FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction – Proofs

Aleksandar Prokopec[1], Heather Miller[1], Philipp Haller[2], Tobias Schlatter[1], and Martin Odersky[1]

[1] EPFL, Switzerland
firstname.lastname@epfl.ch
[2] Typesafe, Inc.
firstname.lastname@typesafe.com

EPFL-REPORT-999999

Abstract. Implementing correct and deterministic parallel programs is challenging. Even though concurrency constructs exist in popular programming languages to facilitate the task of deterministic parallel programming, they are often too low level, or do not compose well due to underlying blocking mechanisms. In this report, we present the detailed proofs of the linearizability, lock- freedom, and determinism properties of FlowPools, a deterministic concurrent dataflow abstraction presented in [1].

Keywords: dataflow, concurrent data-structure, deterministic parallelism

1 Introduction

We first provide a summary of the lemmas and theorems introduced in the associated paper, *FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction* [1]

We define the notion of an abstract pool $\mathbb{A} = (elems, callbacks, seal)$ of elements in the pool, callbacks and the seal size. Given an abstract pool, abstract pool operations produce a new abstract pool. The key to showing correctness is to show that an abstract pool operation corresponds to a FlowPool operation– that is, it produces a new abstract pool corresponding to the state of the FlowPool after the FlowPool operation has been completed.

Lemma 1.1 Given a FlowPool consistent with some abstract pool, CAS instructions in lines 21, 63 and 66 do not change the corresponding abstract pool.

Lemma 1.2 Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 22 changes it to the state consistent with an abstract pool $(\{elem\} \cup elems, cbs, seal)$. There exists a time $t_1 \geq t_0$ at which every callback $f \in cbs$ has been called on $elem$.

```
def create()
  new FlowPool {
    start = createBlock(0)
    current = start
  }

def createBlock(bidx: Int)
  new Block {
    array = new Array(BLOCKSIZE)
    index = 0
    blockindex = bidx
    next = null
  }

def append(elem: Elem)
  b = READ(current)
  idx = READ(b.index)
  nexto = READ(b.array(idx + 1))
  curo = READ(b.array(idx))
  if check(b, idx, curo) {
    if CAS(b.array(idx + 1), nexto, curo) {
      if CAS(b.array(idx), curo, elem) {
        WRITE(b.index, idx + 1)
        invokeCallbacks(elem, curo)
      } else append(elem)
    } else append(elem)
  } else {
    advance()
    append(elem)
  }

def check(b: Block, idx: Int, curo: Object)
  if idx > LASTELEMPOS return false
  else curo match {
    elem: Elem =>
      return false
    term: Terminal =>
      if term.sealed = NOSEAL return true
      else {
        if totalElems(b, idx) < term.sealed
          return true
        else error("sealed")
      }
    null =>
      error("unreachable")
  }

def advance()
  b = READ(current)
  idx = READ(b.index)
  if idx > LASTELEMPOS
    expand(b, b.array(idx))
  else {
    obj = READ(b.array(idx))
    if obj is Elem WRITE(b.index, idx + 1)
  }

def expand(b: Block, t: Terminal)
  nb = READ(b.next)
  if nb is null {
    nb = createBlock(b.blockindex + 1)
    nb.array(0) = t
    if CAS(b.next, null, nb)
      expand(b, t)
  } else {
    CAS(current, b, nb)
  }

def totalElems(b: Block, idx: Int)
  return b.blockindex * (BLOCKSIZE - 1) + idx

def invokeCallbacks(e: Elem, term: Terminal)
  for (f <- term.callbacks) future {
    f(e)
  }

def seal(size: Int)
  b = READ(current)
  idx = READ(b.index)
  if idx <= LASTELEMPOS {
    curo = READ(b.array(idx))
    curo match {
      term: Terminal =>
        if ¬tryWriteSeal(term, b, idx, size)
          seal(size)
      elem: Elem =>
        WRITE(b.index, idx + 1)
        seal(size)
      null =>
        error("unreachable")
    }
  } else {
    expand(b, b.array(idx))
    seal(size)
  }

def tryWriteSeal(term: Terminal, b: Block,
  idx: Int, size: Int)
  val total = totalElems(b, idx)
  if total > size error("too many elements")
  if term.sealed = NOSEAL {
    nterm = new Terminal {
      sealed = size
      callbacks = term.callbacks
    }
    return CAS(b.array(idx), term, nterm)
  } else if term.sealed ≠ size {
    error("already sealed with different size")
  } else return true

def foreach(f: Elem => Unit)
  future {
    asyncFor(f, start, 0)
  }

def asyncFor(f: Elem => Unit, b: Block, idx: Int)
  if idx <= LASTELEMPOS {
    obj = READ(b.array(idx))
    obj match {
      term: Terminal =>
        nterm = new Terminal {
          sealed = term.sealed
          callbacks = f ∪ term.callbacks
        }
        if ¬CAS(b.array(idx), term, nterm)
          asyncFor(f, b, idx)
      elem: Elem =>
        f(elem)
        asyncFor(f, b, idx + 1)
      null =>
        error("unreachable")
    }
  } else {
    expand(b, b.array(idx))
    asyncFor(f, b.next, 0)
  }
```

