Generic Proofs of Consensus Numbers for Abstract Data Types

Edward Talmage[1] and Jennifer Welch[2]

1 **Parasol Laboratory, Texas A&M University, College Station, USA**
etalmage@tamu.edu
2 **Parasol Laboratory, Texas A&M University, College Station, USA**
welch@cse.tamu.edu

Abstract

The power of shared data types to solve consensus in asynchronous wait-free systems is a fundamental question in distributed computing, but is largely considered only for specific data types. We consider general classes of abstract shared data types, and classify types of operations on those data types by the knowledge about past operations that processes can extract from the state of the shared object. We prove upper and lower bounds on the number of processes which can use data types in these classes to solve consensus. Our results generalize the consensus numbers known for a wide variety of specific shared data types, such as compare-and-swap, augmented queues and stacks, registers, and cyclic queues. Further, since the classification is based directly on the semantics of operations, one can use the bounds we present to determine the consensus number of a new data type from its specification.

We show that, using sets of operations which can detect the first change to the shared object state, or even one at a fixed distance from the beginning of the execution, any number of processes can solve consensus. However, if instead of one of the first changes, operations can only detect one of the most recent changes, then fewer processes can solve consensus. In general, if each operation can either change shared state or read it, but not both, then the number of processes which can solve consensus is limited by the number of consecutive recent operations which can be viewed by a single operation. Allowing operations that both change and read the shared state can allow consensus algorithms with more processes, but if the operations can only see one change a fixed number of operations in the past, we upper bound the number of processes which can solve consensus with a small constant.

1998 ACM Subject Classification E.1 Distributed Data Structures

Keywords and phrases Distributed Data Structures, Abstract Data Types, Consensus Numbers, Distributed Computing, Crash Failures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.32

1 Introduction

Determining the power of shared data types to implement other shared data types in an asynchronous crash-prone system is a fundamental question in distributed computing. Pioneering work by Herlihy [7] focused on implementations that are both wait-free, meaning any number of processes can crash, and linearizable (or atomic). As shown in [7], this question is equivalent to determining the *consensus number* of the data types, which is the maximum number of processes for which linearizable shared objects of a data type can be used to solve the consensus problem. If a data type has consensus number n, then in a system with n processes, shared objects of this type can be used to implement shared objects

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 32; pp. 32:1–32:16


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of any other type. Thus, knowing the consensus number of a data type gives us a good idea of its computational strength.

We wish to provide tools with which it is easy to determine the consensus number of any given data type. So far, most known consensus number results are for specific data types. These are useful, since we know the upper and lower bounds on the strength of many commonly-used objects, but are of no help in determining the consensus number of a new shared data type. Further, even among the known bounds, there are some that seem similar, and even have nearly identical proofs of their bounds, but these piecemeal proofs for each data type give no insight into those relations.

1.1 Summary of Results

We define a general schema for classifying data types, based on their sequential specifications, which we call *sensitivity*. If the information about the shared state which an operation returns can be analyzed to extract the arguments to a particular subsequence of past operations, we say that the data type is sensitive to that subsequence. For example, a register is sensitive to the most recent write, since a read returns the argument to that write. A stack is sensitive to the last $Push$ which does not have a matching Pop, since a Pop will return the argument to that $Push$. We define several such classes in this paper, such as data types sensitive to the kth change to the state, data types sensitive to the kth most recent change, and data types sensitive to the l consecutive recent changes.

We show a number of bounds, both upper and lower, on the number of processes which can use shared objects whose data types are in these different sensitivity classes to solve wait-free consensus. Specifically, we begin by showing that information about the beginning of a history of operations of a shared data type allows processes to solve consensus for any number of processes. This is a natural result, since the ordering of operations on the shared objects allows the algorithm to break symmetry.

An augmented queue, as in [7], using $Enqueue$ and $Peek$ is such a data type, as $Peek$s can always determine what value was enqueued first, and all processes can decide that value. Other examples include a Compare-and-Swap (CAS) object using a function which stores its argument if the object is empty and returns the contents without changing them, if it is not. Repeated applications of this operation have the effect of storing the argument to the first operation executed and returning it to all subsequent operations. There are data types which are stronger than this, such as with operations which return the entire history of operations on the shared object, but our result shows that that strength is unneeded for consensus.

Next, we consider what happens if a data type has operations which depend on the last operations executed. We show that if a data type has only operations whose return values depend exclusively on one operation at a fixed distance back in history, then that data type can only solve consensus for a small, constant number of processes. A data type whose operations which cannot atomically both read and change the shared state, consensus is only possible for one process. If a data type's operations reveal some number l of consecutive changes to the shared state, then it can solve consensus for l processes.

These data types model the scenario when there is limited memory. If we want to store a queue, but only have enough memory to store k elements, we can throw away older elements, yielding a data type sensitive to recent operations. A cyclical queue has such behavior, and with operations $Enqueue$ and $Peek$, where $Peek$ returns the kth-most recent argument to $Enqueue$, has consensus number 1. To solve consensus for more processes with a similar data type, we show that knowledge of consecutive past operations is sufficient. If instead of only one recent argument, we can discern a contiguous sequence of them, we can solve consensus

for more processes. Using the same cyclical k-queue, if our *Peek* operation is replaced with a *ReadAll* which tells the entire contents of the queue atomically, we show that we can solve consensus for k processes. This parameterized result suggests a fundamental property of the amount of necessary information for solving consensus.

