*

node in $\mathcal{F}$. (By convention, a node is a descendent of itself.)

(b) If S is 1-valent in $\mathcal{F}$ and is also in $\mathcal{L}$, then S is 1-valent in $\mathcal{L}$.

(c) If S is 0-valent in $\mathcal{F}$, then it is in $\mathcal{L}$ and is 0-valent in $\mathcal{L}$.

(d) If S is in $\mathcal{L}$ and $children(P, S, \mathcal{F}) = \{S'\}$, then S' is in $\mathcal{L}$.

Proof. Immediate consequences of the definition of $\mathcal{L}$. □

LEMMA 7.3. *$\mathcal{L}$ is 1-full.*

Proof. We must show that every subtree rooted at a 1-valent node in $\mathcal{L}$ is full. Consider any 1-valent node S in $\mathcal{L}$. S cannot have any 0-valent descendents in $\mathcal{F}$: by Lemma 7.2(c), any such descendents would exist in $\mathcal{L}$ and be 0-valent in $\mathcal{L}$, contradicting that S is 1-valent in $\mathcal{L}$. Thus, S is 1-valent in $\mathcal{F}$. Consequently, all nodes in $subtree(S, \mathcal{F})$ are 1-valent, and therefore none of them can be an avoidable 1-valent node in $\mathcal{F}$. Hence, by Lemma 7.2(a), all nodes in $subtree(S, \mathcal{F})$ are in $\mathcal{L}$, proving that $subtree(S, \mathcal{L}) = subtree(S, \mathcal{F})$. Therefore, $subtree(S, \mathcal{L})$ is full. □

LEMMA 7.4. *Let S be any node in $\mathcal{L}$ and P be any process such that $|children(P, S, \mathcal{F})| = 2$, each node in $children(P, S, \mathcal{L})$ is univalent and one of them is 1-valent in $\mathcal{L}$. Then $children(P, S, \mathcal{L}) = children(P, S, \mathcal{F})$ and both nodes in $children(P, S, \mathcal{L})$ are 1-valent in $\mathcal{L}$.*

Proof. Let $children(P, S, \mathcal{F}) = \{S', S''\}$. By the hypothesis of the lemma we may assume, without loss of generality, that $S' \in children(P, S, \mathcal{L})$ and S' is 1-valent in $\mathcal{L}$. It remains to show that S'' is also in $\mathcal{L}$ and is 1-valent in $\mathcal{L}$.

By Lemma 7.3, $\mathcal{L}$ is 1-full. Since S' is 1-valent in $\mathcal{L}$, $subtree(S', \mathcal{L}) = subtree(S', \mathcal{F})$ and thus S' is 1-valent in $\mathcal{F}$. Since S' is in $\mathcal{L}$, by Lemma 7.2(a), S' is not a proper descendent of an avoidable 1-valent node in $\mathcal{F}$. Therefore, as S'' is a P-sibling of S' in $\mathcal{F}$, S'' is not a proper descendent of an avoidable 1-valent node in $\mathcal{F}$ either. Also, S'' is not an avoidable 1-valent node in $\mathcal{F}$, because its only P-sibling in $\mathcal{F}$, namely S', is a 1-valent node in $\mathcal{F}$. By Lemma 7.2(a), S'' is in $\mathcal{L}$. Furthermore, S'' must be a 1-valent node in $\mathcal{F}$. If not, then by definition S' would be an avoidable 1-valent node in $\mathcal{F}$ and, by Lemma 7.2(a), S' would not be in $\mathcal{L}$ — a contradiction. Since S'' is 1-valent in $\mathcal{F}$ and is also in $\mathcal{L}$, by Lemma 7.2(b), S'' is 1-valent in $\mathcal{L}$. □

7.3. The impossibility result. The proof that no algorithm can solve Consensus between two processes using only negation objects and registers is a bivalence argument based on the pruned computation tree $\mathcal{L}$. The only remaining tricky point is to ensure that the computation tree nodes that exhibit the contradictory behaviour required by the bivalence argument have not been pruned away!

By Lemmata 6.7 and 7.1, $\mathcal{L}$ has a critical node E. By definition of a critical node, there are steps e_0 and e_1 of processes P_0 and P_1, respectively, such that $E \cdot e_0$ and $E \cdot e_1$ are univalent nodes of opposite valence in $\mathcal{L}$. Without loss of generality, suppose that $E \cdot e_0$ is 0-valent and $E \cdot e_1$ is 1-valent in $\mathcal{L}$. (Otherwise, we rename e_0 and e_1 and apply the same argument.) To derive a contradiction, we first show that e_0 and e_1 access the same object (Lemma 7.5). Then, we prove that the object accessed in e_0 and e_1 cannot be a register (Lemma 7.6). Finally, we prove that the object cannot be a negation object (Corollary 7.8 and Lemma 7.9).

LEMMA 7.5. *Steps e_0 and e_1 access the same object.*

Proof. Suppose, for contradiction, that e_0 and e_1 access, respectively, objects O_0 and O_1, where $O_0 \neq O_1$. Then, e_1 is applicable to $E \cdot e_0(I)$ and e_0 is applicable to $E \cdot e_1(I)$ and, in fact, $E \cdot e_0 \cdot e_1(I) = E \cdot e_1 \cdot e_0(I)$. Thus $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are nodes in $\mathcal{F}$. Since $\mathcal{L}$ is 1-full (by Lemma 7.3) and $E \cdot e_1$ is 1-valent in $\mathcal{L}$, $subtree(E \cdot e_1, \mathcal{L}) = subtree(E \cdot e_1, \mathcal{F})$. Since $E \cdot e_1 \cdot e_0$ is in $\mathcal{F}$, it follows that it is also in $\mathcal{L}$. We shall prove that $E \cdot e_0 \cdot e_1$ is also a node in $\mathcal{L}$. This fact implies that e_0 and e_1 commute at E in

$\mathcal{L}$, contrary to Lemma 6.5(a).

To show that $E \cdot e_0 \cdot e_1$ is in $\mathcal{L}$, we consider two cases, depending on whether $children(P_1, E, \mathcal{F})$ has one or two nodes.

CASE 1. $|children(P_1, E, \mathcal{F})| = 1$: Then that unique child has to be $E \cdot e_1$ (since $E \cdot e_1$ is a node in $\mathcal{F}$). Since $O_0 \neq O_1$, each of P_1 and O_1 is in the same state in $E(I)$ as in $E \cdot e_0(I)$. It follows that e_1 is the only step of P_1 that is applicable to $E \cdot e_0(I)$; thus, $children(P_1, E \cdot e_0, \mathcal{F}) = \{E \cdot e_0 \cdot e_1\}$. By Lemma 7.2(d), $E \cdot e_0 \cdot e_1$ is in $\mathcal{L}$.

CASE 2. $|children(P_1, E, \mathcal{F})| = 2$: Recall that all nodes in $children(P_1, E, \mathcal{L})$ are univalent. Recall also that, by assumption, $E \cdot e_1$ is in $children(P_1, E, \mathcal{L})$ and is 1-valent in $\mathcal{L}$. Since $|children(P_1, E, \mathcal{F})| = 2$, by Lemma 7.4, $children(P_1, E, \mathcal{F}) = children(P_1, E, \mathcal{L})$ and both nodes in $children(P_1, E, \mathcal{L})$ are actually 1-valent in $\mathcal{L}$. Let the two nodes in $children(P_1, E, \mathcal{L})$ be $E \cdot e_1$ and $E \cdot e_1'$. Because each of P_1 and O_1 is in the same state in $E(I)$ as in $E \cdot e_0(I)$, both e_1 and e_1' are applicable to $E \cdot e_0(I)$ and access the same object. Therefore, it must be that $E \cdot e_0 \cdot e_1$ or $E \cdot e_0 \cdot e_1'$ is in $\mathcal{L}$. Since $E \cdot e_1$ and $E \cdot e_1'$ are 1-valent in $\mathcal{L}$ and since e_1 was chosen arbitrarily as a 1-valent node in $children(P_1, E, \mathcal{L})$, we may assume without loss of generality that $E \cdot e_0 \cdot e_1$ is in $\mathcal{L}$. (Otherwise, we repeat the argument with e_1' instead of e_1). □

In the remainder of this section, let $\mathcal{O}$ denote the object accessed in both e_0 and e_1.

LEMMA 7.6. *$\mathcal{O}$ is not a register.*

Proof. Assume, by way of contradiction, that $\mathcal{O}$ is a register. Since register is a deterministic type, $children(P_0, E, \mathcal{F})$ and $children(P_1, E, \mathcal{F})$ contain a single node each. Let op_0 and op_1 be the (READ or WRITE) operations applied to $\mathcal{O}$ in steps e_0 and e_1, respectively. There are two cases:

CASE 1. *op_k is a* WRITE *operation, for some $k \in \{0, 1\}$*: Let v be the value written into $\mathcal{O}$ by process P_k in step e_k. Since P_k has the same state in $E(I)$ as in $E \cdot e_{\overline{k}}(I)$, e_k is also applicable to $E \cdot e_{\overline{k}}(I)$ and therefore, $children(P_k, E \cdot e_{\overline{k}}, \mathcal{F}) = \{E \cdot e_{\overline{k}} \cdot e_k\}$. By Lemma 7.2(d), $E \cdot e_{\overline{k}} \cdot e_k$ is in $\mathcal{L}$. Clearly, $\mathcal{O}$ has value v in both $E \cdot e_k(I)$ and $E \cdot e_{\overline{k}} \cdot e_k(I)$. In addition, each other object and process P_k has the same state in $E \cdot e_k(I)$ as in $E \cdot e_{\overline{k}} \cdot e_k(I)$. But then e_k overwrites $e_{\overline{k}}$ at E in $\mathcal{L}$. This contradicts Lemma 6.5(b).

CASE 2. *op_0 and op_1 are both* READ *operations:* Let v be the value of $\mathcal{O}$ in configuration $E(I)$. Since the value of a register is not changed by READ operations, $\mathcal{O}$ has the same value v in $E \cdot e_0(I)$ and $E \cdot e_1(I)$. It follows that e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively. Thus $children(P_0, E \cdot e_0, \mathcal{F}) = \{E \cdot e_0 \cdot e_1\}$ and $children(P_1, E \cdot e_1, \mathcal{F}) = \{E \cdot e_1 \cdot e_0\}$. By Lemma 7.2(d), $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are nodes in $\mathcal{L}$ as well. Clearly, the state of each process and each object in $E \cdot e_0 \cdot e_1(I)$ is the same as in $E \cdot e_1 \cdot e_0(I)$. But then e_0 and e_1 commute at E in $\mathcal{L}$. This contradicts Lemma 6.5(a). □

Since $\mathcal{O}$ is not a register and the assumed Consensus algorithm $\mathbf{A}_c$ uses only registers and negation objects, $\mathcal{O}$ must be of type negation. We shall now prove that this is not the case by proving that the state of $\mathcal{O}$ in $E(I)$ is not a legitimate state of type negation. We say that $\mathcal{O}$ is *fresh* in a configuration C of $\mathbf{A}_c$ if its state in C is ☺, and *upset* in C if its state in C is ☹.

For each $k \in \{0, 1\}$, let $j_k \in \{0, 1\}$ and $w_k \in \{0, 1, 2\}$ be the process and initial-value parameters, respectively, of the NEGATE operation to $\mathcal{O}$ of process P_k that is pending in $E(I)$. In other words, the operation to $\mathcal{O}$ of P_k pending in $E(I)$ is NEGATE(j_k, w_k). Recall that the response of a NEGATE operation is 0 or 1. For each $k, u \in \{0, 1\}$, let $d_k^u = (P_k, \text{NEGATE}(j_k, w_k), \mathcal{O}, u)$ — i.e., d_k^u is a step in which P_k

applies NEGATE(j_k, w_k) to $\mathcal{O}$ and receives u as response. Thus, $e_k \in \{d_k^0, d_k^1\}$ for each $k \in \{0,1\}$.

The next lemma states that the application of the two NEGATE operations pending in $E(I)$ does not upset $\mathcal{O}$.

