Restricted Stack Implementations

Matei David, Alex Brodsky, Faith Ellen Fich

Department of Computer Science, University of Toronto,
10 King's College Road,
Toronto, Canada
matei|abrodsky|fich@cs.toronto.edu

Abstract. We introduce a new object, BH, and prove that a system with one BH object and single-writer Registers has the same computational power as a system with countably many commutative and overwriting objects. This provides a simple characterization of the class of objects that can be implemented from commutative and overwriting objects, and creates a potential tool for proving impossibility results.
It has been conjectured that Stacks and Queues shared by three or more processes *are not* in this class. In this paper, we use a BH object to show that two different restricted versions of Stacks *are* in this class. Specifically, we give an implementation of a Stack that supports any number of poppers, but at most two pushers. We also implement a Stack (or Queue) shared by any number of processes, but, in which, all stored elements are the same.

1 Introduction

Stacks and Queues are important and well studied data structures. However, they are not usually available in the hardware and, to use them, one has to implement them from the basic types available in the system. If the distributed system provides Registers and objects with consensus number ∞ (such as Compare&Swap or LL/SC), wait-free Stack and Queue implementations exist, regardless of the number of processes in the system.

Since Stacks and Queues have consensus number 2, they can be implemented in a wait-free manner from Registers and any type of objects of consensus number 2 in a system with at most two processes [Her91]. No such implementations are known when the number of processes is at least three. In fact, it is conjectured that they do not exist [Li01,Dav04b]. Proving this negative result would also solve Herlihy's long-standing open question regarding the ability of Fetch&Add objects to implement every other consensus number 2 object in systems with more than two processes.

In this paper, we consider the problem of implementing wait-free Stacks and Queues in systems where only commutative and overwriting objects (such as Test&Set objects, Fetch&Add objects, Swap objects, and Registers) are available. Two operations *commute* if the order in which they are applied does not change the resulting state of the object. One operation *overwrites* another if applying this operation results in the same object state whether or not the other operation is applied immediately before it. *Commutative and overwriting objects* are objects such that every pair of operations performed by different processes either commute or one overwrites the other. All of them have

consensus number at most 2 [Her91]. The class of all commutative and overwriting read-modify-write objects with consensus number 2 is called Common2. Many objects in this class are provided in real systems.

Afek, Weisberger, Weisman [AWW93] prove that any Common2 object shared by any number of processes can be implemented from Registers and any type of objects of consensus number 2. Hence, if a Queue or Stack can be implemented from Registers and Common2 objects, it can be implemented from Registers and any type of objects of consensus number 2. Proving that such an implementation is impossible would imply that characterizing an object to be of consensus number 2 is insufficient to describe its computational power in systems of more than 2 processes.

Attempts to prove the impossibility of such an implementation for Queues have resulted in the development of a number of restricted implementations of Queues from Registers and Common2 objects. Specifically, there are wait-free implementations of Queues shared by one or two dequeuers and any number of enqueuers [HW90,Li01], and wait-free implementations of Queues shared by one enqueuer and any number of dequeuers [Dav04a].

Another natural restriction is to consider Stacks and Queues with domain size 1, i.e., where all the elements stored in the Stack or Queue are the same. Note that single-valued Stacks and Queues behave identically. Push and Enqueue increase the number of stored elements by one. Pop and Dequeue decrease the number of stored elements by one, if there was at least one, and return whether or not this was the case. We prove that a single-valued Stack shared by any number of pushers and poppers can be implemented from Registers and Common2 objects. This means that the difficulty of implementing a general Stack is not simply coordinating pushers and poppers so that they can all complete their operations, but must involve poppers determining the order in which the steps of different pushers are linearized.

We obtain our implementation by first constructing an implementation of a Stack with arbitrary domain shared by one pusher and any number of poppers. Then we show how to transform it to obtain an implementation of a single-valued Stack shared by any number of pushers and poppers. We also show how to extend the number of pushers from one to two when the domain is arbitrary. In contrast, it is not known how to implement a Queue shared by two or more enqueuers and any number of dequeuers from commutative and overwriting objects.

The implementations in this paper do not directly use Common2 objects. Instead, we introduce a new object, BH, with a single operation, Sign, and we show that, for any number of processes, a BH object can be implemented from a single Fetch&Add object. Then we implement our Stacks from a single BH object and one single-writer Register per pusher. The form of our implementations is very simple: To perform a Push, a process appends information to its single-writer Register, and performs one or two Sign operations (depending on the implementation). To perform a Pop, a process appends information to its single-writer Register, performs two Sign operations, and then collects the single-writer Registers of all pushers.

We also show that any countably infinite collection of Fetch&Add objects and single-writer Registers can be simulated using one BH object and one single-writer Register per process. In this case, a process can perform any operation by appending the

operation and its arguments to its single-writer Register, applying one Sign operation to the BH object, and then reading the single-writer Registers of all other processes. Thus, a system with one BH object and one single-writer Register per process, and a system with Common2 objects and Registers are *equally powerful*. In particular, to show that an object cannot be implemented from Registers and objects in Common2, it suffices to prove that it has no implementation from one BH object and one single-writer Register per process. Moreover, it suffices to prove the lower bound for a restricted class of implementations in which each operation is simulated by an algorithm with a fixed, very simple form. This restriction enables us to better understand the flow of information between processes and to analyze the interaction between them.