1.2 Related Work

Herlihy[7] first introduced the concepts of consensus numbers and the universality of consensus in asynchronous, wait-free systems. He showed that a consensus object could provide a wait-free and linearizable implementation of any other shared object. Further, he showed that different objects could only solve consensus for certain numbers of processes. This gives a hierarchy of object types, sorted by the maximum number of processes for which they can solve consensus. He also proved consensus numbers for a number of common objects.

Many researchers have worked to understand exactly what level of computational power this represents, and when consensus numbers make sense as a measure of computational power. Jayanti and Toueg [8] and Borowsky, et al. [3] established that consensus numbers of specific data types make sense when multiple objects of the type and R/W registers are used, regardless of the objects' initial states. Bazzi et al. [2] showed that adding registers to a deterministic data type with consensus number greater than 1 does not increase the data type's consensus number. Other work establishes that non-determinism collapses the consensus number hierarchy [9, 10], that consensus is impossible with Byzantine [1], and what happens when multiple shared objects can be accessed atomically [11].

Ruppert [12] provides conditions with which it is possible to determine whether a data type can solve consensus. He considers two generic classes of data types, RMW types and readable types. RMW types have a generic Read-Modify-Write operation which reads the shared state and changes it according to an input function. Readable types have operations which return at least part of the state of the shared object without changing it. He shows that for both of these classes, consensus can be solved among n processes if and only if they can discern which of two groups the first process to act belonged to. This condition, called n-discerning, is defined in terms of each of the classes of data types. This has a similar flavor to our first result below, where seeing what happened first is useful for consensus. We define our conditions more directly as properties of the sequential specification of a shared object and also consider different perspectives on what previous events are visible.

Chordia et al. [5] have lower bounds on the number of processes which can solve consensus using classes of objects with definitions similar to [12]–the duration for which two operation orderings are distinguishable affects the objects' consensus power–using algebraic properties, as we do. These results are not directly comparable to those in [12], since they have different assumptions about the algorithms and exact data returned. [5] also does not provide upper bounds, which we focus on.

In another direction, Chen et al. [4] consider the edge cases of several data types, when operations' return values are not traditionally well-defined. An intuitive example is the effect of a *Dequeue* operation on an empty queue, where it could return $\perp$ or return an arbitrary value, never return a useful value again, or a number of other possibilities. They consider a few different possible failure modes, and show that the consensus numbers of objects are different when they have different behaviors when the object "breaks" in such a case. These results are orthogonal to our paper, as they primarily focus on queues and stacks, and assume that objects break in some permanent way when they hit such an edge case. We assume that there is a legal return value for any operation invocation, and that objects will continue to operate even after they hit such an edge case.

2 Definitions and Model

We consider a shared-memory model of computation, where the programming system provides a set of shared objects, accessible to processes. Each object is linearizable (or atomic) and thus will be modeled as providing operations that occur instantaneously. Each object has an *abstract data type*, which gives the interface by which processes will interact with the object. A data type T provides two things: (1) A set of operations OPS which specify an association of arguments and return values as *operation instances* $OP(arg, ret), OP \in OPS$ and (2) A sequential specification ℓ_T which is a set of all the *legal* sequences of operation instances. We use arg_{OP} and ret_{OP} to denote the sets of possible arguments and return values, respectively, to instances of operation OP. Given any sequence ρ of operation instances, we use $\rho|_{args}$ to denote the sequence of arguments to the instances in ρ.

We assume the following constraints on the set of legal sequences:

- *Prefix Closure*: If a sequence ρ is legal, every prefix of ρ is legal.
- *Completeness*: If a sequence ρ is legal, for every operation OP in the data type and every argument $arg \in arg_{OP}$, there exists a response $ret \in ret_{OP}$ such that $\rho \cdot OP(arg, ret)$ is legal (where $\cdot$ is concatenation).
- *Determinism*: If a sequence $\rho \cdot OP(arg, ret)$ is legal, there is no $ret' \neq ret$ such that $\rho \cdot OP(arg, ret')$ is legal.

We say that two finite legal sequences ρ_1 and ρ_2 of operation instances are *equivalent* (denoted $\rho_1 \equiv \rho_2$) if and only if for every sequence ρ_3, the sequence $\rho_1 \cdot \rho_3$ is legal if and only if $\rho_2 \cdot \rho_3$ is legal.

We classify all operations of a data type into two classes, not necessarily disjoint. Informally, *accessors* return some value about the state of a shared object and *mutators* change the state of the object. An operation may be both an accessor and a mutator, in which case we call it a *mixed* operation. If it is an accessor but not a mutator, we say that it is a *pure* accessor. Similarly, pure mutators are mutators but not accessors. Formally,

▶ **Definition 1.** An operation OP of an abstract data type T is a *mutator* if there is some legal sequence ρ of instances of operations of T and some instance op of OP such that $\rho \not\equiv \rho \cdot op$.

▶ **Definition 2.** An operation OP of an abstract data type T is an *accessor* if there is some legal sequence ρ of instances of operations of T, an instance op of some operation of T such that $\rho \cdot op$ is legal, and an instance aop of OP such that $\rho \cdot aop$ is legal, but $\rho \cdot op \cdot aop$ is not legal.