LEMMA 7.7. *For each $k \in \{0,1\}$, let d_k be any step in $\{d_k^0, d_k^1\}$ such that $E \cdot e_{\overline{k}} \cdot d_k$ is a node in $\mathcal{F}$. Then, $\mathcal{O}$ is not upset in $E \cdot e_{\overline{k}} \cdot d_k(I)$.*

Proof. Suppose, for contradiction, that $\mathcal{O}$ is upset in $E \cdot e_{\overline{k}} \cdot d_k(I)$, for some $k \in \{0,1\}$. We first show that $\mathcal{O}$ is also upset in $E \cdot e_k \cdot d_{\overline{k}}(I)$. Since $\mathcal{O}$ is upset in $E \cdot e_{\overline{k}} \cdot d_k(I)$, by the specification of negation, it is easy to verify that one of the following conditions must hold (in verifying this, recall that e_0 and d_0 are NEGATE(j_0, w_0) steps to $\mathcal{O}$, and e_1 and d_1 are NEGATE(j_1, w_1) steps to $\mathcal{O}$):

- $\mathcal{O}$ is upset in $E(I)$.
- $\mathcal{O}$ is fresh in $E(I)$, and either
  - (a) $j_0 = j_1 = 0$, or
  - (b) $j_0 = j_1 = 1$ and $w_0 \neq w_1$.
- The state of $\mathcal{O}$ in $E(I)$ is of the form $\langle v_0, v_1, u_0, u_1 \rangle$ and either
  - (a) $v_0 \neq \perp$ and, for some $\ell \in \{0,1\}$, $j_\ell = 0$; or
  - (b) $v_1 \neq \perp$ and, for some $\ell \in \{0,1\}$, $j_\ell = 1$ and $w_\ell \neq v_1$.

In each of these cases, it can be verified that $\mathcal{O}$ is also upset in $E \cdot e_k \cdot d_{\overline{k}}(I)$.

Since $\mathcal{O}$ is upset in $E \cdot e_1 \cdot d_0(I)$, it is easy to verify that the step $d_0' \in \{d_0^0, d_0^1\} \setminus \{d_0\}$ is also applicable to $E \cdot e_1(I)$ and $\mathcal{O}$ is upset in $E \cdot e_1 \cdot d_0'(I)$ as well. Therefore, $children(P_0, E \cdot e_1, \mathcal{F}) = \{E \cdot e_1 \cdot d_0^0, E \cdot e_1 \cdot d_0^1\}$, and $\mathcal{O}$ is upset in both $E \cdot e_1 \cdot d_0^0(I)$ and $E \cdot e_1 \cdot d_0^1(I)$. Likewise, as $\mathcal{O}$ is upset in $E \cdot e_0 \cdot d_1(I)$, $children(P_1, E \cdot e_0, \mathcal{F}) = \{E \cdot e_0 \cdot d_1^0, E \cdot e_0 \cdot d_1^1\}$, and $\mathcal{O}$ is upset in both $E \cdot e_0 \cdot d_1^0(I)$ and $E \cdot e_0 \cdot d_1^1(I)$. Since $\mathcal{L}$ is 1-full (by Lemma 7.3) and $E \cdot e_1$ is 1-valent in $\mathcal{L}$, it follows that both $E \cdot e_1 \cdot d_0^0$ and $E \cdot e_1 \cdot d_0^1$ are nodes in $\mathcal{L}$. Furthermore, they are 1-valent in $\mathcal{L}$ because $E \cdot e_1$ is. Therefore, as $e_0 \in \{d_0^0, d_0^1\}$, $E \cdot e_1 \cdot e_0$ is a 1-valent node in $\mathcal{L}$. Since $\mathcal{L}$ is a computation tree, there exists $d \in \{d_1^0, d_1^1\}$ such that $E \cdot e_0 \cdot d$ is a node in $\mathcal{L}$. This node is 0-valent in $\mathcal{L}$ because $E \cdot e_0$ is. As we have just shown, $\mathcal{O}$ is upset in $E \cdot e_0 \cdot d(I)$. Therefore, the state of P_0 and each object is the same in $E \cdot e_1 \cdot e_0(I)$ as in $E \cdot e_0 \cdot d(I)$. By Lemma 6.4, $E \cdot e_1 \cdot e_0$ and $E \cdot e_0 \cdot d$ have the same valence in $\mathcal{L}$. This contradicts the fact that $E \cdot e_1 \cdot e_0$ is 1-valent and $E \cdot e_0 \cdot d$ is 0-valent in $\mathcal{L}$.

We reached this contradiction by assuming $\mathcal{O}$ is upset in $E \cdot e_{\overline{k}} \cdot d_k$, for some $k \in \{0,1\}$. Therefore, $\mathcal{O}$ is not upset in $E \cdot e_0 \cdot d_1$ or in $E \cdot e_1 \cdot d_0$. □

Since a negation object, once it has become upset, it remains upset forever, we have:

COROLLARY 7.8. *$\mathcal{O}$ is not upset in $E(I)$, $E \cdot e_0(I)$, or $E \cdot e_1(I)$.*

We now rule out the remaining possibilities for the state of $\mathcal{O}$ in $E(I)$.

LEMMA 7.9. *$\mathcal{O}$ is neither fresh nor in a state of the form $\langle v_0, v_1, u_0, u_1 \rangle$ in $E(I)$.*

Proof. Suppose, for contradiction, that $\mathcal{O}$ is either fresh or in a state of the form $\langle v_0, v_1, u_0, u_1 \rangle$ in $E(I)$. There are two cases, depending on the values of j_0 and j_1 (the process parameters of the two NEGATE operations pending in $E(I)$).

CASE 1. *For some $k \in \{0,1\}$, $j_k = 0$:* If $j_{\overline{k}} = 0$, then both $d_{\overline{k}}^0$ and and $d_{\overline{k}}^1$ are applicable to $E \cdot e_k(I)$, and $\mathcal{O}$ is upset in both $E \cdot e_k \cdot d_{\overline{k}}^0(I)$ and $E \cdot e_k \cdot d_{\overline{k}}^1(I)$, contrary to Lemma 7.7. Thus, $j_{\overline{k}} = 1$. We distinguish two subcases:

SUBCASE 1(a). *$\mathcal{O}$ is fresh in $E(I)$:* Since $\mathcal{O}$ is fresh in $E(I)$, by the specification of negation, both d_1^0 and d_1^1 are applicable to $E(I)$. Thus, $E \cdot d_1^0$ and $E \cdot d_1^1$ are in $\mathcal{F}$. Since E is a critical node in $\mathcal{L}$, by definition, every node in $children(P_1, E, \mathcal{L})$ is univalent. Recall that $E \cdot e_1$ is 1-valent. By Lemma 7.4, both $E \cdot d_1^0$ and $E \cdot d_1^1$

are nodes in $\mathcal{L}$ and are 1-valent in $\mathcal{L}$.

Let $u \in \{0,1\}$ be such that $e_0 = d_0^u$. We further distinguish two cases, depending on whether the initial-value parameters, w_0 and w_1, of the two pending NEGATE operations in $E(I)$ are equal or not:

(i) $w_0 \neq w_1$: By the specification of negation, e_0 is the only step of P_0 that is applicable to $E \cdot d_1^u(I)$, and d_1^u is the only step of P_1 that is applicable to $E \cdot e_0(I)$. Therefore, by Lemma 7.2(d), $E \cdot d_1^u \cdot e_0$ and $E \cdot e_0 \cdot d_1^u$ are in $\mathcal{L}$. Furthermore, $E \cdot d_1^u \cdot e_0$ is 1-valent in $\mathcal{L}$ (because, as we just showed, $E \cdot d_1^u$ is); and $E \cdot e_0 \cdot d_1^u$ is 0-valent in $\mathcal{L}$ (because $E \cdot e_0$ is). By the specification of negation, it can be checked that $\mathcal{O}$ is in the same state in $E \cdot e_0 \cdot d_1^u(I)$ as in $E \cdot d_1^u \cdot e_0(I)$ (namely, in state $\langle w_0, w_1, u, u\rangle$ if $j_0 = 0$, and in state $\langle w_1, w_0, u, u\rangle$ if $j_0 = 1$). Therefore, $E \cdot e_0 \cdot d_1^u$ and $E \cdot d_1^u \cdot e_0$ are both nodes in $\mathcal{L}$, and each process and each object is in the same state in $E \cdot e_0 \cdot d_1^u(I)$ as in $E \cdot d_1^u \cdot e_0(I)$. This implies that e_0 and d_1^u commute at E in $\mathcal{L}$, contrary to Lemma 6.5(a).

(ii) $w_0 = w_1$: The argument of Subcase 1(a)-i also applies in this subcase by replacing d_1^u with $d_1^{\overline{u}}$ throughout. (But now the state of $\mathcal{O}$ in $E \cdot e_0 \cdot d_1^{\overline{u}}(I)$ and $E \cdot d_1^{\overline{u}} \cdot e_0(I)$ is $\langle w_0, w_0, u, \overline{u}\rangle$ if $j_0 = 0$ and $\langle w_0, w_0, \overline{u}, u\rangle$ if $j_0 = 1$.)

SUBCASE 1(b). *$\mathcal{O}$ is in a state of the form $\langle v_0, v_1, u_0, u_1\rangle$ in $E(I)$*: First we claim that $v_0 = \perp$. If not, then by the specification of negation, $\mathcal{O}$ is upset in $E \cdot e_k(I)$ (since e_k is a NEGATE$(0, w_k)$ step to $\mathcal{O}$). This contradicts Corollary 7.8. Thus, $v_0 = \perp$. Given this, by the definition of the states of negation, it must be that $u_0 = \perp$ and $v_1, u_1 \neq \perp$. Recall that, in this case, $e_{\overline{k}}$ is a NEGATE$(1, w_{\overline{k}})$ step to $\mathcal{O}$. Since $v_1 \neq \perp$ and by Corollary 7.8 $\mathcal{O}$ is not upset in $E \cdot e_{\overline{k}}(I)$, it must be that $w_{\overline{k}} = v_1$. Then, it is easy to verify that $d_{\overline{k}}^{u_1}$ is the only step of $P_{\overline{k}}$ that is applicable to $E(I)$. Thus, $e_{\overline{k}} = d_{\overline{k}}^{u_1}$. We further distinguish two cases:

(i) $w_k \neq w_{\overline{k}}$: From the specification of negation, we can check that $d_k^{u_1}$ is the only step of P_k that is applicable to $E(I)$ and $E \cdot e_{\overline{k}}(I)$. Thus, $e_k = d_k^{u_1}$, and by Lemma 7.2(d), $E \cdot e_{\overline{k}} \cdot e_k$ is in $\mathcal{L}$. Also, $e_{\overline{k}}$ is the only step of $P_{\overline{k}}$ that is applicable to $E \cdot e_k$ (since $\mathcal{O}$ is in state $\langle w_k, v_1, u_1, u_1\rangle$ in $E \cdot e_k(I)$). By Lemma 7.2(d), $E \cdot e_k \cdot e_{\overline{k}}$ is a node in $\mathcal{L}$.
By the specification of negation, it can be checked that $\mathcal{O}$ is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$ (namely, in state $\langle w_k, v_1, u_1, u_1\rangle$). Therefore, each process and each object is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$. But then e_0 and e_1 commute at E in $\mathcal{L}$, contrary to Lemma 6.5(a).

(ii) $w_k = w_{\overline{k}}$: The argument in Subcase 1(b)-i also applies in this case by replacing $d_k^{u_1}$ with $d_k^{\overline{u}_1}$ throughout. (But now $\mathcal{O}$ is in state $\langle w_k, v_1, \overline{u}_1, u_1\rangle$ in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$.)

CASE 2. *For both $k \in \{0,1\}$, $j_k = 1$*: In this case, it must be that $w_0 = w_1 = w$, for some $w \in \{0,1\}$. (Otherwise, for each $k \in \{0,1\}$, both $d_{\overline{k}}^0$ and $d_{\overline{k}}^1$ are applicable to $E \cdot e_k(I)$ and $\mathcal{O}$ is upset in $E \cdot e_k \cdot d_{\overline{k}}^0(I)$ and $E \cdot e_k \cdot d_{\overline{k}}^1(I)$, contrary to Lemma 7.7.) We distinguish two subcases:

SUBCASE 2(a). *$\mathcal{O}$ is fresh in $E(I)$*: The argument of Subcase 1(a)-i applies here as well (but now the state of $\mathcal{O}$ in $E \cdot e_0 \cdot d_1^u(I)$ and $E \cdot d_1^u \cdot e_0(I)$ is $\langle \perp, w, \perp, u\rangle$).

SUBCASE 2(b). *$\mathcal{O}$ is in a state of the form $\langle v_0, v_1, u_0, u_1\rangle$ in $E(I)$*: We further distinguish two cases:

(i) $v_1 = \perp$: By the definition of the states of negation, we have $u_1 = \perp$ and

$v_0, u_0 \neq \perp$. Define u as

$$u = \begin{cases} u_0 & \text{if } w \neq v_0, \\ \overline{u}_0 & \text{if } w = v_0. \end{cases}$$

From the specification of negation, for each $k \in \{0,1\}$, d_k^u is the only step of P_k that is applicable to $E(I)$. Hence, $e_k = d_k^u$, for each $k \in \{0,1\}$. Also, e_k is the only step of P_k that is applicable to $E \cdot e_{\overline{k}}(I)$. Thus, by Lemma 7.2(d), both $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are in $\mathcal{L}$. By the specification of negation, $\mathcal{O}$ is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$ (namely, in state $\langle v_0, w, u_0, u \rangle$). Therefore, each process and each object is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$. But then e_0 and e_1 commute at E in $\mathcal{L}$, contrary to Lemma 6.5(a).