In Section 2, we discuss the BH object and its properties. Section 3 contains our stack implementations. Throughout the paper, we assume that all objects are deterministic and linearizable and we consider only wait-free implementations.

2 The BH object

In this section, we define the BH object, show how to implement it using a single Fetch&Add object, and show how it can be used to implement any collection of Common2 objects. Our goal is to show the existence of such implementations. We do not address their efficiency.

2.1 Definition of BH

Consider an object with only one operation, in which a process appends its own ID to a shared log. We refer to an occurrence of a process ID in the log as a *signature*. We assume process IDs are positive integers, so a list of signatures is a finite sequence of positive integers. The object keeps, by means of its internal state, a complete ordered list of signatures. As a process signs the log, that process receives, in response, the entire list of signatures, including the one being applied by its current operation. This object has consensus number ∞, because processes can decide on the input value of the process whose ID is the first signature in the log. Hence, this does not capture the limited power of a system with Registers and Common2 objects.

Informally, a BH object, short for *Blurred History*, works much like the object described above, but it is restricted so that it can be implemented from Registers and Common2 objects. As before, the object has one operation, Sign, and the state of the BH object is the complete list of signatures applied so far. However, the response a process P_a gets from Sign is not the exact state σ of the BH object, but instead, a set of sequences *indistinguishable* (in the sense defined below) to P_a from σ.

Two sequences of integers σ, σ' are *a-indistinguishable* if there exist $b \neq c$, both different from a, such that $\sigma = \sigma_1 \cdot bc \cdot \sigma_2$, $\sigma' = \sigma_1 \cdot cb \cdot \sigma_2$ and neither b nor c appears in σ_2. In other words, σ' is exactly the same as σ, except for the last consecutive occurrences of two different elements other than a, which are swapped. Two sequences σ, σ' are also a-indistinguishable if there is a sequence σ'' which is a-indistinguishable from both. Thus, a-indistinguishability is transitively closed. We use the terms a-indistinguishable and indistinguishable to P_a to refer to the same relation.

To provide further intuition, we also give a direct, yet equivalent, definition for the notion of indistinguishability. We can view a state σ as providing two types of information:

- the number of signatures by each process, and
- for each signature, which signatures precede it (and, hence, which signatures follow it).

Let σ be the state of a BH object immediately after P_a performs Sign. In response to its operation, P_a will receive the following information from σ:

- the number of signatures in σ by each process, and
- the relative order of each pair of signatures in σ, provided that at least one of the signatures in the pair is by P_a or is followed by another signature by the same process.

Hence, P_a won't be able to tell the *relative order* of the last signatures by other processes, when those signatures are consecutive. For example, if the BH object is in state 123 and P_1 applies Sign, the response will be $\{1231, 1321\}$, which we can write as $1\{23\}1$. As a more elaborate example, if the BH object is in the state 1324451671718 and P_9 applies Sign, P_9 will get the response $1\{23\}4\{45\}1671\{178\}9$. Notice that, in this example, P_9 can derive the exact location of the last (and only) signature by P_6 because it knows the location of the two surrounding signatures (the second by P_1 and the first by P_7).

When two sequences σ, σ' are a-indistinguishable, one is a permutation of the other, and the set of locations of last signatures by processes other than P_a is the same in both. From the response to a Sign operation, P_a can compute the response of any previous Sign operation by some other process P_b, except for possibly the last operation by P_b. To see this, note that P_a knows the location of any signature of P_b except possibly the last signature of P_b, and furthermore, later steps (by P_b and by other processes) can only add information about the exact state at the end of P_b's operation. For example, if P_1 receives the response $124\{24\}353151$ to a Sign operation, it can see that the response P_5 got from its first Sign operation is $124\{234\}5$. In this example, P_1 knows the location of the first signature by P_3, but it can see that P_5 couldn't have had that information from the response to its first Sign.

2.2 Implementing a BH object

In this section, we informally explain how to implement a BH object using a Fetch&Add object. A more formal description of this implementation appears in [Dav04b]. The initial state of the BH object is an empty sequence, and the initial value of the Fetch&Add object in our implementation is 0.

We can view the value V stored in the Fetch&Add object as an infinite sequence of bits, $\{b_i \mid i = 0, 1, \ldots\}$. Let N denote the number of processes in the system and, for $i = 1, \ldots, N$, let V_a denote the infinite subsequence of bits $\{b_j \mid (j \bmod N) + 1 = a\}$. Then $V_1, \ldots, V_N$ are mutually disjoint. At any point in time, V_a encodes a finite sequence of non-negative integers as the concatenation of the unary representations of these integers,

each integer separated from the next by 10, and followed by an infinite sequence of 0's. For example, u_1, u_2, u_3 is encoded as $1^{u_1+1}01^{u_2+1}01^{u_3+1}00\cdots$. Then any positive integer can be appended to the end of the sequence by only changing certain bits of V_a from 0 to 1.