Fig. 1. FlowPool operations pseudocode

$t ::=$	terms	$p \in \{(vs, \sigma, cbs) \mid vs \subseteq \mathit{Elem}, \sigma \in \{-1\} \cup \mathbb{N},$
`create` p	pool creation	$cbs \subset \mathit{Elem} \Rightarrow \mathit{Unit}\}$
$p << v$	append	$v \in \mathit{Elem}$
p `foreach` f	foreach	$f \in \mathit{Elem} \Rightarrow \mathit{Unit}$
p `seal` n	seal	$n \in \mathbb{N}$
t_1 ; t_2	sequence	

Fig. 2. Syntax

Lemma 1.3 Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 124 changes it to the state consistent with an abstract pool $(elems, (f, \emptyset) \cup cbs, seal)$ There exists a time $t_1 \geq t_0$ at which f has been called for every element in $elems$.

Lemma 1.4 Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 105 changes it to the state consistent with an abstract pool $(elems, cbs, s)$, where either $seal = -1 \wedge s \in \mathbb{N}_0$ or $seal \in \mathbb{N}_0 \wedge s = seal$.

Theorem 1.5 [Safety] FlowPool operations `append`, `foreach` and `seal` are consistent with the abstract pool semantics.

Theorem 1.6 [Linearizable operations] FlowPool operations `append` and `seal` are linearizable.

Lemma 1.7 After invoking a FlowPool operation `append`, `seal` or `foreach`, if a non-consistency changing CAS instruction in lines 21, 63, or 66 fails, they must have already been completed by another thread since the FlowPool operation began.

Lemma 1.8 After invoking a FlowPool operation `append`, `seal` or `foreach`, if a consistency-changing CAS instruction in lines 22, 105, or 124 fails, then some thread has successfully completed a consistency changing CAS after some finite number of steps.

Lemma 1.9 After invoking a FlowPool operation `append`, `seal` or `foreach`, a consistency changing instruction will be completed after a finite number of steps.

Theorem 1.10 [Lock-freedom] FlowPool operations `append`, `foreach` and `seal` are lock-free.

Determinism. We claim that the FlowPool abstraction is *deterministic* in the sense that a program computes the same result (which can also be an error) regardless of the interleaving of concurrent operations. Here, we give an outline of the determinism proof. A complete formal proof can be found in section 3.

The following definitions and the determinism theorem are based on the language shown in Figure 2. The semantics of our core language is defined using reduction rules which define transitions between *execution states*. An execution state is a pair $T \mid P$ where T is a set of concurrent threads and P is a set of FlowPools. Each thread executes a *term* of the core language (typically a sequence of terms). State of a thread is represented as the (rest of) the term that it still has to execute; this means there is a one-to-one mapping between threads and terms. For example, the semantics of `append` is defined by the following reduction rule (a complete summary of all the rules can be found in the appendix):

$$\frac{t = p << v \ ; \ t' \quad p = (vs, cbs, -1) \quad p' = (\{v\} \cup vs, cbs, -1)}{t, T \mid p, P \ \longrightarrow \ t', T \mid p', P} \ (\textsc{Append1})$$