We consider only data types with non-vacuous sets of operations, which include both a mutator and an accessor (not necessarily distinct). Any shared object which does not have a mutator is a constant which can be replaced by a local copy and any shared object without an accessor is of no use to any party, since they cannot discern the state of the object. We further consider only data types whose operation set has at least one mutator which accepts at least two distinct arguments.

2.1 Sensitivity

We will use the concept of *sensitivity* to classify operations. The sensitivity of a set of operations is a means of tracking which previous operations on a shared object cause a particular instance to return a specific value. Intuitively, an operation which has a return value will usually return a value dependent on some subset of previous operation instances.

For example, a *read* on a register will return the argument to the last previous *write*. On a queue, an instance of *Dequeue* will return the argument of the first *Enqueue* instance which has not already been returned by a *Dequeue*. We categorize operations by which previous instances (first, latest, first not already used, etc.) we can deduce, or "see", based on the return value of an instance of an accessor operation.

▶ **Definition 3.** Let OPS be a subset of the operations of a data type T. Let OPS_M denote the set of all mutators in OPS. Let S be an arbitrary function that, given a finite sequence $\rho \in \ell_T$, returns a subsequence of ρ consisting only of instances of mutators.

OPS is defined to be *S-sensitive* iff there exist an accessor $AOP \in OPS$ and a computable function $decode : ret_{AOP} \rightarrow$ the set of finite sequences over $\bigcup_{MOP \in OPS_M} arg_{MOP}$ such that for all $\rho \in \ell_T$, $arg \in arg_{AOP}$, and $ret \in ret_{AOP}$ with $\rho \cdot AOP(arg, ret) \in \ell_T$, $decode(ret) = S(\rho)|_{args}$.

▶ **Definition 4.** A subset OPS of the operations of a data type T is *strictly S-sensitive* if for every $\rho \in \ell_T$, every accessor AOP and every instance $AOP(arg, ret)$ with $\rho \cdot AOP(arg, ret) \in \ell_T$, $ret = S(\rho)|_{args}$. That is, $AOP(arg, ret)$ gives no knowledge about the shared state except for $S(\rho)|_{args}$.

An example, for which we will later show bounds on the consensus number, is *k-front-sensitive* sets of operations:

▶ **Definition 5.** A subset OPS of the operations of a data type T is *k-front-sensitive* for a fixed integer k if OPS is S-sensitive where $S(\rho)$ is the kth mutator instance in ρ for every $\rho \in \ell_T$ consisting of instances of operations in OPS which has at least k mutator instances.

In an augmented queue (as in [7]), the operation set $\{Enqueue, Peek\}$ is k-front-sensitive by this definition, where $k = 1$, S returns the first mutator in a sequence of operation instances, the accessor AOP is $Peek$, and the *decode* function is the identity, since the return value of $Peek$ is the argument to the first $Enqueue$ on the queue. In fact, this operation set is also strictly 1-front-sensitive, since the return value of an instance of $Peek$ is the argument to the single first $Enqueue$.

2.2 Consensus

We are studying the binary *Consensus* problem in an *asynchronous wait-free* model with n processes. In an asynchronous model, processes have no common timing. One process can perform an unbounded number of actions before another process performs a single action. A wait-free model allows for up to $n - 1$ processes to fail by *crashing*. A process which crashes ceases to perform any further actions. Processes may fail at any time and give no indication that they have crashed. Processes which do not crash are said to be *correct*. Any algorithm running in this model must be able to continue despite all other processes crashing, while it cannot in a bounded amount of time distinguish between a crashed process and a slow process. Thus, any algorithm in this model must never require a process to wait for any other process to complete an action or reach a certain state.

We say that an *execution* of an algorithm using a shared data type is a sequence of operation instances, each labeled with a specific process and shared object. The projection of an execution onto a single object must be a *legal* operation sequence, by the sequential specification of the data type.

The consensus problem is defined as follows: Every process has an initial *input* value $v \in \{0, 1\}$. After that, if it is correct, it will *decide* a value $d \in \{0, 1\}$. Once a process decides

a value, it cannot change that decision. Further, all correct processes must satisfy three conditions:

- Termination: All correct processes eventually decide some value
- Agreement: All correct processes decide the same value d
- Validity: All correct processes decide a value which was some process' input

An abstract data type T can *implement* consensus if there is an algorithm in the given model which uses objects of T (plus registers) to solve consensus. The *consensus number* of an abstract data type is the largest number of processes n for which there exists an algorithm to implement consensus among n processes using objects of that data type. If there is no such largest number, we say the data type has consensus number ∞.

We use valency proofs, as in [7], to show upper bounds on the number of processes for which an abstract data type can solve consensus. The following lemma was implicit in [7] and made explicit in [12]. We will use this to make proofs of upper bounds on consensus numbers cleaner.

To state the lemma, we recall the concepts of *valency* and *critical configurations*. A *configuration* represents the local states of all processes and the states of all shared objects. When a process p_i executes a step of a consensus algorithm, it causes the system to proceed from one configuration C to another, which we call a *child configuration*, and denote by $p_i(C)$. A configuration is *bivalent* if it is possible, starting from that configuration, for the algorithm to cause all processes to decide 0 and also possible for it to cause all processes to decide 1. A configuration is *univalent* if from that configuration, the algorithm will necessarily cause processes to always reach the same decision value. If this value is 0, the configuration is *0-valent* and if it is 1, the configuration is *1-valent*. A configuration is *critical* if it is bivalent, but all its child configurations are univalent.