We implement every Sign operation by P_a using one Fetch&Add operation on V that appends a number to the sequence encoded in V_a. Since P_a is the only process changing V_a, it can keep the value of V_a in a local register v_a. Whenever P_a needs to append a number to the sequence encoded in V_a, it can inspect v_a to decide which bits of V_a have to be changed from 0 to 1. P_a can then set those bits using a Fetch&Add operation on V with an appropriate argument. For example, if V_a stores $2, 0, 3$, encoded as $111010111100\cdots$, and P_a needs to append the value 1 to this sequence, it has to change the 12-th and 13-th bits of V_a from 0 to 1. P_a can accomplish this by performing a Fetch&Add operation on V with argument $2^{a-1+11N} + 2^{a-1+12N}$.

In our BH implementation, every process P_a has, in addition to v_a, a second local register w_a. This register has initial value 0. It is used to store the last value received by P_a from a Fetch&Add operation on V. A Sign operation by P_a is implemented as follows:

- using v_a and w_a, process P_a computes a value x such that performing a Fetch&Add operation on V with argument x has the effect of appending w_a to the sequence encoded in V_a;
- P_a appends w_a to v_a;
- P_a performs Fetch&Add on V with argument x;
- P_a stores the response from this Fetch&Add operation in w_a;
- using w_a, process P_a computes the response from Sign.

The computation of x was described above. Thus, it remains to explain how to compute the response from the Sign operation.

From the response to its Fetch&Add operation, P_a can compute the number of previous signatures by some other process P_b as the number of runs of 1's in the sequence V_b. It can also compute $u_{b,i}$, the i-th non-negative integer in the sequence encoded by V_b. Note that $u_{b,i}$ is the response P_b received from the Fetch&Add operation it performed during its $(i-1)$-st Sign operation. Hence, P_a can compute which signatures precede the $(i-1)$-st signature by P_b. The only information about the signature log that P_a cannot compute is the relative order of the last signatures by other processes, when those signatures are consecutive. This is precisely the information needed to construct the class of states indistinguishable to P_a from the signature log.

2.3 Implementations using a BH object

In this section, we show that a system with one single-writer Register per process and one BH object can be used to simulate a system with infinitely many Common2 objects and Registers. To do that, we implement a countably infinite collection of Fetch&Add objects and single-writer Registers using one single-writer Register per process and one BH object. Our claim follows from the fact that any Register can be implemented from single-writer Registers [VA86], and that any Common2 object can be implemented from Fetch&Add objects and Registers [AWW93].

Consider a system of countably infinitely many Fetch&Add objects and single-writer Registers. Assume the objects in this system are indexed by positive integers. A process may perform three types of high-level operations: Fetch&Add(k,x), if k is the index of a Fetch&Add object, Read(k), if k is the index of a Register, and Write(k,x), if k is the index of one of the Registers to which it may Write. In a system with one single-writer Register per process and one BH object, we implement each of the three types of operations as follows:

- P_a appends the current high-level operation to its Register;
- P_a Signs the BH object;
- P_a Reads the Registers of all processes;
- P_a locally computes the result of the implemented operation.

Throughout the implementation, the value held in P_a's Register is an ordered list of all the high-level operations that P_a has started. We linearize a high-level operation at the moment the process executing it Signs the BH object. Thus, given the responses P_a gets from its Sign and Read operations, P_a can compute which high-level operations have occurred so far. It can also compute the linearization of these operations, except for what is blurred in the response it gets from the BH object. This information is enough for P_a to compute the result of its high-level operation:

- If the high-level operation is a Write, its response is simply OK.
- If the high-level operation is Read(k), we know that only one process P_b writes to the single-writer Register with index k. In this case, P_a returns the argument of the last Write operation by P_b to this Register that is linearized before this Read. If there is no such Write operation, then the initial value of the Register is returned.
- If the high-level operation is Fetch&Add(k,x), P_a needs to compute the sum of the arguments of all the Fetch&Add operations on this object that are linearized before the current operation. Note that P_a does not need to know the order in which these operations are linearized, since addition is commutative.

Something stronger can be said about a system with one single-writer Register per process and one BH object.

Theorem 1. *Let $\mathcal{S}_1$ be a system with countably infinitely many Common2 objects and Registers. Let $\mathcal{S}_2$ be a system with one single-writer Register per process and one BH object. If there exists an implementation of some object O in $\mathcal{S}_1$, then there exists an implementation of O in $\mathcal{S}_2$. Furthermore, to perform a high-level operation on O, process P_a begins by appending this operation to its single-writer Register and then alternately performs Sign and Reads of all Registers.*

In the construction, a process uses its single-writer Register to record which operation it performs, on which object it performs this operation, and any parameters of this operation. Thus, when implementing a single object that supports a single operation that takes no parameters, the single-writer Registers are not needed.

3 Stack Implementations from a BH object

3.1 A single-pusher Stack implementation

In this section, we give a single-pusher many-popper Stack implementation from one BH object B and an unbounded array V of single-writer Registers, each capable of holding one Stack element and each of which can only be written by the single pusher P_1. Alternatively, V can be a single-writer register capable of holding any sequence of Stack elements. The initial state of B is the empty sequence. Let process P_a be a popper, for $a > 1$. The implementation is presented in Figure 1.

The pusher P_1 holds a local variable *last*, initialized to 0, which is used to store the index of the last slot of V to which P_1 wrote. To push an element x onto the Stack, P_1 increments *last* and writes x into $V[last]$. P_1 then signs B. A signature of P_1 in B is called a *push step*. Recall that this is an occurrence of 1 in the state of B.