(ii) $v_1 \neq \perp$: Since $v_1 \neq \perp$, by the definition of the states of negation, $u_1 \neq \perp$. The argument of Subcase 2(b)-i applies in this case by taking u to be u_1 (instead of defining it as above).

Since all cases lead to a contradiction, the lemma follows. □

By Corollary 7.8 and Lemma 7.9, the state of $\mathcal{O}$ in $E(I)$ is not a legitimate state of type negation. Thus, $\mathcal{O}$ is not a negation object. This, together with Lemma 7.6 which shows that $\mathcal{O}$ is not a register, contradicts our hypothesis that $\mathbf{A}_c$ uses only negation objects and registers. Thus,

THEOREM 7.10. *Type* negation *has Consensus number one.*

8. Type booster has consensus number one. In this section, we show that type booster has Consensus number one. Unfortunately, even the modified version of the bivalence argument of Section 7 does not appear to be helpful in proving that Consensus between two processes cannot be solved using only booster objects and registers. Instead, we use a different technique, originally used (in a somewhat different context) in [12]. Here is the outline of the proof.

Suppose, for contradiction, that type booster has Consensus number at least two. By the universality of Consensus [6], we can implement a negation object for two processes using only booster objects and registers. Thus, the Equality Negation problem for two processes is solvable using only booster objects and registers. Let $\mathbf{A}_{en}$ be an Equality Negation algorithm for two processes that uses a minimal number of booster objects (together with any number of registers).[6] By Theorem 3.1, $\mathbf{A}_{en}$ must use at least one booster object. We will show that $\mathbf{A}_{en}$ can be modified into an Equality Negation algorithm that uses one booster object less than $\mathbf{A}_{en}$, thus contradicting the definition of $\mathbf{A}_{en}$.

In this section we deal exclusively with *full* computation trees of $\mathbf{A}_{en}$. We first establish some properties of such trees that will be needed later on.

LEMMA 8.1. *Let I be any initial configuration of $\mathbf{A}_{en}$, and let $\mathcal{F}$ be the full computation tree of $\mathbf{A}_{en}$ from I. Let S_0, $S_0 \cdot S$, and S_1 be any nodes in $\mathcal{F}$ such that every process that takes a step in S and every object, except a* booster *object O, is in the same state in $S_0(I)$ as in $S_1(I)$. Suppose, further, that*

- *the state of O in $S_1(I)$ is ☹, or*
- *the state of O in $S_1(I)$ is of the form $\langle *, *, *, *, *, d \rangle$, for some $d \in \{0,1\}$, and S contains no* REVEAL *step to O with return value $\overline{d}$ (i.e., S contains no step of the form $(*, \text{REVEAL}(*,*,*), O, \overline{d})$).*

[6] This number must be finite by the Termination property of Equality Negation and the fact that type booster is bounded nondeterministic.

Then, $S_1 \cdot S$ is a node in $\mathcal{F}$, and every process that takes a step in S and every object except O is in the same state in $S_0 \cdot S(I)$ as in $S_1 \cdot S(I)$.

Proof. From the specification of booster, it is easy to verify that

- the only state that is reachable from ☹ is itself, and a REVEAL operation can return either 0 or 1 when applied to a booster object in state ☹;
- every state reachable from a state of the form $\langle *, *, *, *, *, d \rangle$ is either of the same form or ☹, and a REVEAL operation can return d when applied to a booster object in a state of the form $\langle *, *, *, *, *, d \rangle$ or in state ☹.

Let S^i be the schedule consisting of the first i steps of S, for $0 \leq i \leq |S|$. Using the above two properties, by a straightforward induction on i, we can show that for all i, $0 \leq i \leq |S|$, S^i is applicable to $S_1(I)$, and every process that takes a step in S and every object except O is in the same state in $S_0 \cdot S^i(I)$ as in $S_1 \cdot S^i(I)$. The lemma follows immediately from these facts, since $S = S^\ell$, where $\ell = |S|$. □

If the root of a full computation tree of $\mathbf{A}_{en}$ is univalent, the following lemma tells how the valence of the root relates to the possible decisions of the processes.

LEMMA 8.2. *Let $k, u, v \in \{0, 1\}$, I be any initial configuration of $\mathbf{A}_{en}$, $\mathcal{F}$ be the full computation tree of $\mathbf{A}_{en}$ from I, and S be any node of $\mathcal{F}$. If the root of $\mathcal{F}$ is v-valent and P_k has decided u in $S(I)$, then*

$$u = \begin{cases} v & \text{if } k = 0, \\ v & \text{if } k = 1 \text{ and } P_0, P_1 \text{ have different initial values in } I, \\ \overline{v} & \text{if } k = 1 \text{ and } P_0, P_1 \text{ have the same initial value in } I. \end{cases}$$

Proof. Let S_0 be a solo schedule of P_0 such that $S \cdot S_0$ is in $\mathcal{F}$ and P_0 has decided in $S \cdot S_0(I)$ (S_0 exists by Lemma 6.2). Since (by hypothesis) the root of $\mathcal{F}$ is v-valent, P_0 has decided v in $S \cdot S_0(I)$. By hypothesis, P_k has decided u in $S(I)$; since the decision of a process is irrevocable, P_k has decided u in $S \cdot S_0(I)$. If $k = 0$, clearly $u = v$, as wanted. If $k = 1$, then P_0 has decided v in $S \cdot S_0(I)$, and P_1 has decided u in $S \cdot S_0(I)$. By the Negation property, $u = v$, if P_0 and P_1 have different initial values in I; and $u = \overline{v}$, if P_0 and P_1 have the same initial value in I — as wanted. □

LEMMA 8.3. *For some initial configuration I of algorithm $\mathbf{A}_{en}$, the full computation tree $\mathcal{F}$ of $\mathbf{A}_{en}$ from I has a bivalent root.*

Proof. Suppose, for contradiction, that the root of every full computation tree of $\mathbf{A}_{en}$ is univalent. For each $i, j \in \{0, 1, 2\}$, let $I_{i,j}$ be the initial configuration of $\mathbf{A}_{en}$ in which P_0 and P_1 have initial values i and j, respectively, and let $\mathcal{F}_{i,j}$ be the full computation tree of $\mathbf{A}_{en}$ from $I_{i,j}$.

Without loss of generality, assume that the root of $\mathcal{F}_{1,1}$ is 0-valent. By Lemma 6.2 (applied to the root, $S_\perp$, of $\mathcal{F}_{1,1}$), for each $k \in \{0, 1\}$, there is a solo schedule S_k of P_k such that $S_\perp \cdot S_k = S_k$ is a node in $\mathcal{F}_{1,1}$ and P_k has decided in $S_k(I_{1,1})$. By Lemma 8.2, P_k decides k in $S_k(I_{1,1})$; see Figure 8.1(a).

Since $I_{1,1}$ and $I_{1,2}$ differ only in the state of process P_1 and since S_0 is a solo schedule of P_0, by Lemma 6.1, S_0 is node in $\mathcal{F}_{1,2}$ and P_0 has the same state in $S_0(I_{1,2})$ as in $S_0(I_{1,1})$; see Figure 8.1(b). Thus, P_0 decides 0 in $S_0(I_{1,2})$. By the hypothesis that the root of $\mathcal{F}_{1,2}$ is univalent, it follows that the root of $\mathcal{F}_{1,2}$ is 0-valent. By Lemma 6.2 (applied to the root of $\mathcal{F}_{1,2}$), there is a solo schedule S_1' of P_1 such that S_1' is a node in $\mathcal{F}_{1,2}$ and P_1 has decided in $S_1'(I_{1,2})$. By Lemma 8.2, P_1 decides 0 in $S_1'(I_{1,2})$. Repeating the argument in this paragraph with $I_{1,2}$ replaced by $I_{0,1}$ and with the roles of 0 and 1 interchanged, we can prove that there exists a solo schedule S_0' of P_0 such that S_0' is a node in $\mathcal{F}_{0,1}$ and P_0 decides 1 in $S_0'(I_{0,1})$; see Figure 8.1(c).

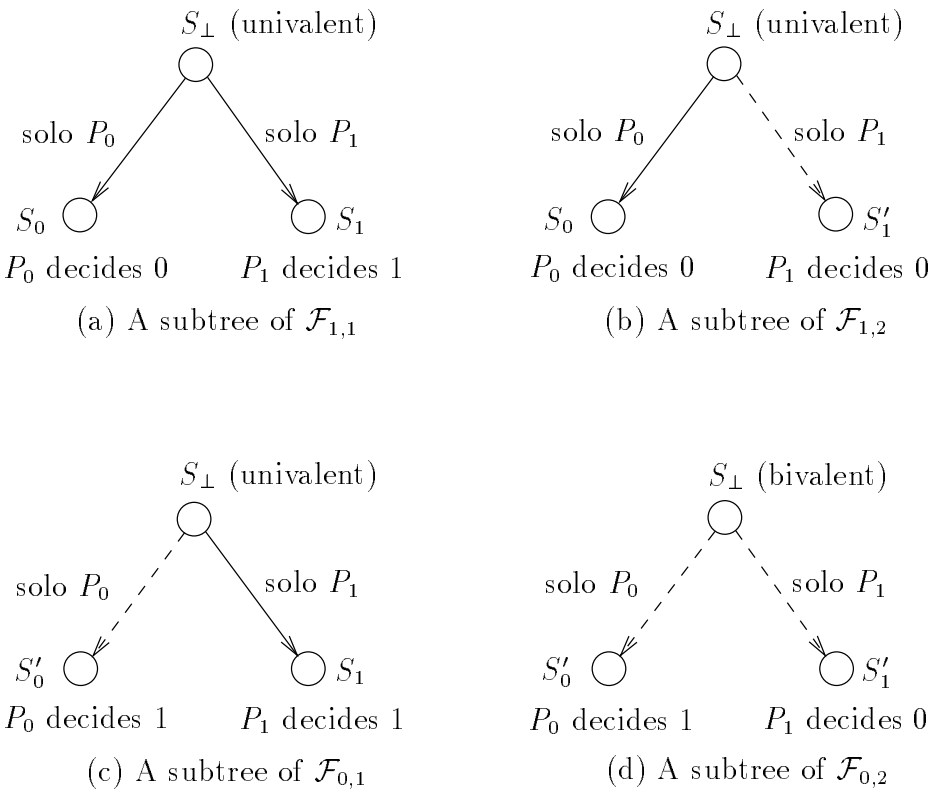

FIG. 8.1. *Subtrees of four full computation trees of* $\mathbf{A}_{en}$

Since $I_{0,1}$ and $I_{0,2}$ differ only in the state of P_1 and S'_0 is a solo schedule of P_0, by Lemma 6.1, S'_0 is a node in $\mathcal{F}_{0,2}$ and P_0 has the same state in $S'_0(I_{0,1})$ as in $S'_0(I_{0,2})$. Since P_0 decides 1 in $S'_0(I_{0,1})$, it also decides 1 in $S'_0(I_{0,2})$. By a similar argument, replacing $I_{0,1}$ and S'_0 by $I_{1,2}$ and S'_1, respectively, and interchanging the roles of P_0 and P_1, we can show that S'_1 is a node in $\mathcal{F}_{0,2}$ and P_1 decides 0 in $S'_1(I_{0,2})$; see Figure 8.1(d). Thus, the root of $\mathcal{F}_{0,2}$ has descendents, namely, S'_0 and S'_1, in which processes have decided different values in the corresponding configurations. By Lemma 8.2, the root of $\mathcal{F}_{0,2}$ cannot be univalent. This contradicts our hypothesis that the root of every full computation tree of $\mathbf{A}_{en}$ is univalent. □

In the remainder of the section we focus exclusively on the initial configuration I and the full computation tree $\mathcal{F}$ of $\mathbf{A}_{en}$ from I whose existence is asserted by Lemma 8.3. Thus, from now on, we refer to the valence of a node without explicitly specifying the computation tree, since that will always be $\mathcal{F}$.