▶ **Lemma 6.** *Every critical configuration has child configurations with different valencies which are reached by different processes acting on the same shared object, which cannot be a register.*

We also restate the following lemma based on Fischer et al. [6].

▶ **Lemma 7.** *A consensus algorithm always has an initial bivalent configuration and must have a reachable critical configuration in every execution.*

Note that we do not require that the set of sensitive operations is the entire set of operations supported by the shared object(s) in the system. There may be other operations. These extra operations do not detract from the ability of a sensitive set of operations to solve consensus, since an algorithm may just choose not to use any other operations. This means that our proofs of the ability to solve consensus are powerful. Impossibility proofs do not get this extra strength, as a clever combination of operations which are not sensitive in a particular way may allow stronger algorithms.

3 k-Front-Sensitive Data Types

We begin by proving a result that generalizes the consensus number of augmented queues. We observe that if all processes can determine which among them was the first to modify a shared object, then they can solve consensus by all deciding that first process' input. For, example, in an augmented queue, any number of processes can solve consensus by each enqueuing their input value, then using peek to determine which enqueue was first [7].

More generally, processes do not need to know which mutator was first, as long as they can all determine, for some fixed integer k, the argument of the kth mutator executed on the shared object. Thus, we have the following general theorem, which applies to either a mutator and pure accessor or to a mixed operation. An example (for $k = 1$) is an augmented queue, where *Peek* returns the first argument ever passed to an *Enqueue*, requiring no decoding. Another similar example is a Compare-And-Swap operation which places a value into a shared register in an initial state and leaves any other value it finds in the object, leaving the argument of the first operation instance still in the shared object, and thus decodable at each subsequent operation. For any k, a mixed operation which stores a value and returns the entire history of past changes, satisfies the definition, since the first argument is always visible to later operations.

▶ **Theorem 8.** *The consensus number of a data type containing a k-front-sensitive subset of operations is ∞.*

We give a generic algorithm (Algorithm 1) which we can instantiate for any k-front-sensitive set of operations (which has a mutator with at least two possible distinct arguments) to solve consensus among any number of processes and prove its correctness as a consensus algorithm. The mutator and accessor in the algorithm are not necessarily distinct operations.

Algorithm 1 Consensus algorithm for a data type with a k-front-sensitive subset of operations, OPS, using a mutator OP and accessor AOP, in OPS

```
1: for i = 1 to k do
2:     OP(input)
3: end for
4: result ← AOP(arg)                    ▷ Arbitrary argument arg
5: val ← decode(result)
6: decide(val)
```

Proof. We must show that this algorithm satisfies the three properties of a consensus algorithm.

- *Termination*: Each process performs a finite number of operations, never waiting for another process. Thus, even in a wait-free system, where any number of other processes may have crashed, all running processes will terminate in a finite length of time.
- *Validity*: By the definition of sensitivity, the decision value at each process will be an argument to a past mutator, and only processes' input values are passed as inputs to mutators on the shared object. Thus, each decision value is some process' input value, and is valid.
- *Agreement*: $decode(result)$ will return the argument to the kth mutator instance at all processes. Since each process completes k mutators before it invokes AOP, there are guaranteed to be at least k mutators preceding the instance of AOP in line 4. Thus, each process decides the same value.

No part of the algorithm or proof is constrained by the number of participating processes, which means that this algorithm solves consensus for any number of processes using a k-front-sensitive data object, so the consensus number of any shared object with a k-front-sensitive set of operations is ∞. ◀

4 Consensus with End-Sensitive Data Types

While data types which "remember" which mutator went first, or kth as above, are intuitively very useful for consensus, other data types can also solve consensus, though not necessarily for an arbitrary number of processes. As a motivating example, consider the difference in semantics and consensus numbers between stacks and queues, shown in [7]. Both store elements given them in an ordered fashion, and the basic version of each has consensus number 2. However, adding extra power to a queue in the form of a *peek* operation gives it consensus number ∞, while adding a similar operation *top* to stacks does not give them any extra power.

If we view the difference between an augmented queue and an augmented stack in terms of sensitivity, $Enqueue$ and $Peek$ on a queue are front-sensitive, while $Push$ and Top on a stack are end-sensitive. That is, queues see what operation was first, while stacks see which was latest. When processes cannot tell how far in the algorithm other processes have gotten, though, due to asynchrony, knowing what operation was latest is not helpful for consensus, as another mutator could finish after some process decides, and that other process will see a different last value. We explore generalizations of this problem and what power still remains in end-sensitive data types.

Unfortunately, the picture for data types with end-sensitive operations sets is more complex than that for front-sensitive types. Here, we have variations depending on exactly which part of the end of the previous history is visible or partly visible to an accessor. It is also important that shared objects have a pure accessor, or some other means of maintaining the state of the object, or else every operation will change what future operations see, making it difficult or impossible to come to a consensus.

We begin with a symmetric definition to that in Section 3, but for recent operations instead of initial, and show that it is not useful for consensus. We then show that certain subclasses, which are sensitive to more than one past operation, have higher consensus numbers.