To pop an element off the Stack, P_a first performs two Sign operations on B. From the result of its second operation, which is an equivalence class of a-indistinguishable sequences, P_a selects any representative σ. Then P_a locally computes a function f of σ (see Figure 1). The value of this function is either 0, in which case P_a returns ε, indicating an empty Stack, or a positive integer, which P_a uses to index V. In the second case, the value stored in that location of V is the result of P_a's Pop. The signatures of poppers in B are called *pop steps*. The signature produced by the first Sign occurring in a Pop operation is called a *first pop step*, and the signature produced by the second Sign is called *a second pop step*.

The heart of this implementation is the function f. It takes a BH state σ as input, and decides which value the process computing it should pop from the Stack. Inside the function, we consider each Push operation ϕ, starting with the latest, and try to match it with the earliest completed Pop operation α that starts after the push step of ϕ. If no such α exists, we erase ϕ from σ and continue. On the other hand, if α exists, we erase both α and ϕ from σ and continue. If α turns out to be the Pop operation that invoked f on σ, which is the case if the second pop step of α is the last signature in σ, we decide that α should return the value pushed on the Stack by ϕ.

For the purposes of proving the correctness of this implementation, it will be convenient to assume that P_1 is pushing the values $1, 2, 3, \ldots$, thus identifying the value stored in a cell of V with the index of that cell.

A crucial fact in proving the correctness of this algorithm is given in Lemma 5, where we show that the choice of σ made in line 6 does not affect the output of a Pop operation. Specifically, the result of applying f to two indistinguishable states is the same. In order to establish this result, we prove several Lemmas saying that, under certain conditions, swapping two consecutive steps of σ does not change the result of f. The last pop step in σ, which is the second pop step of the Pop operation that invoked f, is never moved.

Let $\sigma = \tau_1, a, b, \tau_2$ and $\sigma' = \tau_1, b, a, \tau_2$, where $a \neq b$ and τ_2 is not empty. Lemmas 1, 2 and 3 describe situations in which $f(\sigma) = f(\sigma')$.

Lemma 1. *Swapping two consecutive pop steps by different processes, of which at least one is a first pop step, does not affect the result of f. Formally, if a is a first pop step and b is a pop step, then $f(\sigma) = f(\sigma')$.*

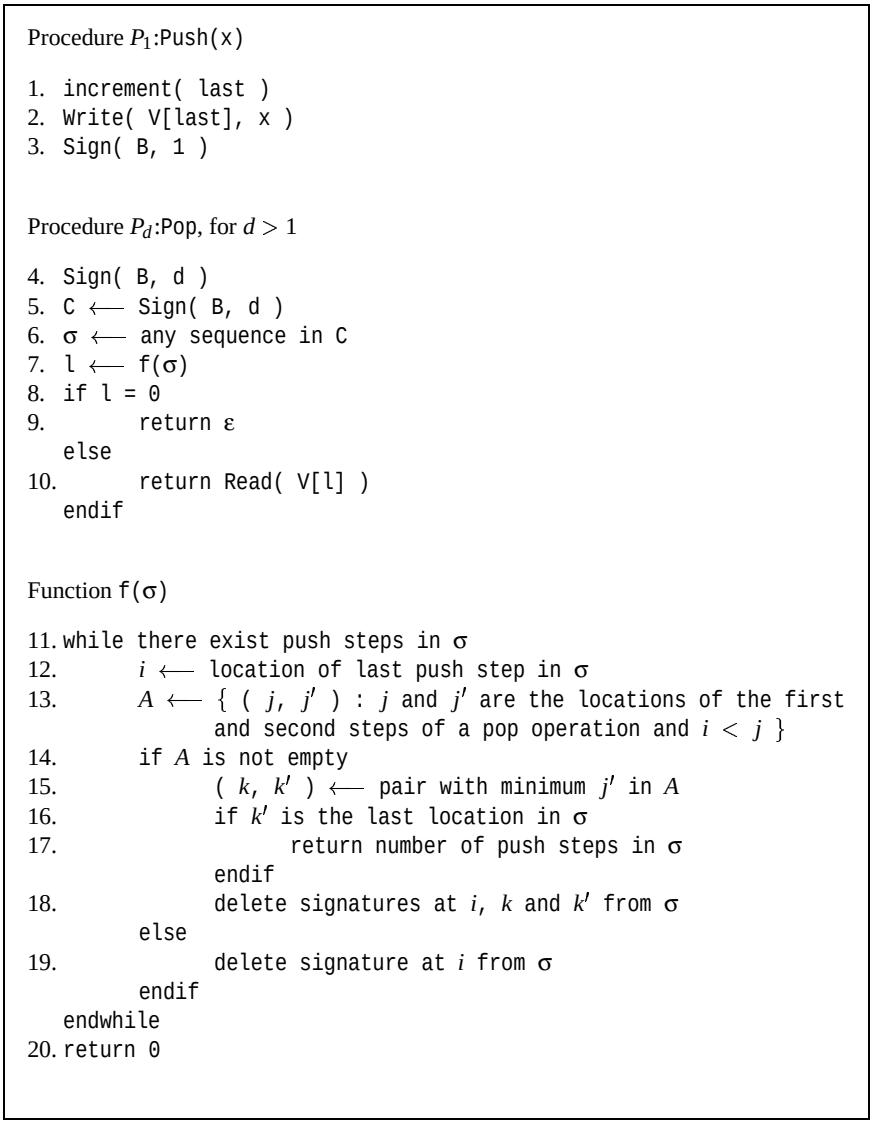

Fig. 1. A Single-Pusher Implementation

Proof. During every iteration of the `while` loop, membership in A is determined in line 13 by the order between push steps and first pop steps, and the selection of a pop operation in line 15 is determined by the order between second pop steps. Hence, the computations of f on σ and σ' take exactly the same decision during every iteration of the `while` loop.