Append simply adds the value v to the pool p, yielding a modified pool p'. Note that this rule can only be applied if the pool p is not sealed (the seal size is -1). The rule for $foreach$ modifies the set of callback functions in the pool:

$$\frac{\begin{array}{c} t = p \ \texttt{foreach} \ f \ ; \ t' \quad p = (vs, cbs, n) \\ T' = \{g(v) \mid g \in \{f\} \cup cbs, v \in vs\} \quad p' = (vs, \{f\} \cup cbs, n) \end{array}}{t, T \mid p, P \ \longrightarrow \ t', T, T' \mid p', P} \ (\textsc{Foreach2})$$

This rule only applies if p is sealed at size n, meaning that no more elements will be appended later. Therefore, an invocation of the new callback f is scheduled for each element v in the pool. Each invocation creates a new thread in T'.

Programs are built by first creating one or more FlowPools using `create`. Concurrent threads can then be started by (a) appending an element to a FlowPool, (b) sealing the FlowPool and (c) registering callback functions (`foreach`).

Definition 1.11 [Termination] A term t terminates with result P if its reduction ends in execution state $\{t : t = \{\epsilon\}\} \mid P$.

Definition 1.12 [Interleaving] Consider the reduction of a term t: $T_1 \mid P_1 \longrightarrow T_2 \mid P_2 \longrightarrow \ \ldots \ \longrightarrow \{t : t = \{\epsilon\}\} \mid P_n$. An *interleaving* is a reduction of t starting in $T_1 \mid P_1$ in which reduction rules are applied in a different order.

Definition 1.13 [Determinism] The reduction of a term t is *deterministic iff* either (a) t does not terminate for any interleaving, or (b) t always terminates with the same result for all interleavings.

Theorem 1.14 [FlowPool Determinism] Reduction of terms t is deterministic.

2 Proof of Correctness

Definition 2.1 [Data types] A **Block** b is an object which contains an array $b.array$, which itself can contain elements, $e \in Elem$, where **Elem** represents the type of e and can be any countable set. A given block b additionally contains an index $b.index$ which represents an index location in $b.array$, a unique index identifying the array $b.blockIndex$, and $b.next$, a reference to a successor block c where $c.blockIndex = b.blockIndex + 1$. A **Terminal** *term* is a sentinel object, which contains an integer $term.sealed \in \{-1\} \cup \mathbb{N}_0$, and $term.callbacks$, a set of functions $f \in Elem \Rightarrow Unit$.

We define the following functions:

$$following(b : Block) = \begin{cases} \emptyset & \text{if b.next = null,} \\ b.next \cup following(b.next) & \text{otherwise} \end{cases}$$

$$reachable(b : Block) = \{b\} \cup following(b)$$

$$last(b : Block) = b' : b' \in reachable(b) \wedge b'.next = null$$

$$size(b : Block) = |\{x : x \in b.array \wedge x \in Elem\}|$$

Based on them we define the following relation:

$$reachable(b, c) \Leftrightarrow c \in reachable(b)$$

Definition 2.2 [FlowPool] A **FlowPool** *pool* is an object that has a reference *pool.start*, to the first block b_0 (with $b_0.blockIndex = 0$), as well as a reference *pool.current*. We sometimes refer to these just as *start* and *current*, respectively.

A **scheduled callback invocation** is a pair (f, e) of a function $f \in Elem =>$ $Unit$ and an element $e \in Elem$. The programming construct that adds such a pair to the set of $futures$ is `future { f(e) }`.

The **FlowPool state** is defined as a pair of the directed graph of objects transitively reachable from the reference *start* and the set of scheduled callback invocations called $futures$.

A **state changing** or **destructive** instruction is any atomic write or CAS instruction that changes the FlowPool state.