Since the root of $\mathcal{F}$ is bivalent, by Lemma 6.7, $\mathcal{F}$ has a critical node E. By the definition of critical node, there are steps e_0 and e_1 of processes P_0 and P_1, respectively, such that $E \cdot e_0$ and $E \cdot e_1$ are univalent nodes of opposite valence. Without loss of generality, assume that $E \cdot e_0$ is 0-valent and $E \cdot e_1$ is 1-valent (if not, we simply rename e_0 and e_1). By Lemma 6.6, e_0 and e_1 access the same object, say $\mathcal{O}$. We can show that $\mathcal{O}$ is not a register, by using an argument similar to the one used in the proof

of Lemma 7.6 (except simpler, since we are now working with the full computation tree and need not worry about whether certain schedules applicable to I have been pruned). Thus, $\mathcal{O}$ is a booster object. In what follows, we say that $\mathcal{O}$ is *fresh* in a configuration C of $\mathbf{A}_{en}$ if its state in C is ☺ and *upset* in C if its state in C is ☹.

We now outline how the proof proceeds: To show that the assumed Equality Negation algorithm $\mathbf{A}_{en}$ that uses only booster objects and registers does not exist, we derive a contradiction in two stages. In the first stage, we prove certain facts about the configuration $E(I)$ and the booster object $\mathcal{O}$. Specifically, we show that $\mathcal{O}$ is fresh in $E(I)$ (Corollary 8.5 and Lemma 8.7), that each process has an ENROLL operation (with a different parameter) to $\mathcal{O}$ pending in $E(I)$, and that neither process can decide before one of them has applied a REVEAL operation to $\mathcal{O}$ (Lemma 8.11). Furthermore, if neither process has yet applied a REVEAL operation to $\mathcal{O}$ but each is about to, then the parameters of the two pending REVEAL operations must be such as to ensure that $\mathcal{O}$ does not become upset (Lemma 8.13). These facts are then exploited in the second stage of the proof, to show how to solve Equality Negation using only registers and the booster objects of $\mathbf{A}_{en}$ *except* $\mathcal{O}$. We have thus reached a contradiction, as $\mathbf{A}_{en}$ was assumed to use the smallest possible number of booster objects. The details of these two stages are carried out in the next two subsections.

8.1. Stage one of the proof. We now show that $\mathcal{O}$ is fresh in $E(I)$. We do this by proving that, in $E(I)$, $\mathcal{O}$ is neither upset (Corollary 8.5) nor in a state of the form $\langle V, v_0, v_1, u_0, u_1, d\rangle$ (Lemma 8.7).

LEMMA 8.4. *Object $\mathcal{O}$ is not upset in $E\cdot e_0(I)$ or in $E\cdot e_1(I)$.*

Proof. Suppose, for contradiction, that $\mathcal{O}$ is upset in $E\cdot e_k(I)$, for some $k\in\{0,1\}$. By Lemma 6.2 (applied to node $E\cdot e_{\overline{k}}$ in $\mathcal{F}$), there is a solo schedule $S_{\overline{k}}$ of $P_{\overline{k}}$ such that $E\cdot e_{\overline{k}}\cdot S_{\overline{k}}$ is a node in $\mathcal{F}$ and $P_{\overline{k}}$ has decided in $E\cdot e_{\overline{k}}\cdot S_{\overline{k}}(I)$. Comparing configurations $E(I)$ and $E\cdot e_k(I)$, we note that $P_{\overline{k}}$ and each object other than $\mathcal{O}$ is in the same state in both. Since, by assumption, $\mathcal{O}$ is upset in $E\cdot e_k(I)$ and $e_{\overline{k}}\cdot S_{\overline{k}}$ is a solo schedule of $P_{\overline{k}}$, by Lemma 8.1, $E\cdot e_k\cdot e_{\overline{k}}\cdot S_{\overline{k}}$ is a node in $\mathcal{F}$, and $P_{\overline{k}}$ is in the same state in $E\cdot e_{\overline{k}}\cdot S_{\overline{k}}(I)$ as in $E\cdot e_k\cdot e_{\overline{k}}\cdot S_{\overline{k}}(I)$. Thus, $P_{\overline{k}}$ decides the same value in both $E\cdot e_{\overline{k}}\cdot S_{\overline{k}}(I)$ and $E\cdot e_k\cdot e_{\overline{k}}\cdot S_{\overline{k}}(I)$. By Lemma 6.3, $E\cdot e_{\overline{k}}\cdot S_{\overline{k}}$ and $E\cdot e_k\cdot e_{\overline{k}}\cdot S_{\overline{k}}$ have the same valence. Since $E\cdot e_{\overline{k}}$ and $E\cdot e_k$ are univalent and have descendents of the same valence, they also have the same valence. This contradicts the fact that $E\cdot e_0$ is 0-valent while $E\cdot e_1$ is 1-valent. □

Once a booster object is upset, it remains upset forever. Thus, Lemma 8.4 implies:

COROLLARY 8.5. *Object $\mathcal{O}$ is not upset in $E(I)$.*

LEMMA 8.6. *$E\cdot e_0\cdot e_1$ and $E\cdot e_1\cdot e_0$ are nodes in $\mathcal{F}$.*

Proof. By definition of $\mathcal{F}$, it suffices to show that, for each $k\in\{0,1\}$, e_k is applicable to $E\cdot e_{\overline{k}}(I)$. If e_k is an ENROLL step, the lemma follows immediately by the specification of booster.

Suppose that e_k is a REVEAL step. If $\mathcal{O}$ is fresh in $E(I)$, then $\mathcal{O}$ is upset in $E\cdot e_k(I)$, contrary to Lemma 8.4. Thus, $\mathcal{O}$ is not fresh in $E(I)$. By Corollary 8.5, $\mathcal{O}$ is not upset in $E(I)$. Thus, $\mathcal{O}$ is in a state of the form $\langle *,*,*,*,*,d\rangle$ in $E(I)$, for some $d\in\{0,1\}$. By the specification of booster, and since $\mathcal{O}$ is not upset in $E\cdot e_k(I)$, the response returned by the REVEAL operation in e_k is d. Since $\mathcal{O}$ is in a state of the form $\langle *,*,*,*,*,d\rangle$ in $E(I)$ and is not upset in $E\cdot e_{\overline{k}}(I)$, it must be in a state of the form $\langle *,*,*,*,*,d\rangle$ in $E\cdot e_{\overline{k}}(I)$ as well (because the last component of the state is never changed, once it is set). By the specification of booster, a REVEAL operation to $\mathcal{O}$ applied in such a state can return the response d. Thus, e_k is indeed applicable to $E\cdot e_{\overline{k}}(I)$. □

LEMMA 8.7. *The state of $\mathcal{O}$ in $E(I)$ is not of the form $\langle V, v_0, v_1, u_0, u_1, d\rangle$.*

Proof. Suppose, for contradiction, that the state of $\mathcal{O}$ in $E(I)$ is $\langle V, v_0, v_1, u_0, u_1, d\rangle$. By Lemma 8.6, both $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are nodes in $\mathcal{F}$. We shall show that

the state of $\mathcal{O}$ is the same in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$ (3)

Clearly, the state of each process and object except $\mathcal{O}$ is the same in $E \cdot e_1 \cdot e_0(I)$ as in $E \cdot e_0 \cdot e_1(I)$. By (3), we have thus shown that e_0 and e_1 commute at E in $\mathcal{F}$. This contradicts Lemma 6.5(a), as wanted.

It remains to show (3). Let op_0 and op_1 be the (ENROLL or REVEAL) operations to $\mathcal{O}$ of processes P_0 and P_1 that are pending in $E(I)$, respectively. There are three cases, depending on the types of op_0 and op_1.

CASE 1. *Both operations pending in $E(I)$ are* ENROLL: Let $op_0 = \text{ENROLL}(i_0)$ and $op_1 = \text{ENROLL}(i_1)$, for some $i_0, i_1 \in \{0, 1\}$. For each $k \in \{0, 1\}$, let r_k be the response of $\mathcal{O}$ to the $\text{ENROLL}(i_k)$ operation in step e_k. There are three subcases:

SUBCASE 1(a). $i_0 = i_1 = 0$: From the specification of booster, $\mathcal{O}$ is upset in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$.

SUBCASE 1(b). $i_0 = i_1 = 1$: From the specification of booster, $\mathcal{O}$ is in state $\langle V \cup \{r_0, r_1\}, v_0, v_1, u_0, u_1, d\rangle$ in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$.

SUBCASE 1(c). *For some $k \in \{0, 1\}$, $i_k = 0$ and $i_{\overline{k}} = 1$*: Without loss of generality we assume that $i_0 = 0$ and $i_1 = 1$. Then $v_0 = \perp$; if not, $\mathcal{O}$ would be upset in $E \cdot e_0(I)$, which contradicts Lemma 8.4. By the definition of the set of states of booster, $u_0 = \perp$. From this, it follows that $\mathcal{O}$ is in state $\langle V \cup \{r_1\}, r_0, v_1, \perp, u_1, d\rangle$ in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$.

CASE 2. *One of the two operations pending in $E(I)$ is* ENROLL *and the other is* REVEAL: Without loss of generality, we may assume that $op_0 = \text{ENROLL}(i_0)$ and $op_1 = \text{REVEAL}(i_1, v, u)$, for some $i_0, i_1, u \in \{0, 1\}$ and $v \in \{0, 1, 2\}$. Let r be the response of the $\text{ENROLL}(i_0)$ operation that is returned in step e_0. There are three subcases:

SUBCASE 2(a). $i_0 = 0$: Then $v_0 = \perp$ (otherwise, $\mathcal{O}$ would be upset in $E \cdot e_0(I)$, contrary to Lemma 8.4), and hence $u_0 = \perp$ (by the definition of the set of states of booster). Also, $i_1 = 1$ (otherwise, $\mathcal{O}$ would be upset in $E \cdot e_1(I)$). Thus, $\mathcal{O}$ is in state $\langle V, \perp, v_1, \perp, u_1, d\rangle$ in $E(I)$ and the operations of P_0 and P_1 pending in $E(I)$ are $\text{ENROLL}(0)$ and $\text{REVEAL}(1, v, u)$, respectively. From the specification of booster, the state of $\mathcal{O}$ in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$ is

$$\begin{cases} \langle V, r, v, \perp, u, d\rangle & \text{if } v \in V \textbf{ and } (u_1 = \perp \textbf{ or } (v = v_1 \textbf{ and } u = u_1)), \\ \text{☹} & \text{otherwise.} \end{cases}$$

SUBCASE 2(b). $i_0 = 1$ *and* $i_1 = 0$: Since $\mathcal{O}$ is not upset in $E \cdot e_1(I)$, the Boolean expression

$$v = v_0 \textbf{ and } u_0 = \perp \textbf{ and } (u_1 = \perp \textbf{ or } (v = v_1 \Leftrightarrow u \neq u_1))$$

is true, and $\mathcal{O}$ is in state $\langle V, v_0, v_1, u, u_1, d\rangle$ in $E \cdot e_1(I)$. Thus, $\mathcal{O}$ is in state $\langle V \cup \{r\}, v_0, v_1, u, u_1, d\rangle$ in $E \cdot e_1 \cdot e_0(I)$.

The state of $\mathcal{O}$ in $E \cdot e_0(I)$ is $\langle V \cup \{r\}, v_0, v_1, \perp, u_1, d\rangle$ (recall that $u_0 = \perp$). Since the preceding Boolean expression is true, by the specification of booster, the application of $\text{REVEAL}(0, v, u)$ to $\mathcal{O}$ in state $\langle V \cup \{r\}, v_0, v_1, \perp, u_1, d\rangle$ does not cause $\mathcal{O}$ to become upset; in fact, it causes it to enter state $\langle V \cup \{r\}, v_0, v_1, u, u_1, d\rangle$. Thus, e_1 is applicable to $E \cdot e_0(I)$, and the state of $\mathcal{O}$ in $E \cdot e_0 \cdot e_1(I)$ is $\langle V \cup \{r\}, v_0, v_1, u, u_1, d\rangle$, which is the same as in $E \cdot e_1 \cdot e_0(I)$.

SUBCASE 2(c). $i_0 = 1$ *and* $i_1 = 1$: Similar to Subcase 2(b), except that, in this case, the state of $\mathcal{O}$ in $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$ is $\langle V \cup \{r\}, v_0, v, u_0, u, d\rangle$, instead of $\langle V \cup \{r\}, v_0, v_1, u, u_1, d\rangle$.

CASE 3. *Both operations pending in $E(I)$ are* REVEAL: Let $op_0 = \text{REVEAL}(i_0, v, u)$ and $op_1 = \text{REVEAL}(i_1, v', u')$, for some $i_0, i_1, u, u' \in \{0,1\}$ and $v, v' \in \{0,1,2\}$. There are three subcases.

SUBCASE 3(a). $i_0 = i_1 = 0$: Then $\mathcal{O}$ is upset in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$.