The same argument can be used to show:

Lemma 2. *Swapping a consecutive push step and second pop step does not affect the result of f. Formally, if a is a push step and b is a second pop step, $f(\sigma) = f(\sigma')$.*

Lemma 3. *Swapping two consecutive second pop steps does not affect the result of f. Formally, if both a and b are second pop steps, $f(\sigma) = f(\sigma')$.*

Proof. We use induction on the number of executions of the `while` loop to show that $f(\sigma) = f(\sigma')$.

The only difference in the computations of f on σ and σ' can arise in an iteration in which both pop operations involved in the swap are in the set A, one of them is selected in $f(\sigma)$ and the other is selected in $f(\sigma')$. Let τ and τ' be the respective sequences at the beginning of that iteration. Without loss of generality, we must have

$$\tau = \tau_1, 1, \tau_2, a_1, \tau_3, b_1, \tau_4, a_2, b_2, \tau_5 \text{ and}$$
$$\tau' = \tau_1, 1, \tau_2, a_1, \tau_3, b_1, \tau_4, b_2, a_2, \tau_5,$$

where the 1 following τ_1 is the last step by the pusher P_1. Hence $\tau_2, \tau_3, \tau_4, \tau_5$ contain no 1's. The steps a_1 and a_2 are the first and second steps of a pop operation by P_a, and b_1 and b_2 are the first and second steps of a pop operation by P_b. Notice that in this case τ_3 cannot contain pop steps by P_a because P_a has a pending operation.

In this scenario, $1, a_1, a_2$ are deleted in $f(\sigma)$ and $1, b_1, b_2$ are deleted in $f(\sigma')$. Let

$$\overline{\tau} = \tau_1, \tau_2, \tau_3, b_1, \tau_4, b_2, \tau_5 \text{ and}$$
$$\overline{\tau}' = \tau_1, \tau_2, a_1, \tau_3, \tau_4, a_2, \tau_5.$$

Then $f(\sigma) = f(\tau) = f(\overline{\tau})$ and $f(\sigma') = f(\tau') = f(\overline{\tau}')$. The computation of f is not affected by what popper is performing a particular Pop operation. It is only affected by the locations of pop steps in the sequence. Hence, $f(\overline{\tau}) = f(\overline{\tau}'')$, where

$$\overline{\tau}'' = \tau_1, \tau_2, \tau_3, a_1, \tau_4, a_2, \tau_5.$$

Since τ_3 contains no pop steps by P_a and no push steps, $\overline{\tau}'$ can be transformed into $\overline{\tau}''$ by repeatedly swapping the first pop step a_1 with pop steps that immediately precede it. By Lemma 1, $f(\overline{\tau}') = f(\overline{\tau}'')$.

Lemma 4. *Removing the first step of an incomplete Pop does not affect the result of f.*

Proof. The first step of an incomplete pop operation is never considered when building the set A, nor when selecting a pop operation from A, so removing it will cause no change in the computation of f.

Lemma 5. *Let σ be the BH state at the end of a pop operation by some process P_d. Let σ' be a sequence indistinguishable to P_d from σ. Then $f(\sigma) = f(\sigma')$.*

Proof. By properties of the BH object, there is a sequence of states $\sigma^{(0)}, \sigma^{(1)}, \ldots, \sigma^{(m)}$ with $\sigma = \sigma^{(0)}$ and $\sigma^{(m)} = \sigma'$ such that any two consecutive states $\sigma^{(e)}, \sigma^{(e+1)}$ can be obtained from one another by swapping two consecutive last steps by some processes other than P_d. We have three possibilities:

- One of these steps is a first pop step. Since it is the last step by that process, it must be part of an incomplete pop operation. By Lemma 4, removing it will not affect the result of f. But removing it erases the difference between $\sigma^{(e)}$ and $\sigma^{(e+1)}$, so $f(\sigma^{(e)}) = f(\sigma^{(e+1)})$.
- Both steps are second pop steps. By Lemma 3, $f(\sigma^{(e)}) = f(\sigma^{(e+1)})$.
- One is a push step, the other is a second pop step. By Lemma 2, $f(\sigma^{(e)}) = f(\sigma^{(e+1)})$.

Inductively, $f(\sigma) = f(\sigma')$.

Next, we assign linearization points for Push operations and for completed Pop operations. We do not linearize any incomplete Pop operations (which only apply one Sign). A Push operation is linearized at its push step. Let α be a complete Pop operation and let σ_α be the BH state when α is completed. We define the linearization point of α as follows:

- If there are no Push operations deleted unmatched (i.e. on line 19) during the computation of f on σ_α, then α is linearized at its second pop step.
- Otherwise, α is linearized immediately before the Push operation ϕ deleted on line 19 whose push step occurs earliest in σ_α. If multiple Pop operations are linearized at the same place, they are put in the same order that their second pop steps appear in σ_α.

