Distrib. Comput. (1997) 10: 117–127

DISTRIBUTED COMPUTING

On the use of registers in achieving wait-free consensus*

Rida A. Bazzi[1], **Gil Neiger**[2], **Gary L. Peterson**[3]

[1] Computer Science and Engineering Department, College of Engineering and Applied Sciences, Arizona State University, P.O. Box 875406, Tempe, AZ 85287-5406, USA
[2] Intel Corporation, JF3-359, 2111 N.E. 25th Avenue, Hillsboro, OR 97124-5964, USA
[3] Computer and Information Science Program, Spelman College, 350 Spelman Lane SW, P.O. Box 333, Atlanta, GA 30314-0339, USA

Summary. The computational power of concurrent data types has been the focus of much recent research. Herlihy showed that such power may be measured by the type's ability to implement wait-free consensus. Jayanti argued that this ability could be measured in different ways, depending, for example, on whether or not read/write registers could be used in an implementation. He demonstrated the significance of this distinction by exhibiting a nondeterministic type whose ability to implement consensus was increased with the availability of registers. We show that registers cannot increase the ability to implement wait-free consensus of any deterministic type or of any type that can, without them, implement consensus for at least two processes. These results significantly impact the study of the wait-free hierarchies of concurrent data types. In particular, the combination of these results with other recent work suggests that Jayanti's h_m hierarchy is robust for certain classes of deterministic types.

Key words: Registers – Consensus – Wait-free computation – Wait-free hierarchies – Robustness

1 Introduction

Achieving consensus in the presence of process failures is of fundamental importance in distributed computing. A large body of research has studied algorithms for achieving consensus in three domains: (1) synchronous message-passing systems, (2) asynchronous message-passing systems, and (3) asynchronous read/write memory systems. While the first domain has produced a large number of deterministic algorithms, it has been shown that such algorithms do not exist in the other two [6, 8, 10, 11, 20]. Because of these results, researchers also consider algorithms for consensus in asynchronous shared-object systems with primitives more powerful than simple reads and writes [1, 2, 7, 11, 14–18, 20, 24, 26].

Another reason for taking this approach stems from the study of wait-free implementations of concurrent data types. Here, researchers ask questions such as the following: "is there a wait-free implementation of type T_1 using objects of type T_2?" A concurrent implementation of a data type is *wait-free* if any process can complete any operation of the implementation in a finite number of its own steps regardless of the behavior and speed of other processes. Wait-free implementations are desirable in asynchronous systems because they prevent slow processes from slowing down faster ones. In addition, they can tolerate any number of stopping failures. Herlihy [11] showed a direct connection between a type's ability to implement wait-free consensus (i.e., provide an implementation of consensus that is wait-free) and its ability to provide wait-free implementations of other types. In particular, he showed that consensus is *universal*: for any $n > 0$, if type T can implement wait-free consensus in systems with n processes, then T can provide a wait-free implementation of any type in such systems. In light of this result, Herlihy evaluated the power of a data type by assigning it a *consensus number*; this is the maximum number of processes for which the type can be used to implement wait-free consensus. He thus cast the universe of concurrent data types into a hierarchy, each level of which contains types with a particular consensus number.

Jayanti [14] refined this study by asking the following question: what does it mean to say that a type can implement wait-free consensus? He argued that an answer required addressing the following questions.

1. Can more than one object of the type be used in the implementation?
2. Can read/write registers (also called read/write memory) also be used in the implementation?

Because these questions can be answered together in four different ways, Jayanti identified four possible hierarchies of types, one of which corresponds to Herlihy's assignment

* The first two authors were supported in part by the National Science Foundation under grants CCR-9106627 and CCR-9301454. The second author was supported in part by a scholarship from the Hariri Foundation. The third author was supported in part by the W.F. Kellogg Foundation under a grant to the Center for Scientific Applications of Mathematics at Spelman College

Correspondence to: G. Neiger

of consensus numbers (answering "no" to question 1 and "yes" to 2). He called these h_1, h_1^r, h_m, and h_m^r. A subscript "1" indicates that only one object of a type can be used, while a subscript "m" indicates that many can be used. A superscript "r" indicates that registers may be used, while its absence indicates that they may not. Jayanti indicated that Herlihy's hierarchy is h_1^r.[1]

Given these four hierarchies, Jayanti naturally asked if they were distinct and, if so, which best measured the computational power of different data types. He argued that a hierarchy does not properly measure this power if it is not robust. Informally, a hierarchy is *robust* if no collection of types at low levels can implement a type at a higher level. Jayanti showed that none of h_1, h_1^r, and h_m could be robust if it were not equal to h_m^r. He then showed that both h_1^r and h_m were different from h_m^r, proving that only h_m^r might be robust (h_1 cannot equal h_m^r if either h_m or h_1^r does not). Jayanti left the robustness of h_m^r as an open question.

Recall that Jayanti's hierarchy h_m was defined by answering "yes" to question 1 above and "no" to question 2; it differs from h_m^r on whether or not registers may be used in implementations of consensus. Jayanti proved $h_m \neq h_m^r$ (and h_m to not be robust) by exhibiting a type that was at different levels in the two hierarchies. This type was specified *nondeterministically*; that is, there is (at least) one sequence of operations on the type for which more than one behavior is possible. This raises an obvious question: can the same result be shown with a deterministic type? Since most commonly used concurrent data types are deterministic, a positive answer to this question would imply that the non-robustess of h_m would hold even for the restricted class of deterministic types.

We answer this question negatively. That is, we show that the two hierarchies give equal values for any deterministic type. Thus, the nondeterminism used by Jayanti is necessary. We also demonstrate other results relevant to the use of registers in implementing wait-free consensus. For all types (even nondeterministic ones), the two hierarchies can differ only at the first level: if h_m assigns a type a value greater than 1, then h_m^r assigns it the same value.

These results confirm that, in most cases, registers do not play a special role in achieving wait-free consensus. Other papers [3, 24] have claimed h_m^r to be robust for certain classes of deterministic types. Combined with the results of this paper, those results would also imply that h_m is also robust for these types.

Our results are proven through the introduction of a new concurrent data type called the *one-use bit.* An object of this type is a bit that can be read at most once and written at most once. Our main results stem from the following facts.

- A finite number of one-use bits can implement a read/write register in a wait-free implementation of consensus (this is shown in Sect. 4).
- Almost any type can be used to implement a one-use bit (this is shown in Sect. 5).

These results show that almost any type can be used to implement read/write registers in a wait-free implementation of consensus. Thus, the availability of registers does not increase the ability of such a type to implement consensus if one is allowed multiple objects of the type. The types that cannot implement one-use bits are so weak that they cannot implement consensus with or without the aid of registers.

[1] The reader should note that, while h_1^r matches Herlihy's definitions, his impossibility proofs apply to h_m^r. Because Jayanti's work was the first to demonstrate types requiring multiple objects to solve consensus, the distinction was not made earlier

2 Background

This section presents the definitions and background material necessary to present and interpret the results of this paper.

2.1 Types

We define a concurrent data type with an automata-based definition. Processes interact with an object of a type by invoking accesses on the object's *ports* and receiving responses on those ports. This section presents formal definitions of these concepts.

A *type* is a 5-tuple $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$. The components are n, the number of ports the type has (this limits the number of processes that may access the type); Q, a (possibly infinite) set of *states*; I, a set of access *invocations*; R, a set of access *responses*; and δ, a transition function. Such a type is called an *n-ported type.* An *object* of type T (also called a T-*object*) is an instance of T that specifies, for each port, which processes (if any) access the object through that port. Let N_n be $\{1, 2, \ldots, n\}$. Type T may be either *deterministic*, in which case $\delta : Q \times N_n \times I \mapsto Q \times R$, or *nondeterministic*, in which case $\delta : Q \times N_n \times I \mapsto 2^{Q \times R}$. In keeping with traditional automata theory [13], we assume that, for any nondeterministic type T, $\delta(q, j, i)$ is finite for all q, j, and i. This specification of a type indicates how processes may access an object of type T (using invocations in I), how the object communicates to processes (using responses in R), and what are the legal sequential histories of the type (specified by δ). If an object of type T is in state q when invocation $i \in I$ appears on port $j \in N_n$, then the object changes to state q' and returns response r over port j if and only if $\langle q', r \rangle = \delta(q, j, i)$ (if T is deterministic) or $\langle q', r \rangle \in \delta(q, j, i)$ (if T is nondeterministic).