SUBCASE 3(b). $i_0 \neq i_1$: Without loss of generality, we may assume that $i_0 = 0$ and $i_1 = 1$. It must be the case that $v = v_0$ and $u_0 = \perp$ (otherwise $\mathcal{O}$ is upset in $E \cdot e_0(I)$, contrary to Lemma 8.4). Also $v' \in V$ and either $u_1 = \perp$, or $v' = v_1$ and $u' = u_1$ (otherwise $\mathcal{O}$ is upset in $E \cdot e_1(I)$). Thus, the state of $\mathcal{O}$ in $E(I)$ is either $\langle V, v, \perp, \perp, \perp, d\rangle$ or $\langle V, v, v', \perp, u', d\rangle$. In either case, it can be verified from the specification of booster that the state of $\mathcal{O}$ in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$ is

$$\begin{cases} \langle V, v, v', u, u', d\rangle & \text{if } v = v' \Leftrightarrow u \neq u', \\ \text{☹} & \text{otherwise.} \end{cases}$$

SUBCASE 3(c). $i_0 = i_1 = 1$: Since $\mathcal{O}$ is not upset in either one of $E \cdot e_0(I)$ or $E \cdot e_1(I)$, it must be that $v, v' \in V$ and $\mathcal{O}$ is in state $\langle V, v_0, v, u_0, u, d\rangle$ in $E \cdot e_0(I)$ and in $\langle V, v_0, v', u_0, u', d\rangle$ in $E \cdot e_1(I)$. From the specification of booster, it can be checked that the state of $\mathcal{O}$ in $E \cdot e_0 \cdot e_1(I)$ is

$$\begin{cases} \langle V, v_0, v', u_0, u', d\rangle & \text{if } v' = v \text{ and } u' = u, \\ \text{☹} & \text{otherwise.} \end{cases}$$

Similarly, the state of $\mathcal{O}$ in $E \cdot e_1 \cdot e_0(I)$ is

$$\begin{cases} \langle V, v_0, v, u_0, u, d\rangle & \text{if } v = v' \text{ and } u = u', \\ \text{☹} & \text{otherwise.} \end{cases}$$

Thus, $\mathcal{O}$ is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$.
This completes the proof of (3), and thereby of the lemma. □

By Corollary 8.5 and Lemma 8.7, $\mathcal{O}$ must be fresh in $E(I)$. Furthermore, each P_k must have an ENROLL operation to $\mathcal{O}$ pending in $E(I)$. (Otherwise, by the specification of booster, $\mathcal{O}$ would be upset in $E \cdot e_k(I)$, contrary to Lemma 8.4.) For each $k \in \{0,1\}$, let $j_k \in \{0,1\}$ be the parameter of the ENROLL operation of P_k pending in $E(I)$.

The next lemma states that the parameters of the two ENROLL operations to $\mathcal{O}$ pending in $E(I)$ are different.

LEMMA 8.8. $j_0 \neq j_1$.

Proof. Suppose, for contradiction, that $j_0 = j_1$. As in the proof of Lemma 8.7, we shall show that $\mathcal{O}$ is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$. This fact, together with Lemma 8.6, implies that e_0 and e_1 commute at E in $\mathcal{F}$, thus contradicting Lemma 6.5(a).

Recall that $j_0, j_1 \in \{0,1\}$. There are two cases:
CASE 1. $j_0 = j_1 = 0$: Then $\mathcal{O}$ is upset in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$.
CASE 2. $j_0 = j_1 = 1$: Let r_0 and r_1 be the responses of $\mathcal{O}$ returned to the ENROLL operations in e_0 and e_1, respectively. Then $\mathcal{O}$ is in state $\langle \{r_0, r_1\}, \perp, \perp, \perp, \perp, 1\rangle$, in both $E \cdot e_0 \cdot e_1(I)$ and $E \cdot e_1 \cdot e_0(I)$. □

We next show that, after applying its ENROLL operation that is pending in $E(I)$, no process can decide unless some process has applied at least one REVEAL operation to $\mathcal{O}$ (Lemma 8.11). In the next two lemmata, we establish some properties of schedules applicable to $E(I)$ used to prove this fact.

LEMMA 8.9. *For any $k \in \{0,1\}$ and any node $E \cdot S$ in $\mathcal{F}$, if P_k is the process that takes the first step in S, then $E \cdot S$ is k-valent.*

Proof. Later we prove that

for each $k \in \{0,1\}$, the nodes in $children(P_k, E, \mathcal{F})$ have the same valence. (4)

Let e be the first step in S, which is taken by process P_k, for some $k \in \{0,1\}$. Thus, $E \cdot e$ is a node in $children(P_k, E, \mathcal{F})$. By (4), all nodes in $children(P_k, E, \mathcal{F})$ are univalent and have the same valence. One of these children, namely $E \cdot e_k$, is k-valent. Therefore, $E \cdot e$ is also k-valent. Since $E \cdot S$ is a descendent of $E \cdot e$, it follows that $E \cdot S$ is k-valent.

It remains to prove (4). We prove it for $k = 0$. (The proof for $k = 1$ can be obtained by interchanging the roles of 0 and 1 in the argument below.) For each $v \in \{0,1,2\}$, let $e_0^v = (P_0, \text{ENROLL}(j_0), \mathcal{O}, v)$. Since $\mathcal{O}$ is fresh in $E(I)$ and an ENROLL operation applied to a fresh booster object can return any value in $\{0,1,2\}$, it follows that $children(P_0, E, \mathcal{F}) = \{ E \cdot e_0^v : 0 \leq v \leq 2 \}$. Since E is a critical node, all nodes in $children(P_0, E, \mathcal{F})$ are univalent. To prove (4), it suffices to show that for any $v, v' \in \{0,1,2\}$, $E \cdot e_0^v$ and $E \cdot e_0^{v'}$ have the same valence. From the specification of booster and the fact that $\mathcal{O}$ is fresh in $E(I)$ we have:

(a) For any $v \in \{0,1,2\}$, the state of $\mathcal{O}$ in $E \cdot e_0^v(I)$ is $\langle *, *, *, *, *, j_0 \rangle$.

(b) For any $v \in \{0,1,2\}$, if S is a schedule applicable to $E \cdot e_0^v(I)$, then the state of $\mathcal{O}$ in $E \cdot e_0^v \cdot S(I)$ is either of the form $\langle *, *, *, *, *, j_0 \rangle$ or ☹.

(c) A REVEAL operation can return j_0 when applied to a booster object in a state of the form $\langle *, *, *, *, *, j_0 \rangle$ or in state ☹.

Let v be any element of $\{0,1,2\}$. By Lemma 6.2 (applied to node $E \cdot e_0^v$ in $\mathcal{F}$), there is a solo schedule S of P_1 such that $E \cdot e_0^v \cdot S$ is a node in $\mathcal{F}$ and P_1 has decided in $E \cdot e_0^v \cdot S(I)$. Since $\mathcal{F}$ is a full computation tree, by (a)–(c) we may assume, without loss of generality, that S contains no step of the form $(*, \text{REVEAL}(*, *, *), \mathcal{O}, \overline{j}_0)$.

For any $v' \in \{0,1,2\}$, the state of P_1 and each object except $\mathcal{O}$ is the same in $E \cdot e_0^v(I)$ as in $E \cdot e_0^{v'}(I)$. Since the state of $\mathcal{O}$ in $E \cdot e_0^{v'}(I)$ is of the form $\langle *, *, *, *, *, j_0 \rangle$ and since S does not contain a REVEAL step to $\mathcal{O}$ with return value $\overline{j}_0$, the hypothesis of Lemma 8.1 applies for $S_0 = E \cdot e_0^v$ and $S_1 = E \cdot e_0^{v'}$. Thus, $E \cdot e_0^{v'} \cdot S$ is a node in $\mathcal{F}$, and P_1 is in the same state in $E \cdot e_0^v \cdot S(I)$ as in $E \cdot e_0^{v'} \cdot S(I)$. This implies that P_1 decides the same value in these two configurations. By Lemma 6.3, nodes $E \cdot e_0^v \cdot S$ and $E \cdot e_0^{v'} \cdot S$ have the same valence. Hence, $E \cdot e_0^v$ and $E \cdot e_0^{v'}$ (which are univalent since E is critical) must also have the same valence. This completes the proof of (4) and thereby of the lemma. □

LEMMA 8.10. *Let $E \cdot S$ be any node in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$, and d_k be the first step of P_k in S, for any $k \in \{0,1\}$. Let $\hat{S}$ be the schedule obtained from S by moving d_k to the beginning (i.e., if $S = S' \cdot d_k \cdot S''$ for some S' and S'', then $\hat{S} = d_k \cdot S' \cdot S''$). Then, $E \cdot \hat{S}$ is a node in $\mathcal{F}$, and the state of each process and each object other than $\mathcal{O}$ is the same in $E \cdot \hat{S}(I)$ as in $E \cdot S(I)$.*

Proof. Let S' and S'' be such that $S = S' \cdot d_k \cdot S''$. Recall that process P_k has an ENROLL(j_k) to $\mathcal{O}$ pending in $E(I)$. Since S' is a solo schedule of $P_{\overline{k}}$ (because d_k is the first step of P_k in S), the operation of P_k pending in $E \cdot S'(I)$ is still ENROLL(j_k) to $\mathcal{O}$. Thus, d_k is an ENROLL step to $\mathcal{O}$. By the specification of booster, d_k is applicable to $E(I)$. Clearly, as d_k is a step of P_k that accesses (only) $\mathcal{O}$, the state of $P_{\overline{k}}$ and each object other than $\mathcal{O}$ is the same in $E(I)$ as in $E \cdot d_k(I)$. Also, the state of $\mathcal{O}$ in $E \cdot d_k(I)$ is of the form $\langle *, *, *, *, *, j_k \rangle$ (recall that d_k is an ENROLL(j_k) step to $\mathcal{O}$). Since $E \cdot S'$ is a node in $\mathcal{F}$ and S' is a solo schedule of $P_{\overline{k}}$ that does not contain a REVEAL step to $\mathcal{O}$, by Lemma 8.1 (applied for $S_0 = E$, $S_1 = E \cdot d_k$ and $S = S'$), $E \cdot d_k \cdot S'$ is a node in $\mathcal{F}$, and the state of $P_{\overline{k}}$ and each object other than $\mathcal{O}$ is the same in $E \cdot d_k \cdot S'(I)$ as in $E \cdot S'(I)$. It follows that each process and each object other than $\mathcal{O}$ is in the

same state in $E \cdot d_k \cdot S'(I)$ as in $E \cdot S' \cdot d_k(I)$. By hypothesis, $E \cdot S' \cdot d_k \cdot S''$ is a node in $\mathcal{F}$ and S'' does not contain a REVEAL step to $\mathcal{O}$. Again by Lemma 8.1 (applied for $S_0 = E \cdot S' \cdot d_k$, $S_1 = E \cdot d_k \cdot S'$ and $S = S''$), $E \cdot d_k \cdot S' \cdot S''$ is a node in $\mathcal{F}$, and the state of each process and each object other than $\mathcal{O}$ is the same in $E \cdot S' \cdot d_k \cdot S''(I)$ as in $E \cdot d_k \cdot S' \cdot S''(I)$. □

LEMMA 8.11. *Let $E \cdot S$ be any node in $\mathcal{F}$ such that some process has decided in $E \cdot S(I)$. Then, S has a* REVEAL *step to $\mathcal{O}$.*

Proof. Suppose, for contradiction, that some process has decided in $E \cdot S(I)$, but S contains no REVEAL step to $\mathcal{O}$. Without loss of generality, we may assume that both processes take steps in S. (If S contains only steps of process P_k for some $k \in \{0,1\}$, we extend S by a step of $P_{\overline{k}}$, which cannot be a REVEAL step to $\mathcal{O}$ because the operation of $P_{\overline{k}}$ pending in $E(I)$ is an ENROLL to $\mathcal{O}$, and consider the extended schedule instead.)