Note that, in the second case, the push step of ϕ occurs between the two pop steps of α: Since ϕ occurs in σ_α, the second pop step cannot occur before the push step. If the first pop step occurs after the push step, then A is not empty at the end of the first iteration of the computation of f on σ_α and ϕ is deleted on line 18, rather than line 19.

Furthermore, if $\tau_0, d_1, \tau_1, 1, \tau_2, d_2$ is the BH state when the Pop operation α completes, d_1 and d_2 are the two pop steps of α, and τ_2 does not contain any push step or both pop steps of any Pop operation, then α is not linearized at its second pop step. This follows from the fact that A is empty during the first iteration of f on $\tau_0, d_1, \tau_1, 1, \tau_2, d_2$ and, hence, the last Push operation is deleted unmatched.

Given σ, we define $h(\sigma)$ to be the Stack history associated with σ. It contains the sequence of operations in the order they are linearized, together with their return values. For example, $h(11216264241266)$ is the sequence

(Push, OK), (Push, OK), (Pop by P_2, 2), (Push, OK), (Pop by P_6, 3),
(Pop by P_4, 1), (Pop by P_2, ε), (Push, OK), (Pop by P_6, 4).

Theorem 2. *For every state σ, the Stack history $h(\sigma)$ is legal.*

Proof. We use induction on the number of push steps in σ.

First, let σ be a history with no push steps. Any Pop operation which is completed during σ will output ε, hence $h(\sigma)$ is legal.

Now let $k \geq 0$ and assume that, for all sequences σ' with at most k push steps, $h(\sigma')$ is legal. Let σ be a history with $k+1$ push steps. Let ϕ denote the last Push operation.

First, consider the case where σ contains no completed Pop operations that start after its last push step ℓ. Then $\sigma = \pi, \ell, \rho$, where ρ contains no push steps. Let $\sigma' = \pi, \rho$.

All Push operations in σ' return the same result, OK, as in σ and each is linearized in the same place in $h(\sigma')$ and $h(\sigma)$.

Any Pop operation α whose second pop step occurs before ℓ in σ is linearized before ϕ in $h(\sigma)$ Moreover, since $\sigma_\alpha = \sigma'_\alpha$, it follows that α has the same result in σ and σ' and is linearized in the same place in $h(\sigma)$ and $h(\sigma')$.

Now let α be a Pop operation whose second pop step occurs after ℓ in σ. Since there are no completed Pop operations that start after ℓ, the first pop step of α occurs in π. Then $\sigma = \tau_0, d_1, \tau_1, \ell, \tau_2, d_2, \tau_3$, where d_1 and d_2 are the pop steps of α and τ_2, τ_3 does not contain any push step or both pop steps of any Pop operation. By the observation following the definition of the linearization points, α is not linearized at its second pop step and ϕ is deleted unmatched during the computation of f on σ_α. It follows that α has the same result in both σ and σ'.

If α is not linearized immediately before ℓ, then it is linearized immediately before a push step that occurs in π and whose Push operation is also deleted during the computation of f on σ_α. Hence, α is linearized in the same place in $h(\sigma)$ and $h(\sigma')$.

Each Pop operation that is linearized immediately before ℓ in $h(\sigma)$ is linearized at its second pop step in $h(\sigma')$. These operations are linearized after the last Push operation in $h(\sigma')$ and are linearized in the same relative order as they are in $h(\sigma)$.

Thus $h(\sigma) = h(\sigma'), (\text{Push}, \text{OK})$. By the induction hypothesis, $h(\sigma')$ is legal. Thus, so is $h(\sigma)$.

Now consider the case where σ contains at least one completed Pop operation that starts after the last push step ℓ. Then $\sigma = \tau_0, \ell, \tau_1, d_1, \tau_2, d_2, \tau_3$, where d_1 and d_2 are the two steps of the first completed Pop operation α that starts after ℓ. Then τ_1, τ_2, τ_3 contains no push steps and τ_1, τ_2 does not contain both steps of any Pop operation.

Since ϕ ends before α begins, it is linearized before α. Any other (Pop) operation β that is linearized between ϕ and α must be linearized at β's second pop step, c_2. Since this occurs before d_2, the definition of α implies that β's first pop step, c_1, must occur before ℓ. Thus $\sigma_\beta = \rho_0, c_1, \rho_1, \ell, \rho_2, c_2$, where ρ_2 does not contain any push step or the both pop steps of any Pop operation. But then the observation following the definition of the linearization points says that β is not linearized at c_2, which is a contradiction. Thus, there are no operations linearized between ϕ and α in $h(\sigma)$.

During the first iteration of the computation of f on $\sigma_\alpha = \tau_0, \ell, \tau_1, d_1, \tau_2, d_2$, variable i contains the location of ℓ and, by the choice of α, variables k and k' contain the locations of α's first and second pop steps. Thus f returns the number of push steps in σ_α, which is the index of the location in V to which ϕ writes. By Lemma 5, the sequence

that is chosen on line 6 during α gives the value for f as σ_α. Hence α returns the value pushed by ϕ.

Now we argue that removing both ϕ and α does not change the relative order of the linearization points of the the remaining operations nor the results of these operations. Let $\sigma' = \tau_0, \tau_1, \tau_2, \tau_3$. Note that the linearization points of each Push operation (except ϕ) is the same in σ' as in σ. We consider a number of different cases for completed Pop operations.