▶ **Definition 9.** A subset OPS of the operations of a data type T is *k-end-sensitive* for a fixed integer k if OPS is S-sensitive where $S(\rho)$ is the kth-last mutator instance in ρ for every $\rho \in \ell_T$ consisting entirely of instances of operations in OPS and containing at least k mutator instances, and $S(\rho)$ is a null operation instance $\perp(\perp, \perp)$, if there are not at least k mutator instances in ρ.

This definition does not lead to as simple a result as that for front-sensitive sets of operations. As we will show, there is no algorithm for solving consensus for n processes with an arbitrary k-end-sensitive set of operations, for $n > 1$. We will give a number of more fine-grained definitions, showing that different subsets of the class of k-end-sensitive operation sets range in power from consensus number 1 to consensus number ∞.

Consider a set of operations which is S-sensitive, where for all ρ, $S(\rho)$ is the entire sequence of mutator instances in ρ. This set of operations is both k-end-sensitive and k-front-sensitive, for $k = 1$. By the result from Section 3, we know that such a set of operations has consensus number ∞. A similar result holds for any k for which an operation set is k-front-sensitive. Thus, in this section, we will only consider operation sets which are not k-front-sensitive for any k and consider only the strength and limitations of end-sensitivity.

4.1 k-End-Sensitive Types

Unlike front-sensitive data types, if a set of operations is strictly k-end-sensitive, for some fixed k, the data type does not have infinite consensus number. This is a result of the fact that

the kth-last mutator is a constantly moving target, as processes execute more mutators. As we will show, in an asynchronous system, if there are more than one or three processes in the system (depending on the types of operations in the set), operations can be scheduled such that the "moving target" is always obscured for some processes, so they cannot distinguish which process took a step first after a critical configuration, which prevents them from safely deciding any value. We formalize this in the following theorems.

▶ **Theorem 10.** *For $k > 2$, any data type with a strictly k-end-sensitive operation set consisting only of pure accessors and pure mutators has consensus number 1.*

Proof. Suppose we have a consensus algorithm A for at least 2 processes, p_0 and p_1, using such an operation set. Consider a critical configuration C of an execution of algorithm A, as per Lemmas 6, 7. If p_0 is about to execute a pure accessor, p_1 will not be able to distinguish C from the child configuration $p_0(C)$ when running alone, by the definition of a pure accessor. Thus, it will decide the same value in the executions where it runs from either of those states, which contradicts the fact that they have different valencies. If p_1's next operation is a pure accessor, a similar argument holds.

Thus, both processes' next operations from configuration C must be mutators. Assume without loss of generality that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent. Then the states $C_0 = p_1(p_0(C))$ and $C_1 = p_0(p_1(C))$ are likewise 0-valent and 1-valent, respectively.

We construct a pair of executions, extending C_0 and C_1, in which at least one process cannot learn which configuration it is executing from. By the Termination condition for consensus algorithms, at least one process must decide in a finite number of steps, and since the two executions return the same values to the first process to decide, it will decide the same value after $p_1(p_0(C))$ as after $p_0(p_1(C))$, despite those configurations having different valencies. This is a contradiction to the supposed correctness of A, showing that no such algorithm can exist.

We construct the first execution, from C_0, as follows. Assuming for the moment that both processes continue to execute mutators (we will discuss what happens when they don't, below), let p_0 run alone until it is ready to execute another mutator. Then pause p_0 and let p_1 run alone until it is also ready to execute a mutator, and pause it. Let p_0 run alone again until it has completed $k-2$ mutators and is ready to execute another. Next, allow p_1 to run until has executed one mutator, and is prepared to execute a second. We then continue to repeat this sequence, allowing p_0 to run alone again for $k-2$ mutators, then p_1 for one, etc.

The second execution is constructed identically from C_1 except that after C_1, p_0 first runs until it has executed $k-3$ mutators and is ready to execute another, then p_1 executes a mutator. After that, the processes alternate as in the first execution, with p_0 executing $k-2$ mutators and p_1 executing one.

We know that each process, running alone from C_0 (or C_1), must execute at least $k-2$ mutators to be able to see what mutator was first after C, since we have a strictly k-end-sensitive set of operations, which means that any correct algorithm must execute at least that many mutators, since it must be able to distinguish $p_0(C)$ from $p_1(C)$. The way we construct the executions, though, we interleave the operation instances in such a way that each process sees only its own operation instances, and cannot distinguish these executions from running alone from C_0 (or C_1). It is an interesting feature of this construction that we do not force any processes to crash. In fact, we need both processes to continue running to ensure that they successfully hide their own operations from each other.

If we denote any mutator by m and any accessor by a, with subscripts to indicate the process to which the operations belong and superscripts for repetition (in the style of regular

expressions), we can represent these two execution fragments, restricted to the shared object operated on in configuration C, as follows:

$$m_0 \cdot m_1 \cdot \cdot a_0^* \cdot a_1^* \cdot (m_0 \cdot a_0^*)^{k-2} \cdot (m_1 \cdot a_1^*) \cdot (m_0 \cdot a_0^*)^{k-2} \cdots$$

$$m_1 \cdot m_0 \cdot \cdot a_1^* \cdot a_0^* \cdot (m_0 \cdot a_0^*)^{k-3} \cdot (m_1 \cdot a_1^*) \cdot (m_0 \cdot a_0^*)^{k-2} \cdots$$

Since the return value of each accessor is determined by the kth most recent mutator, all operations are pure, and operations are deterministic, we can see that corresponding accessor instances will return the same value in the two executions. Thus, neither process can distinguish the two executions. This is true despite the possibility of operations on other shared objects. To discern the two runs, each process must determine which process executed an operation first after C, and that can only be determined by operations on this shared object. Thus, as long as the return values to operations on this object are the same, since the algorithm is deterministic, the processes will continue to invoke the same operations in the two runs, and will be unable to distinguish the two executions.