A type is *oblivious* if, for all $q \in Q, j_1, j_2 \in N_n$, and $i \in I$, $\delta(q, j_1, i) = \delta(q, j_2, i)$. An oblivious type does not distinguish identical accesses on different ports. For oblivious types, we often abuse notation and omit the second (port number) input to the transition function. For non-oblivious types, we require that at most one process is allowed to access each port of an object; other researchers [3] have made other assumptions. It is also traditional to require that each process is allowed to access at most one port of an object [5]; we do not require this.

Invocation i on port j is *useless* if there is some response r such that, for all states q, $\delta(q, j, i) = \langle q, r \rangle$ (if T is deterministic) or $\delta(q, j, i) = \{\langle q, r \rangle\}$ (if T is nondeterministic). Useless invocations are sometimes needed to specify some types completely, and we do not provide implementations for them in the sequel.

An *operation* on type $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ is an element of $N_n \times I \times R$, operation $\langle j, i, r \rangle$ representing the execution of invocation i on port j with response r being returned. A *sequential history of* T *from a state* q_0 is a sequence of alternating states and operations (starting with q_0) meeting certain conditions. In particular, consider the sequence

$$H = q_0; \langle j_1, i_1, r_1 \rangle; q_1; \langle j_2, i_2, r_2 \rangle; q_2; \ldots.$$

It must be that, for all k, $\langle q_k, r_k \rangle = \delta \langle q_{k-1}, j_k, i_k \rangle$ (if T is deterministic) or $\langle q_k, r_k \rangle \in \delta(q_{k-1}, j_k, i_k)$ (if T is nondeterministic). We say that state q' *is reachable from* q if q' appears in some sequential history from q. If H contains k operations, then the *length* of H, denoted $|H|$, is k. We define $invs(H, j)$ to be the sequence of invocations in H on port j and $resps(H, j)$ to be the sequence of responses returned to port j.

2.2 Implementations

This section defines what it means for one type to be implemented by others. Informally, an implementation is a set of objects (appropriately initialized) and deterministic programs that operate on these objects. There is one program for each process and for each invocation for the type being implemented. More formally, let $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ and let $S = \{O_1, O_2, \ldots, O_m\}$ be a set of objects such that O_j is of type $\mathsf{T}_j = \langle n_j, Q_j, I_j, R_j, \delta_j \rangle$. An *implementation of* T *from state* $q \in Q$ *from* S is a tuple of initial states $\langle q_1, q_2, \ldots, q_m \rangle$ $(q_j \in Q_j)$ and a deterministic program $P_{k,l}$ for each $i_k \in I$ and each $l \in N_n$. Each program of the implementation specifies how the implementing objects are to be accessed and what response should be returned to the invocation associated with that program. The implementation also specifies, for each port of each implementing object O_j, the corresponding port number of T. If port l of T corresponds to port l_j of O_j, this means that, when a program $P_{k,l}(i_k \in I)$ accesses O_j, it does so through port l_j. We require that each port of each O_j correspond to at most one port of T. It is also traditional to require that each port of T correspond to at most one port of each O_j, although we do not require this.

There is *an implementation of* T *from* S if implementations exist from all states of T. Such an implementation is correct if all resulting histories are *wait-free* [11] and *linearizable* [12]. By wait-free, we mean that, in all histories of the implementation, any process performing an infinite number of steps completes every implementing program that it begins. By linearizable, we mean that each execution of the implementation must be equivalent to a sequential history of the type. There must be a linear ordering of the implemented operations (i.e., port-invocation-response triples) in the execution such that (1) the ordering is that of a sequential history from the appropriate state and (2) if the execution of two implemented operations does not overlap in real time, then they appear in the sequential history in their real-time order. (For further details of these definitions, consult Herlihy [11], Herlihy and Wing [12], or Jayanti [14].)

2.3 Some specific types

This section defines some specific types that are used in the sequel.

2.3.1 Consensus types

The ability of a type to solve consensus is central to this paper. We define consensus as a type and consider the ability of different types to implement a consensus object (see Sect. 2.2 below). The *n-process binary consensus type* cons_n is an oblivious type defined to be $\langle n, Q, I, R, \delta \rangle$, where $Q = \{\bot, 0, 1\}$, $I = \{i_0, i_1\}$, $R = \{0, 1\}$, and δ is defined as follows:

$$\delta(\bot, i_0) = \langle 0, 0 \rangle \qquad \delta(a, i_b) = \langle a, a \rangle \text{ for any } a, b \in \{0, 1\}$$
$$\delta(\bot, i_1) = \langle 1, 1 \rangle$$

(We can specify δ without regard to port numbers as cons_n is oblivious.) Usually, consensus objects are chosen to have state $\bot$ initially. A process proposes 0 (respectively, 1) to a cons_n-object by invoking i_0 (respectively, i_1). Note that the first invocation on the object determines all future responses, which are identical. This response is sometimes called the *consensus value* of the object.

If there is an implementation of cons_n from S, we say that *S implements n-process consensus.* Note that it is trivial to implement cons_n from any state except $\bot$ so, in the sequel, we will equate an implementation of cons_n with an implementation from $\bot$.

2.3.2 Registers

Another type important in this paper is the *single-reader, single-writer bit.* Note that, because such a bit has only one writing process, that process always knows the value of the bit and thus needs to write to it only to change it from 0 to 1 or from 1 to 0. We call this action a "flip", and it will be used in place of "write" below. Formally, the single-reader, single-writer bit is a 2-ported type called $\mathsf{bit}_{\mathrm{mu}}$ (the "mu" indicates that it can be used a multiple number of times and distinguishes it from one-use bits defined in Sect. 3 below) and is defined to be $\langle 2, Q_{\mathrm{mu}}, I_{\mathrm{mu}}, R_{\mathrm{mu}}, \delta_{\mathrm{mu}} \rangle$, where $Q_{\mathrm{mu}} = \{0, 1\}$, $I_{\mathrm{mu}} = \{read, flip\}$, $R_{\mathrm{mu}} = \{0, 1, ok\}$ and δ_{mu} is defined as follows, where $v \in \{0, 1\}$.

$$\delta_{\mathrm{mu}}(v, 1, read) = \langle v, v \rangle \quad \delta_{\mathrm{mu}}(v, 2, read) = \langle v, ok \rangle$$

$$\delta_{\mathrm{mu}}(v, 1, flip) = \langle v, ok \rangle \quad \delta_{\mathrm{mu}}(v, z, flip) = \langle 1 - v, ok \rangle$$

The process connected to port 1 (the reader) of a $\mathsf{bit}_{\mathrm{mu}}$-object can discover the state of the object using a *read* invocation, while the process connected to port 2 (the writer) can change the state using a *flip* invocation. Note that *flip* invocations on port 1 and *read* invocations on port 2 are useless.

Many papers (including this one) use the term "register" or "read/write memory" to refer to a similar type that is multi-reader, multi-writer, and multi-valued. We call such a type a *general register* and denote it by reg. We do not provide a formal description of this extension; it is similar to $\mathsf{bit}_{\mathrm{mu}}$, except that it is oblivious (any port can be used either to read or to write the register) and can hold

m different values instead of just 2. Instead of *flip*, reg-objects support a set of m different *write* invocations, each of which allows a process to set the object to a specified value. Implementations of reg from $\mathsf{bit}_{\mathrm{mu}}$ are discussed in Sect. 4.1.

2.4 The universality of consensus and wait-free hierarchies

Herlihy [11] demonstrated that the consensus types cons_n are *universal* in the following sense: there is a wait-free implementation of any n-ported type T from some set of general registers (reg-objects) and cons_n-objects. Because of this, Herlihy proposed evaluating different types by assigning them consensus numbers. The *consensus number* of type T is the largest integer n for which some set of reg-objects and a single T-object can implement cons_n.