Let P_ℓ, $\ell \in \{0,1\}$, be the process that does *not* take the first step in S, and $\hat{S}$ be the schedule obtained from S by moving the first step of P_ℓ to the beginning. By Lemma 8.10, $E \cdot \hat{S}$ is a node in $\mathcal{F}$, and the state of each process and each object except $\mathcal{O}$ is the same in $E \cdot \hat{S}(I)$ as in $E \cdot S(I)$. In particular, the process that has decided in $E \cdot S(I)$ decides the same value in $E \cdot \hat{S}(I)$. By Lemma 6.3, $E \cdot S$ and $E \cdot \hat{S}$ have the same valence. This contradicts Lemma 8.9, since P_ℓ takes the first step in $\hat{S}$ while $P_{\overline{\ell}}$ takes the first step in S. □

To summarise: $\mathcal{O}$ is fresh in $E(I)$, each P_k has an ENROLL(j_k) operation to $\mathcal{O}$ pending in $E(I)$, and $j_0 \neq j_1$. By Lemma 8.11, no process can decide unless a REVEAL operation is applied to $\mathcal{O}$. Thus, by the Termination requirement of Equality Negation, there is a schedule S applicable to $E(I)$ so that no process has taken a REVEAL step to $\mathcal{O}$ in S, and each process has a REVEAL operation to $\mathcal{O}$ pending in $E \cdot S(I)$. S contains ENROLL steps of both processes because each process P_k, $k \in \{0,1\}$, has an ENROLL(j_k) operation to $\mathcal{O}$ pending in $E(I)$ but has a REVEAL operation to $\mathcal{O}$ pending in $E \cdot S(I)$. By Lemma 8.8, S contains both ENROLL(0) and ENROLL(1) steps to $\mathcal{O}$.

The next lemma proves that, in any execution starting from configuration $E(I)$, if each of the two processes is about to apply its first REVEAL operation to $\mathcal{O}$, then the application of these two operations does not cause $\mathcal{O}$ to become upset.

LEMMA 8.12. *Consider any node $E \cdot S$ in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$ and both processes have a* REVEAL *operation to $\mathcal{O}$ pending in $E \cdot S(I)$. For each $k \in \{0,1\}$, let d_k be any step of P_k and $c_{\overline{k}}$ be any step of $P_{\overline{k}}$ such that $E \cdot S \cdot d_k \cdot c_{\overline{k}}$ is a node in $\mathcal{F}$. (Note that d_k and $c_{\overline{k}}$ are* REVEAL *steps that access $\mathcal{O}$.) Then, $\mathcal{O}$ is not upset in $E \cdot S \cdot d_k \cdot c_{\overline{k}}(I)$.*

Proof. Suppose, to the contrary, that $\mathcal{O}$ is upset in $E \cdot S \cdot d_k \cdot c_{\overline{k}}(I)$, for some $k \in \{0,1\}$. Let P_ℓ, $\ell \in \{0,1\}$, be the process that does *not* take the first step in S, and $\hat{S}$ be the schedule obtained from S by moving the first step of P_ℓ to the beginning. By Lemma 8.10, $E \cdot \hat{S}$ is a node in $\mathcal{F}$, and each process and each object except $\mathcal{O}$ is in the same state in $E \cdot S(I)$ as in $E \cdot \hat{S}(I)$. Let $\hat{d}_k$ be any istep of P_k such that $E \cdot \hat{S} \cdot \hat{d}_k$ is a node in $\mathcal{F}$. We shall prove that

$$E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}} \text{ is a node in } \mathcal{F}, \text{ and } \mathcal{O} \text{ is upset in } E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}(I). \tag{5}$$

Given (5), process $P_{\overline{k}}$ and each object (including $\mathcal{O}$) is in the same state in $E \cdot S \cdot d_k \cdot c_{\overline{k}}(I)$ as in $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}(I)$. By Lemma 6.4, $E \cdot S \cdot d_k \cdot c_{\overline{k}}$ and $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}$ have the same valence. This contradicts Lemma 8.9, since P_ℓ takes the first step in $\hat{S}$, while $P_{\overline{\ell}}$ takes the first step in S.

It remains to show (5). There are three cases:

CASE 1. *$\mathcal{O}$ is upset in $E \cdot S(I)$*: Since S contains only ENROLL operations to $\mathcal{O}$, it must contain two ENROLL(0) operations to $\mathcal{O}$ (otherwise $\mathcal{O}$ would not be upset in $E \cdot S(I)$). Then so does $\hat{S}$, and $\mathcal{O}$ is upset in $E \cdot \hat{S}(I)$. It is immediate from the specification of booster, that $\mathcal{O}$ is upset in $E \cdot \hat{S} \cdot \hat{d}_k(I)$, and $c_{\overline{k}}$ is applicable to $E \cdot \hat{S} \cdot \hat{d}_k(I)$. Therefore, $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}$ is a node in $\mathcal{F}$, and $\mathcal{O}$ is upset in $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}(I)$ (because, once a booster object becomes upset, it remains upset forever).

CASE 2. *$\mathcal{O}$ is not upset in $E \cdot S(I)$ but it is upset in $E \cdot S \cdot d_k(I)$*: S contains exactly one ENROLL(0) and at least one ENROLL(1) step. (We have already argued, in the paragraph preceding Lemma 8.12, that S contains both ENROLL(0) and ENROLL(1) steps. It cannot contain two ENROLL(0) steps because, otherwise, $\mathcal{O}$ would be upset in $E \cdot S(I)$, contrary to the hypothesis of this case.) By assumption, S does not contain any REVEAL steps. Since $\mathcal{O}$ is fresh in $E(I)$ and it is not upset in $E \cdot S(I)$, the state of $\mathcal{O}$ in $E \cdot S(I)$ is $\langle V, v, \perp, \perp, \perp, j_{\overline{\ell}} \rangle$, where V is the set of return values of the ENROLL(1) steps in S, and v is the return value of the ENROLL(0) step in S. (To see why the last component of the state is $j_{\overline{\ell}}$, recall that $P_{\overline{\ell}}$ has an ENROLL$(j_{\overline{\ell}})$ operation to $\mathcal{O}$ pending in $E(I)$. Since $P_{\overline{\ell}}$ is the process that takes the first step in S, that step involves an ENROLL$(j_{\overline{\ell}})$ operation to $\mathcal{O}$. Since $\mathcal{O}$ is fresh in $E(I)$, when that ENROLL$(j_{\overline{\ell}})$ operation is applied to $\mathcal{O}$, the state of $\mathcal{O}$ is set to a state of the form $\langle *, *, \perp, \perp, \perp, j_{\overline{\ell}} \rangle$. The last component of the state is not changed, once it is set.) Similarly, the state of $\mathcal{O}$ in $E \cdot \hat{S}(I)$ is $\langle V, v, \perp, \perp, \perp, j_\ell \rangle$ — i.e., the same as in $E \cdot S(I)$, except for the last component.

Since $\mathcal{O}$ is upset in $E \cdot S \cdot d_k(I)$ it follows, from the specification of booster, that d_k is a REVEAL(i_k, v_k, u_k) step to $\mathcal{O}$ so that one of the following holds:

(a) $i_k = 0$ and $v_k \neq v$ (i.e., v_k is not the return value of the ENROLL(0) step in S);
(b) $i_k = 1$ and $v_k \notin V$ (i.e., v_k is not the return value of some ENROLL(1) step in S).

In either case, it is immediate from the specification of booster that the application of REVEAL(i_k, v_k, u_k) to $\mathcal{O}$ in state $\langle V, v, \perp, \perp, \perp, j_\ell \rangle$ will cause $\mathcal{O}$ to become upset (and can return any value in $\{0, 1\}$). Since the state of process P_k is the same in $E \cdot S(I)$ as is $E \cdot \hat{S}(I)$, $\hat{d}_k$ is also a REVEAL(i_k, v_k, u_k) step to $\mathcal{O}$. Thus, $\mathcal{O}$ is upset in $E \cdot \hat{S} \cdot \hat{d}_k(I)$. Hence, $c_{\overline{k}}$ is applicable to $E \cdot \hat{S} \cdot \hat{d}_k(I)$. This implies that $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}$ is a node in $\mathcal{F}$, and $\mathcal{O}$ is upset in $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}(I)$.

CASE 3. *$\mathcal{O}$ is not upset in $E \cdot S \cdot d_k(I)$ but it is upset in $E \cdot S \cdot d_k \cdot c_{\overline{k}}(I)$*: Let the REVEAL operation applied to $\mathcal{O}$ in d_k be REVEAL(i_k, v_k, u_k). Since $\mathcal{O}$ is not upset in $E \cdot S \cdot d_k(I)$, and S does not contain any REVEAL steps, we conclude the following about the state of $\mathcal{O}$ in $E \cdot S \cdot d_k(I)$:

- If $i_k = 0$ then that state is $\langle V, v, \perp, u_k, \perp, j_{\overline{\ell}} \rangle$, where V is the set of return values of the ENROLL(1) steps in S, v is the return value of the ENROLL(0) step in S, and $v_k = v$.
- If $i_k = 1$ then that state is $\langle V, v, v_k, \perp, u_k, j_{\overline{\ell}} \rangle$, where V is the set of return values of the ENROLL(1) steps in S, v is the return value of the ENROLL(0) step in S, and $v_k \in V$.

As argued in the preceding case, $\hat{d}_k$ is also a REVEAL(i_k, v_k, u_k) step to $\mathcal{O}$. Thus, the state of $\mathcal{O}$ in $E \cdot \hat{S} \cdot \hat{d}_k(I)$ is as above, except that the last component is j_ℓ instead of $j_{\overline{\ell}}$. Since $\mathcal{O}$ is upset in $E \cdot S \cdot d_k \cdot c_{\overline{k}}(I)$, from the specification of booster it follows that $c_{\overline{k}}$ is a REVEAL$(i_{\overline{k}}, v_{\overline{k}}, u_{\overline{k}})$ step to $\mathcal{O}$ so that one of the following holds:

(a) $i_{\overline{k}} = 0$ and $v_{\overline{k}} \neq v$ (i.e., $v_{\overline{k}}$ is not the return value of the ENROLL(0) step in S).
(b) $i_{\overline{k}} = 1$ and $v_{\overline{k}} \notin V$ (i.e., $v_{\overline{k}}$ is not the return value of some ENROLL(1) step in S).
(c) $i_{\overline{k}} = i_k = 0$.
(d) $i_{\overline{k}} = i_k = 1$, and either $v_{\overline{k}} \neq v_k$, or $u_{\overline{k}} \neq u_k$.

(e) $i_{\overline{k}} \neq i_k$ and it is not the case that $v_{\overline{k}} = v_k \Leftrightarrow u_{\overline{k}} \neq u_k$.

In each of these cases, it can be seen that the application of a REVEAL$(i_{\overline{k}}, v_{\overline{k}}, u_{\overline{k}})$ operation to $\mathcal{O}$ when in a state of the form $\langle V, v, \perp, u_k, \perp, j_\ell \rangle$ or $\langle V, v, v_k, \perp, u_k, j_\ell \rangle$ (the only possible states of $\mathcal{O}$ in $E \cdot \hat{S} \cdot \hat{d}_k(I)$) will cause $\mathcal{O}$ to become upset and can return any value in $\{0, 1\}$. Thus, $c_{\overline{k}}$ is applicable to $E \cdot \hat{S} \cdot \hat{d}_k(I)$. Therefore, $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}$ is a node in $\mathcal{F}$, and $\mathcal{O}$ is upset in $E \cdot \hat{S} \cdot \hat{d}_k \cdot c_{\overline{k}}(I)$.

This completes the proof of (5), and thereby of the lemma. □

Using Lemma 8.12, we can show that, starting from $E(I)$, if each of the two processes is about to apply its first REVEAL operation to $\mathcal{O}$, then the parameters of these two REVEAL operations satisfy certain properties.

LEMMA 8.13. *Consider any node $E \cdot S$ in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$ and both processes have a* REVEAL *operation to $\mathcal{O}$ pending in $E \cdot S(I)$. For each $k \in \{0, 1\}$, let i_k, v_k and u_k be the parameters of the* REVEAL *operation to $\mathcal{O}$ of P_k that is pending in $E \cdot S(I)$. Then,*

(a) For each $k \in \{0, 1\}$, v_k is the response of an ENROLL(i_k) *step to $\mathcal{O}$ contained in S.*

(b) i_0 and i_1 cannot be both equal to 0.

(c) If $i_0 = i_1 = 1$ then $v_0 = v_1$ and $u_0 = u_1$.

(d) If $i_0 \neq i_1$ then $v_0 = v_1 \Leftrightarrow u_0 \neq u_1$.

Proof. For each $k \in \{0, 1\}$, let d_k be a step of P_k and $c_{\overline{k}}$ be a step of $P_{\overline{k}}$ such that $E \cdot S \cdot d_k \cdot c_{\overline{k}}$ is a node in $\mathcal{F}$. By the specification of booster, if any of the properties (a)–(d) failed to hold, $\mathcal{O}$ would be upset in $E \cdot S \cdot d_k \cdot c_{\overline{k}}(I)$, for any $k \in \{0, 1\}$. This would contradict Lemma 8.12. Thus, (a)–(d) must all hold. □

8.2. Stage two of the proof. We now describe an algorithm that solves Equality Negation between two processes, Q_0 and Q_1, and uses one booster object fewer than $\mathbf{A}_{en}$. Intuitively, the algorithm works as follows: Each process Q_k simulates the actions of the corresponding process P_k in an execution of $\mathbf{A}_{en}$, and uses this execution to determine what value to decide. In this simulation, the processes are allowed to access all objects used by $\mathbf{A}_{en}$, except $\mathcal{O}$, each of which is initialised to the state it has in configuration $E(I)$.