This interleaving of operation instances works as long as both processes continue to invoke mutators. Each process must decide after a finite time, though, so they cannot continue to invoke mutators indefinitely. When a process ceases to invoke mutators, we can no longer schedule operations as before to continue hiding its past operations. There are two possible cases for which process(es) finish their mutators first in the two executions.

First, one process (WLOG p_0) may execute its last mutator before the other does, in both executions. When p_0 executes its last mutator in each execution, let it continue to run alone until it decides. Since configuration C, it has only seen its own mutators, and since the data type is strictly k-end-sensitive and no more mutators are executed, will continue to see only its own past mutators in both executions. Thus, the two executions are identical for p_0 and it will decide the same value in both, contradicting their differing valencies.

Second, it may be that in one execution, p_0 executes its last mutator before p_1 does and in the other, p_1 executes its last mutator before p_0. Each process will follow the same progression of local states in both executions, so this case can only arise when p_0's last mutator in the first execution is the last in a block of $k-2$ mutators it runs by itself, and thus first in such a block in the second execution. In the first execution, after p_0 executes its last mutator, let it run alone, as in the first case. In the second execution, after p_1 executes its last mutator, pause it, and allow p_0 to run alone, executing its last mutator and continuing until it decides. By the same argument as case 1, p_0 decides the same value in both executions, contradicting the fact that they have the same valency.

Thus, the assumed consensus algorithm cannot actually exist. ◀

If mixed operations are allowed, the above proof does not hold, as a mixed operation immediately after C will potentially have a different return value than it would in a different execution where there is an intervening mutator. We can show the following:

▶ **Theorem 11.** *For $k > 2$, any data type with an operation set which is strictly k-end-sensitive has consensus number at most 3.*

The proof of this theorem is almost identical to the previous, and is therefore omitted. The primary difference, which yields a higher bound, is that the first two processes which execute operations in a critical configuration crash immediately after those operations, since they may have seen different shared states depending on their order. The other two (assumed) processes can continue in a manner similar to the above, hiding their executions from each other, and not satisfying the univalency of the configurations, yielding a contradiction.

4.2 1- and 2-End-Sensitive Types

The bounds in the previous section require $k > 2$, so we here explore what bounds hold when $k \leq 2$. We continue to consider strictly k-end-sensitive operations; we will consider operation sets with knowledge of additional operations (that is, with larger sensitive sequences $S(\rho)$) later.

We first consider the case $k = 1$, which implies that accessor operations can see the last previous mutator. If all operations are pure mutators or accessors, then it is intuitive that consensus would not be possible, since we could schedule operations such that each process only saw its own mutators. We show that this is, in fact, the case. This generalizes the bound that registers can only solve consensus for one process. If mixed operations are allowed, then a process can obtain some information about other operations, which we will show is enough to solve consensus for two processes, but no more. We know that this bound of 2 is tight, that is, no lower bound can be proved for the entire class, since *Test&Set*, for example, is sensitive to only the last previous mutator and has consensus number 2 [7].

▶ **Theorem 12.** *Any data type with a strictly 1-end-sensitive operation set with no mixed operations has consensus number 1.*

▶ **Theorem 13.** *Any data type with a strictly 1-end-sensitive operation set has consensus number at most 2.*

The proofs for these theorems are standard bivalency proofs, and can be found in the full version of the paper: Technical Report 2015-11-1 at `http://www.cse.tamu.edu/research/tr`.

Next, we consider $k = 2$. If the sensitive set of operations includes a pure accessor, we show that we can solve consensus for 2 processes. Here, unlike our other results, the presence or absence of a mixed operation does not seem to affect the strength for consensus. Instead, it is important to have a pure accessor, which can see the 2nd-last mutator without changing it, which makes it practical for both processes to see the same value.

Data types without a pure accessor seem to have less power than consensus, since it is impossible to check the shared state without changing it. This makes it very difficult for processes to avoid confusing each other. A similar argument to that for Theorem 11 provides an upper-bound of $n \leq 3$ for this data type. We conjecture that it is lower($n = 1$), but do not yet have the tools to prove this formally.

For now, an upper bound on the consensus number of 2-end-sensitive operation types is an open question, but we conjecture that it will be 2, or perhaps 3 with mixed operations as for k-end-sensitive types with $k > 2$, above.

▶ **Theorem 14.** *For $k = 2$, a data type containing a k-end-sensitive set of operation types which includes a pure accessor has consensus number at least 2, using Algorithm 2.*

The proof of Theorem 14 is left to the full version.

4.3 Knowledge of Consecutive Operations

Operation sets which only allow a process to learn about one past operation are generally limited to solving consensus for at most a small constant number of processors. We now show that knowledge about several consecutive recent operations allows more processes to solve consensus. In effect, we are enlarging the moving target we discussed before. We will show that this does, in fact, allow consensus algorithms on more processes, as many as the size of the target, or the number of consecutive operations we can decode. We will then show that when we know the last mutator instances that have happened, the bound is tight.