Jayanti [14] questioned two of Herlihy's assumptions in assigning a consensus number to a type T: whether or not reg-objects should be used in an implementation of cons_n and whether or not multiple T-objects can be used.[2] To explore the impact of different choices here, he defined four *wait-free hierarchies*:

- $h_1(\mathsf{T}) \geqq n$ if and only if one T-object can implement n-process consensus.
- $h_1^{\mathrm{r}}(\mathsf{T}) \geqq n$ if and only if some set of reg-objects and one T-object can implement n-process consensus.
- $h_{\mathrm{m}}(\mathsf{T}) \geqq n$ if and only if some set of T-objects can implement n-process consensus.
- $h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{T}) \geqq n$ if and only if some set of reg- and T-objects can implement n-process consensus.

(Thus, for example, $h_1(\mathsf{T})$ is the largest n such that a single T-object can implement n-process consensus.) Herlihy's assignment of consensus number corresponds to Jayanti's hierarchy h_1^{r}. It is clear from these definitions that, for all types T, $1 \leqq h_1(\mathsf{T}) \leqq h_1^{\mathrm{r}}(\mathsf{T}) \leqq h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{T})$ and $h_1(\mathsf{T}) \leqq h_{\mathrm{m}}(\mathsf{T}) \leqq h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{T})$. In addition, standard techniques can be used to show that, if T is n-ported, then $h(\mathsf{T}) \leqq n$ (where h is any of the hierarchies given above).

Ideally, the assignment of a consensus (or hierarchy) number to a type should be a good measure of the type's computational power. The larger the number assigned, the more power the type has to implement other types. Indeed, Herlihy's result on the universality of consensus shows that, if $h(\mathsf{T}) = n$ (where h is any of the hierarchies given above) and T' has at most n ports, then there is an implementation of T' using some number of reg- and T-objects.

Given four different ways of assigning these values, it makes sense to consider which is best. Jayanti identified a desirable property of hierarchies that he called *robustness*. Hierarchy h is *robust* if, for every choice of n, T, and $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_m$, the relations $h(\mathsf{T}) \geqq n$ and $h(\mathsf{T}_j) < n$ (for all $1 \leqq j \leqq m$) imply that there is no implementation of T from any set of objects of types $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_m$. Robustness implies that there can be no "synergistic" effect that would allow "weak" types to implement a "strong" one.

Jayanti showed that none of h_1, h_1^{r}, and h_{m} could be robust if it were not equal to $h_{\mathrm{m}}^{\mathrm{r}}$. He then showed that both h_1^{r} and h_{m} were different from $h_{\mathrm{m}}^{\mathrm{r}}$, proving that only $h_{\mathrm{m}}^{\mathrm{r}}$ might be robust (h_1 cannot equal $h_{\mathrm{m}}^{\mathrm{r}}$ if either h_{m} or h_1^{r} does not). Jayanti left the robustness of $h_{\mathrm{m}}^{\mathrm{r}}$ as an open question. Recent papers [3, 24] have claimed that $h_{\mathrm{m}}^{\mathrm{r}}$ is robust for certain classes of deterministic types. However, Moran and Rappoport [21] exhibited a class of deterministic types for which $h_{\mathrm{m}}^{\mathrm{r}}$ is not robust.[3]

Jayanti's proof that h_{m} differs from $h_{\mathrm{m}}^{\mathrm{r}}$ exhibited a type T with $h_{\mathrm{m}}(\mathsf{T}) = 1$ and $h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{T}) \geqq 2$. This type is nondeterministic. The remainder of this paper considers restricted classes of types for which h_{m} is shown equal to $h_{\mathrm{m}}^{\mathrm{r}}$. For these classes, h_{m} is robust if and only if $h_{\mathrm{m}}^{\mathrm{r}}$ is.

3 One-use bits

The main results of this paper stem from the implementation and use of a new concurrent data type called the *one-use bit*. Objects of this type are one-bit registers that can be read only once and written only once. Section 4 shows that objects of this type can be used to implement general registers (reg-objects) in the context of wait-free implementations of consensus, while Section 5 shows that it is easy to implement this type.

The one-use bit type $\mathsf{bit}_{1\mathrm{u}}$ is defined to be $\langle 2, Q_{1\mathrm{u}}, I_{1\mathrm{u}}, R_{1\mathrm{u}}, \delta_{1\mathrm{u}} \rangle$, where $Q_{1\mathrm{u}} = \{\mathit{off}, \mathit{on}, \mathit{dead}\}$, $I_{1\mathrm{u}} = \{\mathit{look}, \mathit{set}\}$, $R_{1\mathrm{u}} = \{\mathit{off}, \mathit{on}, \mathit{ok}\}$, and $\delta_{1\mathrm{u}}$ is defined as follows.

$$\delta_{1\mathrm{u}}(\mathit{off}, 1, \mathit{look}) = \{\langle \mathit{dead}, \mathit{off} \rangle\} \tag{1}$$

$$\delta_{1\mathrm{u}}(\mathit{on}, 1, \mathit{look}) = \{\langle \mathit{dead}, \mathit{on} \rangle\} \tag{2}$$

$$\delta_{1\mathrm{u}}(\mathit{dead}, 1, \mathit{look}) = \{\langle \mathit{dead}, \mathit{off} \rangle, \langle \mathit{dead}, \mathit{on} \rangle\} \tag{3}$$

$$\delta_{1\mathrm{u}}(q, 1, \mathit{set}) = \{\langle q, \mathit{ok} \rangle\} \tag{4}$$

$$\delta_{1\mathrm{u}}(\mathit{off}, 2, \mathit{set}) = \{\langle \mathit{on}, \mathit{ok} \rangle\} \tag{5}$$

$$\delta_{1\mathrm{u}}(\mathit{on}, 2, \mathit{set}) = \{\langle \mathit{dead}, \mathit{ok} \rangle\} \tag{6}$$

$$\delta_{1\mathrm{u}}(\mathit{dead}, 2, \mathit{set}) = \{\langle \mathit{dead}, \mathit{ok} \rangle\} \tag{7}$$

$$\delta_{1\mathrm{u}}(q, 2, \mathit{look}) = \{\langle q, \mathit{ok} \rangle\} \tag{8}$$

(Lines 4 and 8 above hold for any $q \in Q_{1\mathrm{u}}$ and indicate that the specified invocations are useless.)

State *off* is usually chosen as an initial state. The process connected to port 2 can write the $\mathsf{bit}_{1\mathrm{u}}$-object once by invoking *set*. This moves the object from state *off* to state *on*. The process connected to port 1 can read the object once by invoking *look*. If the object is in state *off* or the state *on*, that state is returned to process. After two *set* invocations or one *look* invocation, the object enters the

[2] The reader should note that, while Herlihy's model allowed only one T-object to be used, all the impossibility results that he showed apply for any number of objects

[3] Moran and Rappoport defined a deterministic 4-ported type W with the property that $h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{W}) < 3$ and that there is a wait-free implementation of 4-process consensus using a single W-object, four cons_3-objects, and read/write memory. Since $h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{cons}_3) = 3$, this implies that $h_{\mathrm{m}}^{\mathrm{r}}$ is not robust for any class of types that includes both W and cons_3. However, the proof by Moran and Rappoport of $h_{\mathrm{m}}^{\mathrm{r}}(\mathsf{W}) < 3$ required the restriction that, in a 3-process system, a single process could not access a W-object (which is 4-ported) by more than one port. The model of Borowsky, Gafni, and Afek [3] does not allow this restriction

state *dead*. At this point, no further information can be derived from the object from port 1 because of the type's nondeterminism. Note that this nondeterminism will play no role in our use of the type (Sect. 4); a *look* will never be invoked when the object is in state *dead*.

4 Using one-use bits

Although one-use bits (bit_{1u}-objects) are apparently weaker than general registers (reg-objects, defined in Sect. 2.3.2), we can show that, within the context of wait-free implementations of consensus, they are equally powerful. This is shown through the following three observations.

1. General registers can be implemented using single-reader, single-writer multi-use bits (bit_{mu}-objects, defined in Sect. 2.3.2).
2. For any n, any wait-free implementation of n-process consensus, and any bit_{mu}-object b used by the implementation, there are bounds on the number of times that b is read and flipped in any execution of the implementation.
3. If there are bounds on the number of times that a bit_{mu}-object b can be read and flipped, then b can be implemented by a finite number of single-use bits (bit_{1u}-objects).