The simulation of P_k by Q_k starts by pretending that P_k is in the state it has in configuration $E(I)$. Based on the current state of (the simulated) P_k, Q_k determines the operation *op* that P_k would apply next, and the object O to which *op* would be applied. If $O \neq \mathcal{O}$, then Q_k has access to O and can apply *op* to it directly to determine the response and update the (simulated) state of P_k accordingly. If, however, $O = \mathcal{O}$, this is not possible, since Q_k is not allowed to access $\mathcal{O}$. Instead, in this case, Q_k acts as follows:

- If *op* is an ENROLL(i) operation, P_k pretends that it applied this operation to $\mathcal{O}$ and received as the response the initial value for which Q_k must solve Equality Negation. In other words, Q_k pretends that when P_k enrolls in $\mathcal{O}$ it receives as the "challenge" with which to solve Equality Negation just the input value for which Q_k is required to solve Equality Negation! Q_k then updates the (simulated) state of P_k as if this step were executed and proceeds with the simulation of the next operation of P_k.
- If *op* is a REVEAL operation, Q_k does *not* simulate any more steps of $\mathbf{A}_{en}$. Instead, it decides a value based on the parameters of this REVEAL operation and the parameter of the first ENROLL operation to $\mathcal{O}$ that it previously simulated.

Shared: every object other than $\mathcal{O}$ used in algorithm $\mathbf{A}_{en}$,
each initialised to the state it has in $E(I)$
S : auxiliary variable : schedule of $\mathbf{A}_{en}$ applicable to $E(I)$, initially empty

Code for process Q_k, $k \in \{0, 1\}$

```
1   init := initial value of Q_k
2   state := state of P_k in E(I)
3   j := parameter of the ENROLL operation to O by P_k pending in E(I)
4   while Q_k has not decided do
5       ⟨op, O⟩ := NextOp(P_k, state)
6       if O ≠ O then
7           [[ res := Apply(P_k, op, O); S := S·(P_k, op, O, res) ]]
8           state := NextState(P_k, state, res)
9       else (* O = O *)
10          if op = ENROLL(i) then
11              S := S·(P_k, ENROLL(i), O, init)
12              state := NextState(P_k, state, init)
13          else
14              let i, v and u be such that op = REVEAL(i, v, u)
15              if i ≠ j then
16                  if v = init then decide ū
17                  else decide u
18              else
19                  if i = 0 then (* i = j = 0 *)
20                      decide u
21                  else (* i = j = 1 *)
22                      if v = init then decide u
23                      else decide ū
```

FIG. 8.2. *Solving Equality Negation for two processes using* $\mathbf{A}_{en}$ *with one fewer* booster *object.*

The algorithm is described more formally in Figure 8.2. A few remarks about the conventions used in the pseudocode are in order. Recall, from Figure 5.1, that Apply(P_k, op, O) is the procedure by which P_k invokes operation op on object O; it returns the result of this invocation (and updates the state of O accordingly). In addition, we assume that we are given two functions that describe the behaviour of processes P_0 and P_1 in $\mathbf{A}_{en}$ (these correspond to the functions ν and τ discussed when we formally defined processes in Section 2.2):

- NextOp(P_k, s): returns the pair $\langle op, O \rangle$, where the next operation that P_k executes in $\mathbf{A}_{en}$ when in state s is to apply op to object O.
- NextState(P_k, s, r): returns the state that P_k enters in $\mathbf{A}_{en}$ if it receives response r from the operation it invokes when in state s.

In the algorithm we also use an auxiliary variable S whose value, informally speaking, is the schedule of steps of $\mathbf{A}_{en}$ that have been simulated by Q_0 and Q_1 so far. This variable is not needed by the algorithm, but it is useful in proving its correctness (see Lemma 8.14). In one atomic step, besides accessing an ordinary (local or shared) variable, each process can also modify the auxiliary variable S. To emphasise this, we

bracket with "$[\![\cdots]\!]$" the line that corresponds to a single atomic step affecting both an ordinary variable and the auxiliary variable S (line 7).

LEMMA 8.14. *If $\mathbf{A}_{en}$ is an Equality Negation algorithm for processes P_0 and P_1, then the algorithm in Figure 8.2 also solves Equality Negation for processes Q_0 and Q_1 using one fewer* booster *object than $\mathbf{A}_{en}$.*

Proof. It is obvious from Figure 8.2 that the algorithm uses one booster object less than $\mathbf{A}_{en}$. In the remainder of this proof we show that the algorithm solves Equality Negation for Q_0 and Q_1. To this end, fix an arbitrary execution of the algorithm. We shall prove that in this execution, the three properties of Equality Negation — Termination, Negation and Agreement — are satisfied.

First we establish some properties of this execution. Let S_i be the value of the auxiliary variable S when the schedule it contains consists of exactly i steps. By lines 7 and 11 of Figure 8.2, it is clear that for all i, j, where $0 \leq i \leq j$, if S_j is defined, then S_i is a prefix of S_j. A straightforward induction on i shows that for all $i \geq 0$ such that S_i is defined, the following invariants hold.

(a) S_i is a schedule of $\mathbf{A}_{en}$ that is applicable to $E(I)$ (hence, $E \cdot S_i$ is a node in $\mathcal{F}$) and contains no REVEAL step to $\mathcal{O}$.

(b) Let Q_k be the process that assigns S_i to S, and $state_k$ be the value to which Q_k sets its local variable *state* after assigning S_i to S (cf. lines 8 and 12). Then, for all $j \geq i$, such that S_j is defined and there is no step of P_k in S_j after S_i, $state_k$ is the state of P_k in configuration $S_j(I)$ of $\mathbf{A}_{en}$.

With these invariants, we are now ready to prove that the execution satisfies the three properties of Equality Negation.

Since only two processes execute the algorithm, the Agreement requirement of Equality Negation is trivially satisfied. We now show that the execution satisfies Termination. Suppose, for contradiction, that some correct process Q_k is correct and never decides. This means that the **while** loop of Q_k does not terminate, and hence S_i is defined for all integers $i \geq 0$. By Invariant (a), S_i is applicable to $E(I)$ and contains no REVEAL step to $\mathcal{O}$, for all $i \geq 0$. Thus, there is an infinite schedule S^* of $\mathbf{A}_{en}$ that is applicable to $E(I)$, contains infinitely many steps of P_k, and does not contain a REVEAL step to $\mathcal{O}$. By Lemma 8.11, P_k does not decide in $E \cdot S'(I)$, for all prefixes S' of S^*. This contradicts the fact that $\mathbf{A}_{en}$ satisfies the Termination property. Thus, the execution satisfies Termination.

It remains to show that the execution satisfies the Negation property. Assume that both Q_0 and Q_1 decide in the execution. (Otherwise, Negation is trivially satisfied.) For each $k \in \{0, 1\}$, let $init_k$ be the initial value of process Q_k (i.e., the value assigned to local variable *init* of Q_k in line 1); j_k be the value assigned to local variable j of process Q_k in line 3; and $state_k$ be the value of local variable *state* of process Q_k when Q_k decides. Let S^* be the value of S when Q_0 and Q_1 have both decided (i.e., S^* is the final value of the auxiliary variable S). By the algorithm, the next operation that P_k will execute when in $state_k$ is a REVEAL operation to object $\mathcal{O}$; let i_k, v_k and u_k be the parameters of that REVEAL operation. From Invariant (b), it follows that the operation of P_k pending in $E \cdot S^*(I)$ is a REVEAL(i_k, v_k, u_k) to $\mathcal{O}$. By Invariant (a), $E \cdot S^*$ is a node in $\mathcal{F}$, and S^* contains no REVEAL operations to $\mathcal{O}$. Thus, the hypothesis of Lemma 8.13 holds (with $S = S^*$); we will make use of parts (a)–(d) of this lemma below. By line 11 of the algorithm, the only return values of ENROLL operations to $\mathcal{O}$ that appear in S^* are $init_0$ and $init_1$. Thus, by Lemma 8.13(a), $v_0, v_1 \in \{init_0, init_1\}$. There are three cases:

CASE 1. *For both $k \in \{0, 1\}$, $i_k \neq j_k$*: In this case, each Q_k decides by line 16 or 17

(depending on whether $v_k = init_k$). Also, $i_0 \neq i_1$ (because $i_0, i_1, j_0, j_1 \in \{0,1\}$, and, by Lemma 8.8, $j_0 \neq j_1$). Thus, by Lemma 8.13(d), $u_0 = u_1$ if and only if $v_0 \neq v_1$.

By a case analysis of the four possible combinations of lines by which Q_0 and Q_1 can decide, we have that Q_0 and Q_1 decide the same value if and only if

$$\begin{array}{rll} & (v_0 = init_0 \textbf{ and } v_1 = init_1 \textbf{ and } u_0 = u_1) & /* \ Q_0\text{'s line 16; } Q_1\text{'s line 16 } */ \\ \textbf{or} & (v_0 = init_0 \textbf{ and } v_1 \neq init_1 \textbf{ and } u_0 \neq u_1) & /* \ Q_0\text{'s line 16; } Q_1\text{'s line 17 } */ \\ \textbf{or} & (v_0 \neq init_0 \textbf{ and } v_1 = init_1 \textbf{ and } u_0 \neq u_1) & /* \ Q_0\text{'s line 17; } Q_1\text{'s line 16 } */ \\ \textbf{or} & (v_0 \neq init_0 \textbf{ and } v_1 \neq init_1 \textbf{ and } u_0 = u_1). & /* \ Q_0\text{'s line 17; } Q_1\text{'s line 17 } */ \end{array}$$

Since $u_0 = u_1$ if and only if $v_0 \neq v_1$, the above expression is equivalent to

$$\begin{array}{rl} & (v_0 = init_0 \textbf{ and } v_1 = init_1 \textbf{ and } v_0 \neq v_1) \\ \textbf{or} & (v_0 = init_0 \textbf{ and } v_1 \neq init_1 \textbf{ and } v_0 = v_1) \\ \textbf{or} & (v_0 \neq init_0 \textbf{ and } v_1 = init_1 \textbf{ and } v_0 = v_1) \\ \textbf{or} & (v_0 \neq init_0 \textbf{ and } v_1 \neq init_1 \textbf{ and } v_0 \neq v_1). \end{array}$$

As argued above, $v_0, v_1 \in \{init_0, init_1\}$. Thus, the above disjunction is equivalent to $init_0 \neq init_1$. In other words, Q_0 and Q_1 decide the same value if and only if they have different initial values.

Case 2. *For both $k \in \{0,1\}$, $i_k = j_k$*: By Lemma 8.8, $j_0 \neq j_1$. Thus, for some $k \in \{0,1\}$, $i_k = j_k = 0$, and $i_{\overline{k}} = j_{\overline{k}} = 1$. Then Q_k decides by line 20, and $Q_{\overline{k}}$ decides by line 22 or 23 (depending on whether $v_{\overline{k}} = init_{\overline{k}}$). Thus, Q_k decides u_k. So we have that Q_0 and Q_1 decide the same value if and only if

$$\begin{array}{rll} & (v_{\overline{k}} = init_{\overline{k}} \textbf{ and } u_k = u_{\overline{k}}) & /* \ Q_{\overline{k}}\text{'s line 22 } */ \\ \textbf{or} & (v_{\overline{k}} \neq init_{\overline{k}} \textbf{ and } u_k = \overline{u}_{\overline{k}}). & /* \ Q_{\overline{k}}\text{'s line 23 } */ \end{array}$$

In this case, $i_k \neq i_{\overline{k}}$. By Lemma 8.13(d), $u_k = u_{\overline{k}}$ if and only if $v_k \neq v_{\overline{k}}$. Thus, the above expression is equivalent to

$$(v_{\overline{k}} = init_{\overline{k}} \textbf{ and } v_k \neq v_{\overline{k}}) \textbf{ or } (v_{\overline{k}} \neq init_{\overline{k}} \textbf{ and } v_k = v_{\overline{k}}).$$

S^* does not contain two ENROLL(0) steps: Otherwise, $\mathcal{O}$ would be upset in $E \cdot S^*(I)$ — and therefore still upset after the application of the two REVEAL operations pending in $E \cdot S^*(I)$ — contrary to Lemma 8.12. Since $j_k = 0$, P_k has an ENROLL(0) to $\mathcal{O}$ pending in $E(I)$. Hence, the unique ENROLL(0) to $\mathcal{O}$ in S^* is a step of P_k. By line 11, the response to that unique ENROLL(0) step to $\mathcal{O}$ in S^* is the initial value of P_k, $init_k$. By Lemma 8.13(a) and the fact that $i_k = 0$, v_k is the response of the unique ENROLL(0) step to $\mathcal{O}$ that is contained in S^*. Therefore, $v_k = init_k$. Thus, the above expression is equivalent to

$$(v_{\overline{k}} = init_{\overline{k}} \textbf{ and } init_k \neq v_{\overline{k}}) \textbf{ or } (v_{\overline{k}} \neq init_{\overline{k}} \textbf{ and } init_k = v_{\overline{k}}).$$

As argued earlier, $v_{\overline{k}} \in \{init_0, init_1\}$; so the above expression is equivalent to $init_k \neq init_{\overline{k}}$. In other words, Q_0 and Q_1 decide the same value if and only if they have different initial values.