Algorithm 2 Consensus Algorithm for 2 processes using 2-end-sensitive set of operations using mutator OP and pure accessor AOP

```
1: OP(input)
2: val ← AOP()
3: if decode(val) = ⊥ then
4:     decide(input)
5: else
6:     decide(decode(val))
7: end if
```

This is interesting because the consensus number is not affected by how old the visible operations are, as long as they are at a consistent distance. That is, if we always know a window of history that is a certain fixed number of operations old (no matter what that number is), we can use it to solve consensus. Also interesting is the fact that the bound is parameterized. While knowing a single element of history can solve consensus for a constant number of processes, if we know l consecutive mutators in the history, we can solve consensus for l processes for any natural number l. Thus, knowing more consecutive elements always increases the consensus number.

We could use this to create a family of data types which solve consensus for an arbitrary number of processes, with a direct cost trade-off. If we maintain a rolling cache of several consecutive mutators, we trade off the size of the cache we maintain against the number of processes which can solve consensus. If we only need consensus for a few processes, we know we only need to maintain a small cache. If we have the available capacity to maintain a large cache, we can solve consensus for a large number of processes.

We begin by defining the sensitivity of these large-target operation sets, and giving a consensus algorithm for them. In effect, the algorithm watches for the target to fill up, and as long as it is not full, can determine which process was first. Since we can only see instances as long as the target "window" does not overflow, this gives the maximum number of processes which can use this algorithm to solve consensus. We later show this number is tight, if there are no mixed operations.

▶ **Definition 15.** A subset OPS of the operations of a data type T is *l-consecutive-k-end-sensitive* for fixed integers l and k if OPS is S-sensitive where for every $\rho \in \ell_T$, $S(\rho)$ is the sequence of l consecutive mutator instances in ρ, the last of which is the kth-last mutator instance in ρ. If there are not that many mutator instances in ρ, the missing ones are replaced by $\perp(\perp, \perp)$ in $S(\rho)$.

▶ **Theorem 16.** *Any data type with an l-consecutive-k-end-sensitive set of operations has consensus number at least l, using Algorithm 3.*

We will show that this is the maximum possible number of processes for which we can give an algorithm which solves consensus using any l-consecutive-k-end-sensitive operations set. We do this by considering a special case of that class, l-consecutive-0-end-sensitive with only pure operations, and showing that the bound is tight for it. As with most end sensitive classes, a set of operations which satisfies the definition of l-consecutive-k-end-sensitive may also be sensitive to more, earlier operations, and thus have a higher consensus number. We will show a particular example of such an operation set, to show that there is more work to be done to classify end-sensitive data types.

Theorem 17 below shows an upper bound on the consensus number of strictly l-consecutive-0-end-sensitive operation sets. That is, operation sets in which accessors can learn exactly the

Algorithm 3 Consensus algorithms for l processes using an l-consecutive-k-end-sensitive operation set. (A) Using mutator OP and pure accessor AOP. (B) Using mixed operation BOP.

(A)

```
1: for x = 1 to k do
2:     OP(input)
3:     vals[1..l] ← decode(AOP())
4:     let m = arg min_{n∈1..l}{vals[n] ≠ ⊥}
5:     if m exists then
6:         decide(vals[m])
7:     end if
8: end for
```

(B)

```
1: for x = 1 to k do
2:     vals[1..l] ← decode(BOP(input))
3:     let m = arg min_{r∈1..l}{vals[r] ≠ ⊥}
4:     if m exists then
5:         decide(vals[m])
6:     end if
7: end for
8: decide(input)
```

last l mutators. To achieve this bound, we need to restrict ourselves to operation sets which have no mixed accessor/mutator operations. This is a strong restriction, but we will give an example showing that a mutator which also returns even a small amount of information about the state of the shared object can increase the consensus number of an operation set. The proof of Theorem 17 is given in the full version of the paper.

▶ **Theorem 17.** *Any data type with a strictly l-consecutive-0-end-sensitive set of operations which has no mixed accessor/mutators has consensus number at most l.*

There are sets of operations which are strictly l-consecutive-0-end-sensitive, but have a mixed operation which returns information about the state of the object. We here give an example such set. Specifically, the mixed operation returns a (limited) count of the number of preceding mutators. Even this small amount of extra information is enough to increase the consensus power of a set of operations.

Consider an l-element shared cyclic queue with operations $Enq_l(x)$ and $ReadAll()$. $Enq_l(x)$ is a mixed accessor/mutator which adds x to the tail of the queue, discarding the head element if there are more than l elements in the queue, and returning the number of Enq_l operations which have previously been executed, up to l. If more than l Enq_l operations have been previously executed, the return value will continue to be l. $ReadAll()$ is a pure accessor which returns the entire contents of the l-element queue. This is clearly a strictly l-consecutive-0-end-sensitive set of operations, since the return values of $ReadAll()$ and Enq_l depend on the last l $Enq_l(x)$ calls, but only the last l are visible to each instance of one of these. We show that it has consensus number at least $l+1$ by giving Algorithm 4.