These are shown in Sects. 4.1–4.3 below.

4.1 Implementing general registers

A large body of literature has considered the definition and implementation of a variety of different kinds of read/write registers (or memory) and the relationships between these kinds. Most recent research in wait-free computation has considered registers that are atomic (linearizable), multi-reader, multi-writer, and multi-valued (reg-objects). Earlier research considered weaker kinds of registers. Figure 1 summarizes a sequence of constructions that allow bit_{mu}-objects to be used to implement reg; the bracketed notes are references to the bibliography.

All the constructions are wait-free and exist for any number of processes. The following paragraph details these constructions. Note that this is a very incomplete account of the large volume of results that have been produced, mentioning only those that are necessary for the results of this paper.

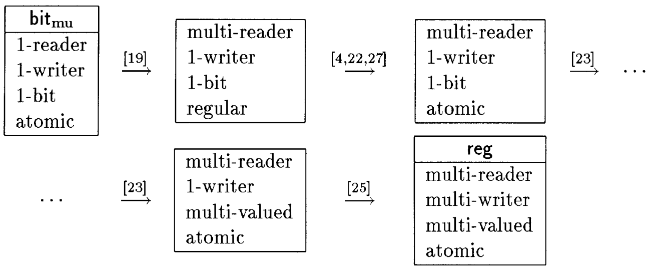

Fig. 1. Implementations of registers

Lamport [19] showed that there is an implementation of multi-reader, single-writer, regular bits from single-reader, single-writer, regular bits. Since regular bits are weaker than atomic bits, this implementation can also use bit_{mu}-objects. Burns and Peterson [4], Newman-Wolfe [22], and Singh, Anderson, and Gouda [27] all showed that there is an implementation of multi-reader, single-writer, atomic bits from multi-reader, single-writer, regular bits. Peterson [23] showed that there is an implementation of multi-reader, single-writer, atomic, multi-valued registers from multi-reader, single-writer, atomic bits. Peterson and Burns [25] showed that there is an implementation of multi-reader, multi-writer, atomic, multi-valued registers (reg-objects) from multi-reader, single-writer, atomic, multi-valued registers. It follows from all these results that there are implementations of reg from bit_{mu}-objects.

4.2 Access bounds in wait-free consensus

Suppose that there is a wait-free implementation of n-process consensus that uses some number of registers and T-objects. The observations of the previous section allow us to assume that the registers are bit_{mu}-objects. We show that, for each bit_{mu}-object b, there exist constants r_b and f_b such that in no execution of the implementation is b read more than r_b times or flipped more than f_b times.

Consider the executions of the implementation of cons_n as a collection of trees. Each vertex of a tree corresponds to some configuration of the implementing objects (of types bit_{mu} and T) and the "program counters" of the n processes in their implementing functions. The roots of the trees correspond to possible initial configurations: the initial states of the implementing objects and the vector of invocations that the n processes will first apply to the cons_n-object (each may be i_0 or i_1); that is, each process is at the "entry point" of one of its two implementing functions. A configuration C_1 is the parent of C_2 if C_2 results from C_1 through the execution of one low-level operation (on a bit_{mu}-object or a T-object) by one process *in its first invocation on the* cons_n*-object*. (If a configuration can be reached via multiple paths, it appears multiple times.) Any configuration in which some process accesses the cons_n-object a second time does not appear in a tree. Thus, a configuration in which all n processes have completed their first invocations is a leaf vertex.

We consider only first invocations because any later invocations by a process must return the same response as first (see Sect. 2.3.1). We assume that each process stores the first response locally and does not access any of the implementing objects after its first invocation.

Consider any one of these trees. We show by contradiction that it is finite. Assume that it is not. This means that a form of König's Infinity Lemma [9; Theorem 2.8, page 32] applies:

Lemma 1 (König). *If G is an infinite digraph, with root r and finite out-degree for all its vertices, then G has an infinite directed path, starting in r.*

The out-degree of each vertex in our trees is finite. If T is deterministic, it is bounded by n. Any vertex has at most

n children, one for each process. This is because the processes are deterministic, as are $\mathsf{bit}_{\mathrm{mu}}$ and T. If $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ is nondeterministic, the out-degree of a vertex is bounded by n times the size of the largest set $\delta(q, j, i)$ (recall that these sets are finite for nondeterministic types).

König's Lemma now implies that there is an infinite path from the root of the tree. This path corresponds to some execution of the implementation. This means that there is an execution in which some process executes an infinite number of steps but never completes its first invocation on cons_n. This contradicts the fact that the implementation is wait-free.

The tree described is thus finite; let h be its height, the maximum length of a path from the root. There are 2^n such trees. This is because the initial states of the implementing objects are the same in all trees. (the implementation must specify a unique initial state for each such object), and only the choice of the entry points of the n processes can vary. Let $h_{\max}$ be the maximum h over all the trees; since there are finitely many trees, $h_{\max}$ is finite. This means that in no execution are more than $h_{\max}$ steps executed. Thus, at most $h_{\max}$ accesses are invoked on any implementing object in any execution. By choosing $r_b = f_b = h_{\max}$, we know that in no execution of the implementation does any process read a $\mathsf{bit}_{\mathrm{mu}}$-object b more than r_b times or flip it more than f_b times.

4.3 Implementing multi-use bits

This section shows how any $\mathsf{bit}_{\mathrm{mu}}$-object that is accessed a bounded number of times can be implemented with a finite number of $\mathsf{bit}_{1\mathrm{u}}$-objects. Suppose that $\mathsf{bit}_{\mathrm{mu}}$-object b is initialized to v, read at most r_b times, and flipped at most f_b times.

The implementation uses $r_b \cdot f_b$-objects of type $\mathsf{bit}_{1\mathrm{u}}$. These form an $r_b \times f_b$ array $bits[1 \ldots r_b, 1 \ldots f_b]$, all elements of which are initially in state *off*. A *read* on port 1 is implemented by $P_{1,read}$ and a *flip* on port 2 is implemented by $P_{2,\,flip}$ (see below). (Recall that the other invocations are useless and trivially return *ok*.) Each row of the array corresponds to an execution of $P_{1,read}$ and each column to an execution of $P_{2,\,flip}$. Intuitively, an execution of $P_{1,read}$ and one of $P_{2,\,flip}$ "communicate" through one element of the array. If $P_{2,\,flip}$ invokes *set* on that element before $P_{1,read}$ invokes *look* on it, the *flip* will be "seen" by the *look*; otherwise, it will not. $P_{1,read}$ invokes *look* on $\mathsf{bit}_{1\mathrm{u}}$-objects in its corresponding row until it finds one that has not been set. $P_{2,\,flip}$ invokes *set* on all $\mathsf{bit}_{1\mathrm{u}}$-objects in its corresponding column. Port 1 (respectively, port 2) of b corresponds to port 1 (respectively, port 2) of each of the $\mathsf{bit}_{1\mathrm{u}}$-objects. The reading process maintains two local integer variables *Reads* and *Col*, while the writing process maintains local *Row* and *Flips*; these are all initially 1.

The implementing programs use the following notation. If i is an invocation on some type T and O is a T-object, $i(O)$ (called an *O-access*) is used to indicate that the invocation is performed and the result returned. The following are the implementing programs (recall that v is the initial value of the $\mathsf{bit}_{\mathrm{mu}}$-object being implemented):

```
$P_{1,read}$:: while $Col \leqq f_b$ and $look(bits[Reads, Col]) = on$ do
                   $Col := Col + 1$
               $Reads := Reads + 1$
                   return$((v + (Col - 1)) \bmod 2)$

$P_{2,\,flip}$:: for $Row := 1$ to $r_b$ do
                   $set(bits[Row, Flips])$
               $Flips := Flips + 1$
                   return$(ok)$
```

Note that an execution of $P_{1,read}$ does not invoke *look* on every $\mathsf{bit}_{1\mathrm{u}}$-object in its row. Instead, it starts in the column where the previous execution ended and proceeds only until it finds a column in which its access returns *off*. The *on-column* of an execution of $P_{1,read}$ is one less than the value of *Col* at the end of that execution. It is the total number of columns in which any execution of $P_{1,read}$ has seen a $\mathsf{bit}_{1\mathrm{u}}$-object with state *on*. The *index* of an execution of $P_{2,\,flip}$ is the value of *Flips* at the beginning of that execution. This reflects the execution's ordinal position among all executions of $P_{2,\,flip}$; it is also the column of the array to whose entries the execution applies *set*.