A Pop operation whose second pop step is in τ_0 has $\sigma_\alpha = \sigma'_\alpha$. Thus it is linearized at the same point and returns the same value in σ and σ'.

If a Pop operation β has its second pop step in τ_3, then, during the first iteration of the computation of f on σ_β, ϕ and α are matched and deleted on line 18, leaving σ'. Thus β has the same linearization point in σ as it does in σ' and returns the same value.

Finally, consider a Pop operation β whose second pop step is either in τ_1 or τ_2. Since ϕ is matched with α during the computation of f on σ_α and σ_β is a proper prefix of σ_α, ϕ is deleted unmatched during the first iteration of the computation of f on σ_β. The only difference between the resulting sequence and σ'_β is the first pop step d_1 of α, which, by Lemma 4 does not affect the result of f. Hence, β returns the same value in $h(\sigma)$ and $h(\sigma')$.

By definition of α, β's first pop step is in τ_0. Since no operations are linearized between ϕ, which is linearized at ℓ, and α, which is linearized at d_2, β is not linearized at its second pop step in σ. Hence it is linearized immediately before some push step. If that push step is not ℓ, then β has the same linearization point in σ', since the same Push operations are deleted unmatched in the computations of f on σ_β and σ'_β.

The only other Pop operations are those whose second pop steps are in τ_1, τ_2 and which are linearized immediately before ℓ in σ. They are linearized in order of their second pop steps. Since the last Push operation, ϕ, is the earliest unmatched Push operation in the computation of f on σ_β, it must be the only unmatched Push operation. Thus, in σ'_β, there are no unmatched Push operations, so in σ', these operations are linearized at their second pop steps. Hence, they have the same relative order in σ and σ'. Since the linearization point of all other operations are in τ_0 or in τ_3, all operations in $h(\sigma')$ occur in the same order in $h(\sigma)$.

Since we show that each operation in $h(\sigma')$ returns the same result as it does in $h(\sigma)$. It follows that $h(\sigma)$ is exactly equal to $h(\sigma')$ with an inserted pair of consecutive operations, the Push ϕ and the matching Pop α. By the induction hypothesis, $h(\sigma')$ is legal, so, from the specifications of a Stack object, $h(\sigma)$ is also legal.

3.2 A single-valued Stack implementation

The single-popper Stack implementation is based on the observation that the number of times each pusher signs the BH object prior to a Pop is precisely the number of elements that were pushed on the Stack prior to that Pop. If there is only one pusher, there is no ambiguity about the order in which the Push operations occurred. Unfortunately, this is not the case when there are many pushers. For example, suppose processes P_1 and P_2 each pushed a value on the Stack by signing the BH object and then process P_3 popped a value by signing the BH object twice. The resulting state 1233 of the BH object is

indistinguishable to P_3 from 2133, the state that results when P_1 and P_2 perform their operations in the opposite order. Consequently, it is not clear if the value pushed by P_1 or P_2 is the one which should be popped. While we can overcome this problem for the special case of exactly two pushers (see following section), the general solution remains elusive. However, if all the values pushed on the Stack are the same, then the problem of choosing which value to match with which Pop is obviated.

A process performing a Pop on a single-valued Stack only needs to determine whether or not its Pop operation has some matching Push. It does not matter which pusher performed the Push. This is essentially the problem that is solved by the single-pusher Stack implementation (in the previous section).

To perform a Push, a process appends 1 to its single-writer Register and signs the BH object once. To perform a Pop, a process appends 2 to its single-writer Register, signs the BH object twice, and then reads the Registers of all other processes. Let C denote the equivalence class of BH states returned as a result of the second Sign operation in a Pop. As in line 6, we select any representative σ from C. However, before we compute f on σ, we replace every push step in σ with a push step by a virtual process, P_0. A step by some process P_a is a push step if the corresponding value in P_a's Register is a 1. If f returns 0 on the modified sequence, the Pop returns ε; otherwise the Pop returns the single value in the domain.

The proof of correctness is essentially the same as the the proof for the single-pusher Stack, except for the addition of the following lemma, which handles two Push operations whose order cannot be distinguished.

Lemma 6. *Swapping two consecutive push steps does not affect the result of f.*

3.3 A two-pusher Stack implementation

We will now extend the algorithm given in Section 3.1 to allow two pushers instead of just one. The basic idea is similar to the "helping" mechanism that appears in Herlihy's universal construction [Her91]: the completion of a Push operation by one pusher might "help" linearize a pending Push operation by the other pusher.

Let P_1 and P_2 be the two pushers, and let P_d be a popper, for $d > 2$. We assume that, in addition to a BH object B, we have two unbounded arrays V_1, V_2 of single-writer Registers, where V_a is written by pusher P_a. To push the value x, P_a first writes x in the next available location in V_a. P_a then applies *two* Sign operations on B. Recall that in the single-pusher implementation, a Push operation consisted of only one Sign.

A Pop operation by P_d begins by applying two Sign operations on the BH object. The return value of the second operation is an equivalence class of states indistinguishable to P_d from the real state of B. We then select any representative σ, as in line 6. However, before we can apply function f on σ, we need to transform σ from a two-pusher history into a single-pusher history. This transformation is performed by a new function, g, described below.