Case 3. *For some $k \in \{0,1\}$, $i_k \neq j_k$, and $i_{\overline{k}} = j_{\overline{k}}$*: By Lemma 8.8, $j_k \neq j_{\overline{k}}$, so $i_k = i_{\overline{k}}$. By Lemma 8.13(b), i_k and $i_{\overline{k}}$ cannot both be 0. Hence, we must have $i_k = i_{\overline{k}} = j_{\overline{k}} = 1$ and $j_k = 0$. By Lemma 8.13(c), $v_0 = v_1$ and $u_0 = u_1$. Also, since in this case $i_k \neq j_k$, while $i_{\overline{k}} = j_{\overline{k}} = 1$, Q_k decides by line 16 or 17 (depending on whether $v_k = init_k$), and $Q_{\overline{k}}$ decides by line 22 or 23 (depending on whether $v_{\overline{k}} = init_{\overline{k}}$). Keeping in mind that $u_0 = u_1$, we have that Q_0 and Q_1 decide the same value if and only if

$$\begin{array}{rll} & (v_k = init_k \textbf{ and } v_{\overline{k}} \neq init_{\overline{k}}) & /* \ Q_k\text{'s line 16; } Q_{\overline{k}}\text{'s line 23 } */ \\ \textbf{or} & (v_k \neq init_k \textbf{ and } v_{\overline{k}} = init_{\overline{k}}). & /* \ Q_k\text{'s line 17; } Q_{\overline{k}}\text{'s line 22 } */ \end{array}$$

Since $v_0 = v_1$ and, as argued earlier, $v_0, v_1 \in \{init_0, init_1\}$, the above expression is equivalent to $init_k \neq init_{\overline{k}}$. Hence, Q_0 and Q_1 decide the same value if and only if they have different initial values.

In all cases the Negation property is satisfied. □

We now have all the pieces needed to prove the main result of this section.

THEOREM 8.15. *Type* booster *has Consensus number one.*

Proof. Suppose, to the contrary, that booster has Consensus number at least two. Then, we can solve Consensus for two processes using only booster objects and registers. By the universality of Consensus [6], we can implement a negation object shared by two processes using only booster objects and registers. Hence, we can solve Equality Negation for two processes using only booster objects and registers. Let $\mathbf{A}_{en}$ be an Equality Negation algorithm for two processes that uses a minimal number of booster objects. By Theorem 3.1, $\mathbf{A}_{en}$ uses at least one booster object. By Lemma 8.14, however, we can solve Equality Negation for two processes using one booster object less than $\mathbf{A}_{en}$. This contradicts the definition of $\mathbf{A}_{en}$. □

9. Conclusion. In this paper we showed that if nondeterministic object types are allowed, then the only robust wait-free hierarchy is the trivial one, which lumps all types into level one. In contrast, if only deterministic types are allowed, then the Consensus hierarchy is robust [1, 15]. Our understanding of the results regarding deterministic types is far from complete. In view of the sharply contrasting results regarding the robustness of wait-free hierarchies for deterministic and nondeterministic types it would be interesting to understand the fundamental reason(s) why the restriction to determinism inherently rules out types with properties like negation and booster.

The unfortunate fact that (nontrivial) wait-free hierarchies are not robust suggests that it would be desirable to investigate other ways of classifying types according to their strength. A classification based on Consensus numbers is computability-based: it depends on whether objects of a certain type can be used to solve Consensus for a certain number of processes, with no regard as to the efficiency of this solution. It would be interesting to investigate complexity-based classifications, where the level of a type may depend not only on its ability to solve Consensus but also on doing so efficiently (in some appropriate sense). Could a complexity-based hierarchy classify types according to their strength in (efficiently) implementing other types, while being robust in the sense that "powerful" types are not (efficiently) implementable by "weak" ones?

A different line of research might proceed from the observation that the types negation and booster, although well within the set prescribed by the general definition of type given in Section 2.1, are not "natural" types. Is there a sensible definition of "natural" types that excludes negation, booster and other such "anomalous" types, without excluding any useful ones?

Acknowledgements. Our thinking on shared-memory distributed computing has been deeply influenced from discussions with Tushar Chandra, Prasad Jayanti and Sam Toueg. Faith Fich and the anonymous referees made many helpful comments on earlier versions of this paper. Patricia O'Brien suggested the title.

REFERENCES

[1] Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 363–372, August 1994.

[2] Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom versus t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 334–343, August 1994.

[3] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, August 1987.

[4] Edger Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8:569, 1965.

[5] 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, April 1985.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[8] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 145–158, August 1993.

[9] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, pages 69–84, November 1992.

[10] Jon Kleinberg and Sendhil Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 133–144, August 1993.

[11] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Massachusetts, 1986.

[12] Wai-Kau Lo. More on t-resilence vs. wait-freedom. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 110–119, August 1995.

[13] Michael Loui and Hosame Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computer Research*, volume 4, pages 163–183. JAI Press Inc., 1987.

[14] Shlomo Moran and Lihu Rappoport. On the robustness of h^r_m. In *Proceedings of the Tenth International Workshop on Distributed Algorithms*, October 1996.

[15] Gary Peterson, Rida Bazzi, and Gil Neiger. A gap theorem for consensus types. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 344–353, August 1994.

[16] Serge Plotkin. Sticky bits and the universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.

[17] Ophir Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the Eighth International Workshop on Distributed Algorithms*, pages 156–163, September 1994.

[18] Eric Ruppert. Determining Consensus numbers. Technical Report 303/96, Department of Computer Science, University of Toronto, November 1996.

[19] Eric Schenk. *Computability and Complexity Results for Agreement Problems in Shared Memory Distributed Systems.* PhD thesis, University of Toronto, 1996.

[20] Eric Schenk. The consensus hierarchy is not robust (Brief Announcement). In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing*, page 279, August 1997.

Appendix A. Unsolvability of Equality Negation using only registers.

In this appendix we prove Theorem 3.1, which states that there is no algorithm that solves Equality Negation for two processes and uses only registers. The proof is by contradiction. Suppose that there is an Equality Negation algorithm, $\mathbf{A}_{en}$, for processes P_0 and P_1 that uses only registers. For each $k \in \{0,1\}$ and $v \in \{0,1,2\}$, consider a solo execution of process P_k in algorithm $\mathbf{A}_{en}$, where the initial value of P_k is v. Since $\mathbf{A}_{en}$ uses only registers (which are deterministic), the solo execution of P_k with initial value v is uniquely defined. Let Δ_k^v be the decision of P_k in this solo execution.

Shared: all registers used in $\mathbf{A}_{en}$, each initialised as specified by $\mathbf{A}_{en}$
R_0, R_1: register, each initialised to $\perp$

Code for process Q_k, $k \in \{0, 1\}$
1 $R_k :=$ initial value of Q_k
2 $w :=$ Execute$(P_k, \mathbf{A}_{en}, \alpha_k)$
3 **if** $w = \Delta_k^{\alpha_k}$ **then decide** R_k
4 **else decide** $R_{\overline{k}}$

FIG. A.1. *Solving Consensus for processes Q_0 and Q_1 using $\mathbf{A}_{en}$.*

LEMMA A.1. *There exist $\alpha_0, \alpha_1 \in \{0, 1, 2\}$ such that $\alpha_0 = \alpha_1$ if and only if $\Delta_0^{\alpha_0} = \Delta_1^{\alpha_1}$.*

Proof. Suppose, for contradiction, that no choice of values for $\alpha_0, \alpha_1 \in \{0, 1, 2\}$ satisfies the required property. Thus, (a) for all $v \in \{0, 1, 2\}$, $\Delta_0^v \neq \Delta_1^v$; and (b) for all $v_0, v_1 \in \{0, 1, 2\}$ such that $v_0 \neq v_1$, $\Delta_0^{v_0} = \Delta_1^{v_1}$. By (b), $\Delta_0^0 = \Delta_1^1$, $\Delta_1^1 = \Delta_0^2$, and $\Delta_0^2 = \Delta_1^0$. Therefore, $\Delta_0^0 = \Delta_1^0$, which contradicts (a). □

As the lemma below proves, the algorithm in Figure A.1 solves Consensus for processes Q_0 and Q_1. Since this algorithm uses two registers in addition to the objects used by $\mathbf{A}_{en}$, and since $\mathbf{A}_{en}$ uses only regusters, we have a Consensus algorithm for two processes that uses only registers. This contradicts the fact that no such algorithm exists [3, 13]. Therefore, $\mathbf{A}_{en}$ cannot use only registers, as wanted.

LEMMA A.2. *The algorithm in Figure A.1 solves Consensus for processes Q_0 and Q_1.*

Proof. The algorithm in Figure A.1 satisfies Termination, since $\mathbf{A}_{en}$ does.

To prove that the algorithm satisfies Validity, it suffices to show that if process Q_k decides the value in register R_d in an execution of the algorithm in Figure A.1, then Q_d has previously written its initial value into R_d. (Note that R_d is written only once.) This property follows immediately if Q_k decides the value in R_k in line 3. It remains to cover the case where Q_k decides in line 4. In this case, we want to prove that before Q_k decides $R_{\overline{k}}$ in line 4, $Q_{\overline{k}}$ has previously written its initial value into $R_{\overline{k}}$ (line 1). For this, it suffices to show that the execution of P_k that Q_k simulated in line 2 is not a solo P_k execution of $\mathbf{A}_{en}$. (If the simulated execution is not a solo P_k execution, Q_k and $Q_{\overline{k}}$ execute their line 2 concurrently, and therefore $Q_{\overline{k}}$ executes line 1 before Q_k executes line 4.) To see why the execution of P_k that Q_k simulated is not a solo P_k execution, suppose the contrary. Then, by definition of $\Delta_k^{\alpha_k}$, P_k would have decided $\Delta_k^{\alpha_k}$ in that execution, and the value that Q_k assigned to its local variable w in line 2 would have been equal to $\Delta_k^{\alpha_k}$. But then Q_k would be deciding in line 3, not 4.

To prove that the algorithm satisfies Agreement, consider any execution in which both Q_0 and Q_1 decide. Let w_0 and w_1 be the values of the local variables w of Q_0 and Q_1, respectively, assigned in line 2 during that execution. Consider the execution of algorithm $\mathbf{A}_{en}$ by P_0 and P_1 that Q_0 and Q_1 simulated in line 2. In this simulated execution P_0 and P_1 have initial values α_0 and α_1 and decide w_0 and w_1. By the Negation property, $w_0 = w_1$ if and only if $\alpha_0 \neq \alpha_1$. Combining with Lemma A.1, we have $w_0 = w_1$ if and only if $\Delta_0^{\alpha_0} \neq \Delta_1^{\alpha_1}$. Since $w_0, w_1, \Delta_0^{\alpha_0}, \Delta_1^{\alpha_1} \in \{0, 1\}$, it follows that $w_0 = \Delta_0^{\alpha_0}$ if and only if $w_1 \neq \Delta_1^{\alpha_1}$. By the algorithm in Figure A.1, for any $k \in \{0, 1\}$, Q_k decides the value in R_k in line 3 if and only if $w_k = \Delta_k^{\alpha_k}$ and hence

if and only if $w_{\overline{k}} \neq \Delta_{\overline{k}}^{\alpha_{\overline{k}}}$. This, in turn, happens if and only if $Q_{\overline{k}}$ decides the value in $R_{\overline{\overline{k}}} = R_k$ in line 4. This means that both Q_0 and Q_1 decide the value in the same register R_d, for some $d \in \{0, 1\}$. By Validity, a process decides the value in R_d only after Q_d has written its initial value into R_d. Since only one value is written into R_d, Agreement follows. □