To prove the correctness of the implementation, we need to prove it linearizable and wait-free. Wait-freedom is obvious: no execution of $P_{1,read}$ requires more than f_b operations and all executions of $P_{2,\,flip}$ use exactly r_b operations. To show linearizability, we must show that, for each execution of the implementation from a state, there is a sequential history from that state that preserves the real-time ordering of the operations in the execution. Consider an execution of the implementation from state v in which $P_{1,read}$ is executed at most r_b times and $P_{2,\,flip}$ at most f_b times. We now describe a linear ordering of the corresponding operations. The relative order of the *read* operations is that in which they were invoked, as is the relative order of the *flip* operations. A *read* operation is ordered before a *flip* if its execution's *on*-column is less than the index of the execution of the *flip*. It is easy to see that, if an execution of $P_{1,read}$ has *on*-column c, then the corresponding *read* is preceded by c *flip*'s in the linear ordering.

We first show that the resulting linear ordering respects the real-time ordering of the programs' executions. This is obvious for any pair of operations on the same port; it remains only to show it for the ordering of a *read* operation and a *flip* operation. Suppose that the *read*'s execution has *on*-column c and the *flip*'s has index i. We must consider two cases:

- The execution of $P_{1,read}$ completes before that of $P_{2,\,flip}$ begins. In this case, the ith column of the array *bits* is completely *off* when the $P_{1,read}$ executes (the same is true for all previous executions of $P_{1,read}$ as well). This means that the **while** loop will terminate with *Col* less than equal to i; c is one less than this value of *Col*. Thus, $c < i$ and the two operations are ordered correctly.
- The execution of $P_{2,\,flip}$ completes before that of $P_{1,read}$ begins. In this case, the ith column of the array *bits* is completely *on* when $P_{1,read}$ executes, as are all previous columns. When $P_{1,read}$ executes, it will find all these $\mathsf{bit}_{1\mathrm{u}}$-objects *on* and advance *Col* to be at least $i + 1$; since c is one less than this value of *Col*, $c \geqq i$.

Thus, $\neg(c < i)$ and the two operations are ordered correctly.

Finally, we need to show that this linear ordering is indeed a sequential history from v. All *flip* invocations return *ok*, as desired. Consider some *read* operation that is preceded by f *flip* operations. This means that the *on*-column of the corresponding execution of $P_{1,read}$ is f, and this execution ended with $Col = f + 1$. The execution thus returns $(v + (Col - 1)) \bmod 2 = (v + f) \bmod 2$. Since the bit_{1u}-object was initially v and was then flipped f times, this is the correct value.

5 Implementing one-use bits

This section illustrates two cases in which one-use bits (bit_{1u}-objects) can be implemented. These are non-trivial deterministic types and types above level 1 in the hierarchy h_m.

5.1 Non-trivial deterministic types

This section shows that an object of any non-trivial deterministic type T can implement bit_{1u}. Informally, T is *non-trivial* if a T-object, suitably initialized, is capable of providing processes with some information about how it has been accessed. Deriving an implementation of bit_{1u} is much simpler for oblivious types, and this case is presented in Sect. 5.1.1. The general case is presented in Sect. 5.1.2.

5.1.1 Oblivious types

Most, but not all, deterministic oblivious types can implement bit_{1u}. Some types, however, are so weak as to be incapable of implementing any interesting type. Consider, for example, a type $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ such that $|R| = 1$. Because the type must return the same response to every invocation, there is no way that it can supply any useful information. Formally, an oblivious type $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ is *trivial* if, for every state $q \in Q$ and every invocation $i \in I$, there is a response $r_{qi} \in R$ such that, for each state p reachable from q (including q itself), there is a state p' such that $\delta(p, i) = \langle p', r_{qi} \rangle$.[4] A trivial oblivious type, once initialized, returns the same response to each occurrence of a given invocation; processes can gain no information by accessing an object of the type. An oblivious type that is not trivial is *non-trivial*. We now show that an object of any non-trivial oblivious deterministic type can implement bit_{1u}.

Let $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ be a non-trivial oblivious deterministic type. This means that there are states q and p, invocation i, and responses r_q and r_p such that $r_q \neq r_p$, p is reachable from q, $\delta(q, i) = \langle q', r_q \rangle$ (for some state q'), and $\delta(p, i) = \langle p', r_p \rangle$ (for some state p'). In this case, q, p, and i are said to *witness* T*'s non-triviality*. We first show that p, q, and i can be chosen such that p is reachable from q in one operation.

[4] Jayanti, Chandra, and Toueg [15, Sect. 5.1.2] give a slightly stronger definition of a trivial oblivious type. Their definition requires that, for all i, p, and q, $r_{qi} = r_{pi}$

Lemma 2. *Let* $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ *be non-trivial, oblivious, and deterministic. Then there are states q and p and invocation i that witness* T*'s non-triviality such that p is reachable from q in one operation.*

Proof. Let q and p be the states and i the invocation that witness T's non-triviality. Let $l \geqq 1$ be the number of operations between q and p in some sequential history of T (such a history must exist since p is reachable from q) and suppose that p, q, and i were chosen to minimize l. If $l = 1$, we are done. Otherwise, let s be the state reachable from q by the first $l - 1$ operations that lead from q to p. Let $\delta(q, i) = \langle q', r_q \rangle$, $\delta(p, i) = \langle p', r_p \rangle$, and $\delta(s, i) = \langle s', r_s \rangle$. Since $r_q \neq r_p$, r_s must be different from one of them. Note that s is reachable from q and p is reachable from s; thus, either q, s and i or s, p, and i witness T's non-triviality. In either case, fewer than l operations are needed for the reachability. This contradicts the minimality of l. □

We now give an implementation of bit_{1u} from one T-object. Let q, p, and i witness T's non-triviality such that there is an invocation i_s whose operation from q leads to p; Lemma 2 guarantees the existence of such an invocation. We use one T-object O, initialized to state q. A *look* on port 1 of the bit_{1u}-object is performed as follows:

```
P_{1,look}:: if $i(O) = r_q$ then
                 /* $O$ was still in state $q$ */
                 return ($off$)
             else
                 /* $O$ was not in state $q$ */
                 return ($on$)
```

A *set* on port 2 is performed as follows:

```
P_{2,set}:: $i_s(O)$
            return($ok$)
```

(Recall that *set* on port 1 and *look* on port 2 are useless.) Intuitively, state q corresponds to *off*, p to *on*, and any other state to *dead*.

To prove the correctness of the implementation, we need to prove it linearizable and wait-free. Wait-freedom is obvious: each invocation uses exactly one operation on T-object O. To show linearizability, we must show that, for each execution of the implementation, there is a sequential history that preserves the real-time ordering of the operations in the execution. Consider an execution of the implementation. We now describe a linear ordering of the corresponding operations. Since each invocation contains exactly one O-access, order the corresponding operations according to the order of these accesses. It is easy to see that the resulting linear ordering respects the real-time ordering of the programs' executions. If the execution of two invocations does not overlap in real time, then the O-access of the first must precede that of the second, and the two operations are ordered correctly.

Finally, we need to show that the linear ordering of operations specified above is indeed a sequential history from *off*. All *set* invocations return *ok*, as desired. Because of the nondeterminism in the specification of bit_{1u}, all *look* invocations besides the first can correctly return either *off* or *on*, and they do so. Consider the first *look* invocation and the following three cases:

- The *look* is first in the linear ordering. This means that the O-access of the first execution of $P_{1,look}$ preceded all others and thus occurred when O was in state q. Therefore, this access returned r_q and $P_{1,look}$ returns *off*, as desired.
- The *look* follows exactly one *set* in the linear ordering. This means that the first execution of $P_{2,set}$ first invoked i_s on O in state q. After this, O was in state p. The first execution of $P_{1,look}$ then applied i to O and received response r_p, which is different from r_q. Therefore, $P_{1,look}$ returned *on*, as desired.
- The *look* follows two or more *set* operations in the linear ordering. In this case, it can correctly return either *off* or *on*, and it does so.