The function g takes as arguments a two-pusher history σ, and the arrays V_1, V_2. It constructs a single-pusher history τ and an array V. The two histories, σ and τ, contain exactly the same pop steps. The push steps by P_1 and P_2 in σ are replaced in τ with push steps by a virtual process, P_0. The idea is that a Push operation ϕ is linearized

either at its second step, or at the second step of the first push operation ϕ' by the other pusher which was started and completed after the first step of ϕ. The function g can be computed as follows.

- Find the earliest second push step in σ; call that push operation ϕ'.
- If there is a push operation ϕ which has a first step that occurs before the first push step of ϕ', delete both ϕ and ϕ', and insert two steps by P_0 in τ at the location of the second push step of ϕ'.
- If no such ϕ exists, delete ϕ' and insert a step by P_0 in τ at the location of the second push step of ϕ'.
- Whenever we delete the i-th Push operation by P_a, write $V_a[i]$ into the first empty location in V. In the first case, when we delete ϕ and ϕ', append the value corresponding to ϕ before the one corresponding to ϕ'.
- Repeat until no push operation in σ has two steps.
- At the end, delete any remaining first push steps.

For the purposes of proving correctness, we may assume that the i-th value pushed by P_a and written in $V_a[i]$ is the pair (a,i). For example, if $\sigma = 1\underline{1}123\underline{32}6\underline{1}124\underline{1}3\underline{32}\underline{2}4314\underline{26}$ (where second push steps and second pop steps are underlined), we have $g(\sigma, V_1, V_2) = (\tau, V)$ where $\tau = 03\underline{3}00640\underline{3}3\underline{0}\underline{4}340\underline{6}$ and V = (1,1), (1,2), (2,1), (1,3), (2,2), (2,3).

After computing $g(\sigma) = (\tau, V)$, a Pop operation computes $f(\tau)$. If the latter evaluates to 0, the Pop returns ε; otherwise the Pop returns the element in location $f(\tau)$ of V, the array computed in g. For example, for σ, τ, V from the previous example, $f(\tau) = 2$ and $V[f(\tau)] = (1,2)$.

The following two Lemmas are needed to prove the correctness of this extension.

Lemma 7. *Let σ be the state at the end of a Pop operation by P_d and let σ' be a state indistinguishable to P_d from σ. Let $g(\sigma) = (\tau, V)$ and $g(\sigma') = (\tau', V')$. Then $V = V'$ and $f(\tau) = f(\tau')$.*

Lemma 8. *Let σ be a BH state. Let σ' be any prefix of σ. Let $g(\sigma) = (\tau, V)$ and let $g(\sigma') = (\tau', V')$. Then τ' is a prefix of τ and V' is a prefix of V.*

Finally, we argue that our algorithm is linearizable. Given a two-pusher history σ, let $g(\sigma) = (\tau, V)$. We define the linearization points for Push operations in σ to be the corresponding steps where they appear in τ. We define linearization points for Pop operations in σ the same way they are defined in the single-pusher history τ. By Lemma 8, all Pop operations completed in σ have returned the exact same values as if they had occurred in τ. Since the single-pusher history τ is linearizable, so is σ.

4 Conclusions

In this paper, we have showed that it is possible to construct wait-free implementations of certain restricted Stacks using only Registers and Common2 objects. Specifically, it is possible to implement single-valued Stacks (and Queues) shared by any number of process, and general (multi-valued) Stacks shared by one or two pushers and any number of poppers.

Queue implementations exist for any number of enqueuers and at most two dequeuers [Li01], and for one enqueuer and any number of dequeuers [Dav04a]. In a Stack implementation, only the poppers output relevant values. If there are only two poppers, they might be able to agree on the sequence of values to output. This suggests that Stack implementations for any number of pushers and at most two poppers might exist. However, we conjecture that implementing a Stack with domain size 2, shared by three pushers and three poppers, is impossible to implement from Registers and Common2 objects.

Since modern distributed systems do provide more powerful types, our results are mainly of theoretical interest. The BH object is not an object one would want to implement in hardware or use in an efficient implementation. Moreover, the implementations we present use an unbounded size BH object and an unbounded number of single-writer Registers (or single-writer Registers of unbounded size).

However, we believe the BH object is a very useful tool for studying the computational power of Registers and objects in Common2. It provides a simple characterization of the information a process can obtain from such objects during the course of a computation. This makes it much easier to show the existence of algorithms for this model and has the potential of leading to the development of interesting impossibility results dealing with questions at the foundations of our understanding of shared memory distributed computing.

Acknowledgments

This research was supported by an Ontario Graduate Scholarship, the Natural Sciences and Engineering Research Council of Canada, and the Scalable Synchronization Research Group of Sun Microsystems.

References

[AWW93] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 159–170, 1993.

[Dav04a] Matei David. A single-enqueuer wait-free queue implementation. In *Proceedings of DISC 2004*, pages 132–143, 2004.

[Dav04b] Matei David. Wait-free linearizable queue implementations. Master's thesis, Univ. of Toronto, 2004.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HW90] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):495–504, January 1990.

[Li01] Zongpeng Li. Non-blocking implementation of queues in asynchronous distributed shared-memory systems. Master's thesis, Univ. of Toronto, 2001.

[VA86] Paul Vitanyi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243, 1986.