We say that the FlowPool **has an element** e at some time t_0 if and only if the relation $hasElem(start, e)$ holds.

$$hasElem(start, e) \Leftrightarrow \exists b \in reachable(start), e \in b.array$$

We say that the FlowPool **has a callback** f at some time t_0 if and only if the relation $hasCallback(start, f)$ holds.

$$\begin{aligned} hasCallback(start, f) \Leftrightarrow \forall b = last(start), b.array = x^P \cdot t \cdot y^N, x \in Elem, \\ t = Terminal(seal, callbacks), f \in callbacks \end{aligned}$$

We say that a callback f in a FlowPool **will be called** for the element e at some time t_0 if and only if the relation $willBeCalled(start, e, f)$ holds.

$$willBeCalled(start, e, f) \Leftrightarrow \exists t_1, \forall t > t_1, (f, e) \in futures$$

We say that the FlowPool is **sealed** at the size s at some t_0 if and only if the relation $sealedAt(start, s)$ holds.

$$sealedAt(start, s) \Leftrightarrow s \neq -1 \wedge \forall b = last(start), b.array = x^P \cdot t \cdot y^N, \\ x \in Elem, t = Terminal(s, callbacks)$$

FlowPool operations are `append`, `foreach` and `seal`, and are defined by pseudocodes in Figure 1.

Definition 2.3 [Invariants] We define the following invariants for the **FlowPool**:

INV1 $start = b : Block, b \neq null, current \in reachable(start)$
INV2 $\forall b \in reachable(start), b \notin following(b)$
INV3 $\forall b \in reachable(start), b \neq last(start) \Rightarrow size(b) = LASTELEMPOS \wedge b.array(BLOCKSIZE - 1) \in Terminal$
INV4 $\forall b = last(start), b.array = p \cdot c \cdot n$, where:
$p = X^P, c = c_1 \cdot c_2, n = null^N$
$x \in Elem, c_1 \in Terminal, c_2 \in \{null\} \cup Terminal$
$P + N + 2 = BLOCKSIZE$
INV5 $\forall b \in reachable(start), b.index > 0 \Rightarrow b.array(b.index - 1) \in Elem$

Definition 2.4 [Validity] A FlowPool state $\mathbb{S}$ is **valid** if and only if the invariants [INV1-5] hold for that state.

Definition 2.5 [Abstract pool] An **abstract pool** $\mathbb{P}$ is a function from time t to a tuple $(elems, callbacks, seal)$ such that:

$seal \in \{-1\} \cup \mathbb{N}_0$
$callbacks \subset \{(f : Elem => Unit, called)\}$
$called \subseteq elems \subseteq Elem$

We say that an abstract pool $\mathbb{P}$ **is in state** $\mathbb{A} = (elems, callbacks, seal)$ at time t if and only if $\mathbb{P}(t) = (elems, callbacks, seal)$.

Definition 2.6 [Abstract pool operations] We say that an **abstract pool operation** op that is applied to some abstract pool $\mathbb{P}$ in abstract state $\mathbb{A}_0 = (elems_0, callbacks_0, seal_0)$ at some time t **changes** the abstract state of the abstract pool to $\mathbb{A} = (elems, callbacks, seal)$ if $\exists t_0, \forall \tau, t_0 < \tau < t, \mathbb{P}(\tau) = \mathbb{A}_0$ and $\mathbb{P}(t) = \mathbb{A}$. We denote this as $\mathbb{A} = op(\mathbb{A}_0)$.

Abstract pool operation $foreach(f)$ changes the abstract state at t_0 from $(elems, callbacks, seal)$ to $(elems, (f, \emptyset) \cup callbacks, seal)$. Furthermore:

$$\exists t_1 \geq t_0,\ \forall t_2 > t_1, \mathbb{P}(t_2) = (\mathit{elems}_2, \mathit{callbacks}_2, \mathit{seal}_2)$$
$$\wedge \forall (f, \mathit{called}_2) \in \mathit{callbacks}_2, \mathit{elems} \subseteq \mathit{called}_2 \subseteq \mathit{elems}_2$$