5.1.2 General types

The previous section showed that any non-trivial *oblivious* deterministic type can implement one-use bits. The definition of triviality and the proof depended on the obliviousness of the type being used. This section generalizes that result to general types that are not necessarily oblivious.

A deterministic type is *trivial* if, for all ports, all finite sequences of invocations on that port always return the same finite sequence of responses regardless of any invocations performed (and the order in which they are performed) on other ports. In other words, $\mathsf{T} = \langle n, Q, I, R, \delta \rangle$ is trivial if, for all states $q \in Q$, all finite histories H_1 and H_2 from q, and all ports $j \in N_n$, $invs(H_1, j) = invs(H_2, j)$ implies $resps(H_1, j) = resps(H_2, j)$. A type is *non-trivial* if it is not trivial. Thus, for any non-trivial T, there is a state q, finite histories H_1 and H_2 from q, and port j such that $invs(H_1, j) = invs(H_2, j)$ and $resps(H_1, j) \neq resps(H_2, j)$. Call H_1 and H_2 a *non-trivial pair from q on port j.* Note that different sequences of operations may be invoked on ports other than port j in H_1 and H_2.

For the remainder of this section, we will assume that q, H_1, H_2, and j are chosen such that $|H_1| + |H_2|$ is minimal among all non-trivial pairs. Let $\vec{\imath} = invs(H_1, j)$ (which is the same as $invs(H_2, j)$). Since $\vec{\imath}$ is finite, suppose that $\vec{\imath} = \langle i_1, i_2, \ldots, i_k \rangle$ and thus has length k. The following sequence of lemmas demonstrate certain properties of H_1 and H_2. These properties allow one T-object to implement bit_{1u}.

Lemma 3. *The last operation in each of H_1 and H_2 is on port j.*

Proof. Without loss of generality, assume that H_1 ends with an operation on a port other than j. Let H_1' be the prefix of H_1 up to but not including this last operation. Since that operation is not on port j, $invs(H_1', j) = \vec{\imath}$ and $resps(H_1, j) = resps(H_1', j)$. This means that H_1' and H_2 are a minimal pair and $|H_1'| + |H_2| = |H_1| + |H_2| - 1$. This contradicts the minimality of H_1 and H_2. $\square$

Lemma 4. *One of H_1 and H_2 has length k; that is, it consists only of operations on port j.*

Proof. Let H_o be the history from q consisting only of the invocations in $\vec{\imath}$ on port j. Because $resps(H_1, j) \neq resps(H_2, j)$, $resps(H_o, j)$ must differ from at least one of them. Without loss of generality, assume that it differs from $resps(H_2, j)$. In this case, H_o and H_2 are also a non-trivial pair. Since $|H_1| + |H_2|$ is minimal, $|H_o| + |H_2| = k + |H_2| \geqq |H_1| + |H_2|$, so $|H_1| \leqq k$. Since H_1 must contain at least the k operations on port j, $|H_1| = k$.[5] $\square$

Lemma 4 allows us to assume, without loss of generality, that H_1 contains only the k invocations on port j and that H_2 contains at least one invocation on some other port (otherwise, $H_1 = H_2$ and they are not a non-trivial pair).

Lemma 5. *H_2 consists of one operation on some port other than j followed by k operations on port j.*

Proof. We begin by proving that the last k invocations in H_2 are all on port j. Let $|H_2| = l > k$. Suppose that H_2 is chosen so that the last operation o on some port other than j is as late as possible and suppose that this operation is followed by m operations on port j; that is, we are minimizing m over all possible choices for H_2. Since there are only k operations on port j, $0 \leqq m \leqq k$. We wish to prove $m = k$. Lemma 3 implies $m > 0$, so o is immediately followed by at least one operation on port j. Let H_s be a sequential history from q with the same invocations as H_2 in the same order except that the order of o's invocation and that of the immediately following invocation on port j are reversed. In H_s, the last operation on a port other than j is followed by $m - 1$ operations on port j. Since H_2 was chosen to minimize m, H_s and H_1 cannot form a nontrivial pair. Since $invs(H_s, j) = \vec{\imath}$ (the order of invocations on port j did not change), it must be that $resps(H_s, j) = resps(H_1, j)$. Since $resps(H_1, j) \neq resps(H_2, j)$, $resps(H_s, j) \neq resps(H_2, j)$.

Note that H_2 and H_s are identical through their first $l - (m + 1)$ operations. Let q' be the state of each of these histories after these operations and let H_2' and H_s' be the suffixes of H_2 and H_s, respectively, of length $m + 1$. These are both sequential histories from q' containing o and the last m operations on port j. Thus, $invs(H_2', j) = invs(H_s', j)$. Because H_2 and H_s are identical before these suffixes and because $resps(H_2, j) \neq resps(H_s, j)$, it must be that $resps(H_2', j) \neq resps(H_s', j)$. This means that H_2' and H_s' form a nontrivial pair. $|H_2'| + |H_s'| = 2(m + 1)$. By the minimality of H_1 and H_2, $|H_2'| + |H_s'| = 2(m + 1) \geqq |H_1| + |H_2| = k + l$. Since $l > k$, we have $2(m + 1) \geqq k + l > 2k$, so $m + 1 > k$. Because $m \leqq k$ by definition, we have $m = k$, as desired.

We need now to show that $|H_2| = l = k + 1$; this will imply that H_2 is a single operation on a port other than j followed by k operations on port j. Recall that q' is the state of H_2 after its first $l - (m + 1) = l - (k + 1)$ operations and that o is an operation on a port other than j that is executed from state q'. Let H_r be a history from q generated by the first $l - (k + 1)$ operations in H_2 (which lead to q') followed by the k invocations in $\vec{\imath}$; thus, H_r does not include o. Since $|H_r| < |H_2|$, H_1 and H_r cannot

[5] This implies that $H_o = H_1$

form a nontrivial pair; since $invs(H_r, j) = \vec{\imath}$, it must be that $resps(H_r, j) = resps(H_1, j)$. Thus, $resps(H_r, j) \neq resps(H_2, j)$. Let H_r' be the suffix of H_r from q' and recall that H_2' is the suffix of H_2 from q'. Clearly, $invs(H_r', j) = invs(H_2', j) = \vec{\imath}$, $resps(H_r', j) = resps(H_r, j)$, and $resps(H_2', j) = resps(H_2, j)$. Thus, H_r' and H_2' form a nontrivial pair. Since H_1 and H_2 are the shortest such pair, we have $|H_r'| + |H_2'| = k + (k+1) = 2k + 1 \geqq |H_1| + |H_2| = k + l$. Thus, $l \leqq k + 1$. But $l > k$ by definition, so $l = |H_2| = k + 1$.[6] □

We now know, by Lemma 4, that H_1 consists of k operations on port j and, by Lemma 5, that H_2 consists of one invocation, say i_s, on some other port, say j_s, followed by the same k invocations on port j. Let q_s be the state the results from applying i_s to O in state q. We can now show that an object O of any non-trivial deterministic type can be used by two processes to implement bit_{1u}. Initialize O to the state q associated with the shortest nontrivial pair (see above). Port 1 of the bit_{1u}-object (the reading port) is connected to port j of O and a *look* on that port is performed as follows:

```
$P_{1,look}$::  for $l := 1$ to $k$
                    $r[l] := i_l(O)$
                If $\vec{r} = resps(H_1, j)$ then
                    /* writer has not written */
                    return(off)
                else
                    /* writer has written */
                    return(on)
```

The process performs the invocations in $\vec{\imath}$ and checks to see whether or not $resps(H_1, j)$ is returned.[7] Port 2 of the bit_{1u}-object (the writing port) is connected to port j_s of O and a *set* on that port is performed simply with the one invocation i_s from H_2 on port j_s:

```
$P_{2,set}$::  $i_s(O)$
               return(ok)
```

Note that the reader may receive a response that is neither H_1's nor H_2's. However, this still indicates that the writer has written, so *on* can be returned if $\vec{r} \neq resps(H_1, j)$.