The intuition for this algorithm is that all processes but one will be able to see which process was first. The variable *state* will tell how many previous Enq_l instances processes have executed. If this is less than k, all previous Enq_ls are visible, and the process can return the input of the first. If there have been k previous Enq_ls, then we cannot see the first, but we know that there are at most $l+1$ processes and each executed only one Enq_l, so the one process whose Enq_l we cannot see must have been first, and we decide that process' input.

This algorithm shows that mixed operations can give extra strength for consensus, beyond sensitivity, which is difficult to quantify. In general, mixed operations can not only give different return values based on the state of the shared object, but can alter the way they modify the object's state based on its previous state. This allows them to preserve any non-empty state, which means that it can keep a record of which process first modified

Algorithm 4 Algorithm for each process i to solve consensus for $l+1$ processes using a l-element cyclic queue with Enq_l and $ReadAll$

```
1: Write_i(input)                                   ▷ In a shared SWMR register
2: state ← Enq_l(i)
3: l_history ← (ReadAll())
4: if There are state values preceding i in l_history then
5:     decide oldest element in l_history
6: else
7:     j ← processor id not appearing in l_history
8:     decide Read_j()                              ▷ Value from p_j's SWMR register
9: end if
```

the state, giving a front-sensitive data type, which can solve consensus for any number of processes. For example, a *Read-Modify-Write* operation can exhibit this behavior.

5 Conclusion

We have defined a number of classes of operations for shared objects, and explored their power for solving consensus. First, we generalized, with an intuitive result, the common understanding that knowing what process acted on a shared object first allows a consensus algorithm for any number of processes. We then considered what might be possible if only knowledge about recent operations, instead of initial operations, is available.

Here, because the set of recent operations is constantly changing, we must be more precise about what knowledge is available. If operations cannot both change and view the shared state atomically, then the number of processes which can solve consensus is given by the number of consecutive changes a process can view atomically. Further, these do not need to be the most recent changes, as long as processes know how old the data they receive is.

If operations can atomically view and change the shared state, then they generally have the potential for more computational power. We show in a few cases that if an operation set has a mixed operation, then it can solve consensus for one more process than a similarly-sensitive operation set without a mixed operation. Unfortunately, mixed operations may be more expensive to implement than pure accessors or mutators, which would lead to a trade-off between computational power and operation cost.

We point out that the quantity of information learned in a single atomic step is a dominating factor in a data type's consensus power. This appears strongly in types sensitive to consecutive past mutators and weakly in the marginally greater power of mixed operations.

We summarize our results in Table 1. We have results for front-sensitive sets of operations and several subclasses of end-sensitive operation sets. Several of these classes have different consensus numbers if we allow mixed accessor/mutator operations or only allow pure accessors and pure mutators, so we separate those results. Note also that all upper bounds further assume a data type with a strictly sensitive set of operations.

In future work, we wish to fill missing entries in the above table. In addition, we wish to further explore conditions on the knowledge of the execution which operations can extract to classify more operations. More generally, the idea of exploring how information travels through the execution history of a shared object, affecting the return values of different subsequent operations in different ways, is fascinating. As currently defined, sensitivity cannot classify all possible operation sets, so an exploration of classifying and providing generic results for other shared data types is of interest.

Table 1 Summary of Upper and Lower Bounds on Consensus Numbers.

Operation Set			Lower Bounds		Upper Bounds	
			Pure	Mixed	Pure	Mixed
Front-sensitive			∞		-	
End-Sensitive	k-end:	$k > 2$	1	?	1	3
		$k = 1$	1	2	1	?
		$k = 2$	2	?	?	3
	l-consecutive-k-end		l		l $(k = 0)$	?

Another direction is to consider trade-offs between the implementation costs of shared operations and their consensus numbers. It would be interesting to develop a metric which balances an operation's cost with its computational strength. Finding minima of such a metric would be an interesting result, potentially showing the optimal cost for solving consensus for any given number of processes.

Acknowledgments. This work was supported in part by NSF grants 0964696, 1526725. We would also like to thank the anonymous reviewers for their helpful comments.

References

1 Paul C. Attie. Wait-free byzantine consensus. *Inf. Process. Lett.*, 83(4):221–227, 2002. doi:10.1016/S0020-0190(01)00334-9.

2 Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3):117–127, 1997. doi:10.1007/s004460050029.

3 Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'94, pages 363–372, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198126.

4 Wei Chen, Guangda Hu, and Jialin Zhang. On the power of breakable objects. *Theor. Comput. Sci.*, 503:89–108, 2013. doi:10.1016/j.tcs.2013.05.036.

5 Sagar Chordia, Sriram K. Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Asynchronous resilient linearizability. In Yehuda Afek, editor, *Distributed Computing – 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2013. doi:10.1007/978-3-642-41527-2_12.

6 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

7 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

8 Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG'92, Haifa, Israel, November 2-4, 1992, Proceedings*, volume 647 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 1992. doi:10.1007/3-540-56188-9_5.

9 Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000. doi:10.1137/S0097539798335766.

10 Ophir Rachman. Anomalies in the wait-free hierarchy. In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG'94, Terschelling, The Netherlands, September 29 – October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 156–163. Springer, 1994. doi:10.1007/BFb0020431.

11 Eric Ruppert. Consensus numbers of multi-objects. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC'98, Puerto Vallarta, Mexico, June 28 – July 2, 1998*, pages 211–217. ACM, 1998. doi:10.1145/277697.277736.

12 Eric Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4):1156–1168, 2000. doi:10.1137/S0097539797329439.