To prove the correctness of the implementation, we need to prove it linearizable and wait-free. Wait-freedom is obvious: each invocation of $P_{1,look}$ used exactly k operations on shared object O, and each invocation of $P_{2,set}$ uses one. To show linearizability, we must show that, for each execution of the implementation, there is a sequential history that preserves the real-time ordering of the operations in the execution. Consider an execution of the implementation. Order the corresponding operations linearly in any way that is consistent with their real-time ordering except for the following. If the first execution of $P_{1,look}$ overlaps with the second execution of $P_{2,set}$, order the corresponding *look* after the corresponding (second) *set*. If the first execution of $P_{1,look}$ overlaps with the first execution of $P_{2,set}$ and completely precedes the second, order the corresponding *look* before the corresponding (first) *set* if and only if $P_{1,look}$ returns *off*. Note that, by definition, the resulting linear order respects the real-time ordering of the programs' executions.

We need to show that the linear ordering of operations specified above is indeed a sequential history from *off*. As in the proof in Sect. 5.1.1, all *set* invocations and all *look* invocations besides the first return correct values. Consider the first *look* invocation and the following three cases:

- The *look* is first in the linear ordering. This means that the first execution of $P_{1,look}$ precedes all executions of $P_{2,set}$ or it overlapped with the first such execution and returned *off*. In the first case, the invocations on O from $\vec{\imath}$ on port j preceded all other O-accesses. This means that $\vec{r}$ as computed by $P_{1,look}$ was $resps(H_1, j)$ and $P_{1,look}$ returned *off*. In both cases, the correct value (*off*) is returned.
- The *look* follows exactly one *set* in the linear ordering. This means that the first execution of $P_{1,look}$ completely preceded the second execution of $P_{2,set}$. Also, either the first execution of $P_{1,look}$ took place between the first and second executions of $P_{2,set}$ or it overlapped with the first execution and $P_{1,look}$ returned *on*. In the first case, O was in state q_s when $P_{1,look}$ began and the k invocations in $\vec{\imath}$ took place consecutively. Thus, $\vec{r}$ as computed by $P_{1,look}$ equals $resps(H_2, j)$. Since, by definition, this is different from $resps(H_1, j)$, $P_{1,look}$ returned *on*. In both cases, the correct value (*on*) is returned.
- The *look* follows two or more *set* operations in the linear ordering. In this case, it can correctly return either *off* or *on*, and it does so.

5.2 High-level types in h_m

Let T be any type such that $h_m(\mathsf{T}) \geqq 2$. This means that there is an implementation of cons_2 using only T-objects (without registers). We now show that, even if T is nondeterministic, T can implement bit_{1u}. We do this by exhibiting an implementation of bit_{1u} from cons_2. Since T-objects can implement cons_2, then they can implement bit_{1u}.

Let O be a cons_2-object, initialized to state $\perp$. Port 1 (respectively, port 2) of bit_{1u} corresponds to port 1 (respectively, port 2) of O. A *look* on port 1 of bit_{1u} is performed as follows:

```
$P_{1,look}$::  if $i_0(O) = 0$ then
                    return(off)
                else
                    return(on)
```

A *set* on port 2 is performed as follows:

```
$P_{2,set}$::  $i_1(O)$
               return(ok)
```

Basically, the reader proposes 0, meaning "*look* precedes *set*," while the writer proposes 1, meaning "*set* precedes *look*." The "winner" of O determines the consensus value and thus the ordering of the first *look* and the first *set*. Note

[6] This implies that the prefix of H_2 consisting of its first $l - (k+1)$ operations is empty. This means that $q' = q$, $H_2' = H_2$, and $H_r' = H_r = H_1$

[7] One can show that it is sufficient for the process to check only the last response in $\vec{r}$ to see if it matches that of $resps(H_1, j)$

that this implementation returns the same response to all invocations of *look* on port 1; this is permitted by the nondeterministic specification of bit_{1u}.

To prove the correctness of the implementation, we need to prove it linearizable and wait-free. Wait-freedom is obvious: each invocation uses exactly one operation on shared object O. To show linearizability, we must show that, for each execution of the implementation, there is a sequential history that preserves the real-time ordering of the operations in the execution. Consider an execution of the implementation. We now describe a linear ordering of the corresponding operations. Since each invocation contains exactly one access to O, order the corresponding operations according to the order of these accesses. As in Sect. 5.1.1, it is easy to see that the resulting linear ordering respects the real-time ordering of the programs' executions.

Finally, we need to show that the linear ordering of operations specified above is indeed a sequential history from *off*. As in the proofs in Sect. 5.1, all *set* invocations and all *look* invocations besides the first return correct values. Consider the first *look* invocation and the following three cases:

- The *look* is first in the linear ordering. This means that its O-access preceded all others, which was thus in state $\perp$ when the corresponding execution of $P_{1,look}$ proposed 0 (invoked i_0). By the specification of cons_2, O returned 0, so $P_{1,look}$ returned *off*, as desired.
- The *look* follows exactly one *set* in the linear ordering. This means that an execution of $P_{2,set}$ invoked i_1 on O in state $\perp$. After this, O was in state 1. The corresponding execution of $P_{1,look}$ then applied i_0 to O and, by the specification of cons_2, received response 1. Therefore, the $P_{1,look}$ returned *on*, as desired.
- The *look* follows two or more *set* operations in the linear ordering. In this case, it can correctly return either *off* or *on*, and it does so.

6 Applications to wait-free hierarchies

The above results have two important applications to wait-free hierarchies:

Theorem 6. *Suppose that one of the following holds of type* T:

- T *is deterministic*; *or*
- $h_m(\mathsf{T}) \geqq 2$.

Then $h_m(\mathsf{T}) = h_m^r(\mathsf{T})$.

Proof. Let T be a type with one of the above properties. Recall that $1 \leqq h_m(\mathsf{T}) \leqq h_m^r(\mathsf{T})$ for all types T. It thus suffices to show that $h_m^r(\mathsf{T}) \leqq h_m(\mathsf{T})$. The proof is divided into three cases:

- T is deterministic and trivial. This means that, no matter how a T-object is initialized, any sequence of invocations on a port always returns the same sequence of responses. The object can thus be trivially implemented locally (this conclusion requires our assumption that no more than one process can access a particular port). This means that, if $h_m^r(\mathsf{T}) \geqq n$, then registers alone can implement n-process consensus. Since registers cannot implement 2-process consensus [6, 11, 20], this implies that $h_m^r(\mathsf{T}) = 1$. Since $h_m(\mathsf{T}) \geqq 1$ for any T, $h_m^r(\mathsf{T}) \leqq h_m(\mathsf{T})$ as desired.
- T is deterministic and non-trivial. We show that, for all n, $h_m^r(\mathsf{T}) = n$ implies $h_m(\mathsf{T}) \geqq n$. If $h_m^r(\mathsf{T}) = n$, then registers and T-objects can implement n-process consensus. As noted in Sect. 4.1, the registers can be bit_{mu}-objects. Section 4.2 showed that there is bound on the number of times each bit_{mu}-object may be used and Sect. 4.3 showed that, if this is the case, each such bit_{mu}-object may be implemented by a finite number of bit_{1u}-objects. Section 5.1 showed that an object of any non-trivial deterministic type can implement bit_{1u}. Thus, T-objects can implement n-process consensus (without registers). This implies that $h_m(\mathsf{T}) \geqq n$, as desired.
- $h_m(\mathsf{T}) \geqq 2$. Again, we show that, for all n, $h_m^r(\mathsf{T}) = n$ implies $h_m(\mathsf{T}) \geqq n$. As noted above, if $h_m^r(\mathsf{T}) = n$, then some set of T and bit_{1u}-objects can implement n-process consensus. Section 5.2 showed that one T-object can implement bit_{1u}. Thus, some set of T-objects can implement n-process consensus without using registers. This implies that $h_m(\mathsf{T}) \geqq n$, as desired.

In all cases, $h_m^r(\mathsf{T}) \leqq h_m(\mathsf{T})$. This implies $h_m(\mathsf{T}) = h_m^r(\mathsf{T})$. □

Theorem 6 shows that Jayanti's choice of a type T to distinguish h_m and h_m^r was not accidental: it had to be a nondeterministic type with $h_m(\mathsf{T}) = 1$ and $h_m^r(\mathsf{T}) \geqq 2$.[8]

7 Conclusions

The results of this paper show that, in most cases of interest, registers are not "special" when it comes to implementing wait-free consensus. This can simplify the reasoning process: various arguments made with the assumptions that registers are available (e.g., about the hierarchy h_m^r) apply when they are not (e.g., to the hierarchy h_m); the converse is also true.

Theorem 6 shows that, for two large classes of concurrent data types, Jayanti's wait-free hierarchies h_m and h_m^r are equal. One of these is the class of deterministic types, which is of considerable interest. Furthermore, these results pertain to Jayanti's robustness property. His proof that h_m is not robust does not apply, for example, to deterministic types. Recent papers [3, 24] have claimed that h_m^r is robust for certain classes of deterministic types. The results of the current paper would then imply that h_m is also robust for these types.

Although this paper has shown how most interesting types can implement registers in the context of a wait-free consensus algorithm, one should note that this context was required only in Sect. 4.2. Since results similar to that section can be shown for wait-free implementations of any bounded-use type, our implementations of registers are thus applicable also to these implementations.

[8] In fact, Jayanti exhibited the following: for each $k > 1$, a type T_k such that $h_m^r(\mathsf{T}_k) = k$ and $h_m(\mathsf{T}_k) = 1$

Acknowledgements. We are grateful to Scott McCrickard for discussing this work with us. In addition, we thank the anonymous referees for many useful commments.

References

1. Afek Y, Greenberg DS, Merritt M, Taubenfeld G: Computing with faulty shared memory. ACM 42(6): 1231–1274 (1995)
2. Afek Y, Weisberger E, Weisman H: A completeness theorem for a class of synchronization objects. In: Proceedings of the 12th ACM Symposium on Principles of Distributed Computing, pp 159–170. ACM Press, August 1993
3. Borowsky E, Gafni E, Afek Y: Consensus power makes (some) sense! In: Proceedings of the 13th ACM Symposium on Principles of Distributed Computing, pp 363–372. ACM Press, August 1994
4. Burns JE, Peterson GL: Constructing multi-reader atomic values from non-atomic values. In: Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, pp 222–231. ACM Press, August 1987
5. Chandra T, Hadzilacos V, Jayanti P, Toueg S: Wait-freedom versus t-resiliency and the robustness of wait-free hierarchies. In: Proceedings of the 13th ACM Symposium on Principles of Distributed Computing, pp 334–343. ACM Press, August 1994
6. Chor B, Israeli A, Li M: Wait-free consensus using asynchronous hardware. SIAM J Comput 23(4): 701–712 (1994)
7. Cori R, Moran S: Exotic behaviour of consensus numbers. In: Tel G, Vitányi P (eds) Proceedings of the 8th International Workshop on Distributed Algorithms, Lect Notes Comput Sci, vol 857, pp 101–115. Springer, Berlin Heidelberg New York 1994
8. Dolev D, Dwork C, Stockmeyer L: On the minimal synchronism needed for distributed consensus. J ACM 34(1): 77–97 (1987)
9. Even S: Graph algorithms. Computer Science Press, 1979
10. Fischer MJ, Lynch NA, Paterson MS: Impossibility of distributed consensus with one faulty process. J ACM 32(2): 374–382 (1985)
11. Herlihy M: Wait-free synchronization. ACM Trans Program Lang Syst 13(1): 124–149 (1991)
12. Herlihy MP, Wing JM: Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3): 463–492 (1990)
13. Hopcroft JE, Ullman JD: Introduction to automata theory, languages, and computation. Addison-Wesley, 1979
14. Jayanti P: On the robustness of Herlihy's hierarchy. In: Proceedings of the 12th ACM Symposium on Principles of Distributed Computing, pp 145–158. ACM Press, August 1993
15. Jayanti P, Chandra TD, Toueg S: Fault-tolerant wait-free shared objects. In: Proceedings of the 33rd Symposium on Foundations of Computer Science, pp 157–166. IEEE Computer Society Press, October 1992 (A revised and expanded version exists [16])
16. Jayanti P, Chandra TD, Toueg S: Fault-tolerant wait-free shared objects. Technical Report 96-1565, Department of Computer Science, Cornell University, January 1996
17. Jayanti P, Toueg S: Some results on the impossibility, universality, and decidability of consensus. In: Segall A, Zaks S (eds) Proceedings of the 6th International Workshop on Distributed Algorithms. Lect Notes Comput Sci, vol 647, pp 69–84. Springer, Berlin Heidelberg New York 1992
18. Kleinberg J, Mullainathan S: Resource bounds and combinations of consensus objects. In: Proceedings of the 12th ACM Symposium on Principles of Distributed Computing, pp 133–144. ACM Press, August 1993
19. Lamport L: On interprocess communication. Part II: Algorithms. Distrib Comput 1(2): 86–101 (1986)
20. Loui MC, Abu-Amara HH: Memory requirements for agreement among unreliable asynchronous processors. In: Preparata FP (ed) Advances in Computing Research, vol 4, pp 163–183. JAI Press, 1987
21. Moran S, Rappoport L: On the robustness of h_m^r. To appear in: Babaoğlu Ö, Marzullo K (eds) Proceedings of the 10th International Workshop on Distributed Algorithms. Lect Notes Comput Sci. Springer, Berlin Heidelberg New York 1996
22. Newman-Wolfe R: A protocol for wait-free, atomic, multi-reader shared variables. In: Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, pp 232–248. ACM Press, August 1987
23. Peterson GL: Concurrent reading while writing. ACM Trans Program Lang Syst 5(1): 46–55 (1983)
24. Peterson GL, Bazzi RA, Neiger G: A gap theorem for consensus types. In: Proceedings of the 13th ACM Symposium on Principles of Distributed Computing, pp 344–353. ACM Press, August 1994.
25. Peterson GL, Burns JE: Concurrent reading while writing II: the multi-writer case. In: Proceedings of the 28th Symposium on Foundations of Computer Science, pp 383–392. IEEE Computer Society Press, October 1987
26. Plotkin S: Sticky bits and the universality of consensus. In: Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, pp 159–175. ACM Press, August 1989
27. Singh AK, Anderson JH, Gouda MG: The elusive atomic register. J ACM 41(2): 311–339 (1994)

Rida A. Bazzi received his B.E. in Computer and Communications Engineering from the American University of Beirut in 1989, and an M.Sc. and a Ph.D. in Computer Science from Georgia Institute of Technology in 1994. After working as a senior consultant at International Integration Inc. in Cambridge, Massachusetts, he joined the School of Computer Science at Florida International University in August 1995. Currently he is an assistant professor in the Computer Science and Engineering Department at Arizona State University. His major research interests are distributed computing, fault tolerance, and computer vision.

Gil Neiger was born on February 19, 1957 in New York, New York. In June 1979, he received the A.B. in Mathematics and Psycholinguistics from Brown University in Providence, Rhode Island. In February 1985, he spent two weeks picking cotton in Nicaragua in a brigade of international volunteers. In January 1986, he received the M.S. in Computer Science from Cornell University in Ithaca, New York and, in August 1988, he received the Ph.D. in Computer Science, also from Cornell University. Dr. Neiger is currently a Research Scientist at Intel's MicroComputer Research Labs in Hillsboro, Oregon. He is a member of the editorial boards of the *Chicago Journal of Theoretical Computer Science* and the *Journal of Parallel and Distributed Computing.*

Gary L. Peterson received a B.S. degree in Mathematics: Computer Science and a M.S. degree in Mathematics from Portland State University. He has a Ph.D. degree in Computer Science from the University of Washington. He has been on the faculty at the University of Rochester and Georgia Institute of Technology and is currently on the faculty in the Department of Computer and Information Science at Spelman College. His research interests include theory of concurrent control, algorithms and data structures, and complexity theory.