Abstract pool operation $\mathit{append}(e)$ changes the abstract state at t_0 from $(\mathit{elems}, \mathit{callbacks}, \mathit{seal})$ to $(\{e\} \cup \mathit{elems}, \mathit{callbacks}, \mathit{seal})$. Furthermore:

$$\exists t_1 \geq t_0,\ \forall t_2 > t_1, \mathbb{P}(t_2) = (\mathit{elems}_2, \mathit{callbacks}_2, \mathit{seal}_2)$$
$$\wedge \forall (f, \mathit{called}_2) \in \mathit{callbacks}_2, (f, \mathit{called}) \in \mathit{callbacks} \Rightarrow e \in \mathit{called}_2$$

Abstract pool operation $\mathit{seal}(s)$ changes the abstract state of the FlowPool at t_0 from $(\mathit{elems}, \mathit{callbacks}, \mathit{seal})$ to $(\mathit{elems}, \mathit{callbacks}, s)$, assuming that $\mathit{seal} \in \{-1\} \cup \{s\}$ and $s \in \mathbb{N}_0$, and $|\mathit{elems}| \leq s$.

Definition 2.7 [Consistency] A FlowPool state $\mathbb{S}$ is **consistent** with an abstract pool $\mathbb{P} = (\mathit{elems}, \mathit{callbacks}, \mathit{seal})$ at t_0 if and only if $\mathbb{S}$ is a valid state and:

$\forall e \in \mathit{Elem}, \mathit{hasElem}(\mathit{start}, e) \Leftrightarrow e \in \mathit{elems}$
$\forall f \in \mathit{Elem} => \mathit{Unit}, \mathit{hasCallback}(\mathit{start}, f) \Leftrightarrow f \in \mathit{callbacks}$
$\forall f \in \mathit{Elem} => \mathit{Unit}, \forall e \in \mathit{Elem}, \mathit{willBeCalled}(\mathit{start}, e, f) \Leftrightarrow \exists t_1 \geq t_0, \forall t_2 > t_1, \mathbb{P}(t_2) = (\mathit{elems}_2, (f, \mathit{called}_2) \cup \mathit{callbacks}_2, \mathit{seal}_2), \mathit{elems} \subseteq \mathit{called}_2$
$\forall s \in \mathbb{N}_0, \mathit{sealedAt}(\mathit{start}, s) \Leftrightarrow s = \mathit{seal}$

A FlowPool operation op is **consistent** with the corresponding abstract state operation op' if and only if $\mathbb{S}' = \mathit{op}(\mathbb{S})$ is consistent with an abstract state $\mathbb{A}' = \mathit{op}'(\mathbb{A})$.

A **consistency change** is a change from state $\mathbb{S}$ to state $\mathbb{S}'$ such that $\mathbb{S}$ is consistent with an abstract state $\mathbb{A}$ and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$, where $\mathbb{A} \neq \mathbb{A}'$.

Proposition 1. *Every valid state is consistent with some abstract pool.*

Theorem 2.8 [Safety] FlowPool operation `create` creates a new FlowPool consistent with the abstract pool $\mathbb{P} = (\emptyset, \emptyset, -1)$. FlowPool operations `foreach`, `append` and `seal` are consistent with the abstract pool semantics.

Lemma 2.9 [End of life] For all blocks $b \in \mathit{reachable}(\mathit{start})$, if value $v \in \mathit{Elem}$ is written to $\mathit{b.array}$ at some position idx at some time t_0, then $\forall t > t_0, \mathit{b.array}(\mathit{idx}) = v$.

Proof. The CAS in line 22 is the only CAS which writes an element. No other CAS has a value of type *Elem* as the expected value. This means that once the CAS in line 22 writes a value of type *Elem*, no other write can change it.

Corollary 1. *The end of life lemma implies that if all the values in b.array are of type Elem at* t_0*, then* $\forall t > t_0$ *there is no write to b.array.*

Lemma 2.10 [Valid hint] For all blocks $b \in reachable(start)$, if $b.index > 0$ at some time t_0, then $b.array(b.index - 1) \in Elem$ at time t_0.

Proof. Observe every write to $b.index$ – they are all unconditional. However, at every such write occurring at some time t_1 that writes some value idx we know that some previous value at $b.array$ entry $idx - 1$ at some time $t_0 < t_1$ was of type $Elem$. Hence, from Lemma 2.9 it follows that $\forall t \geq t_1, b.array(idx - 1) \in Elem$.

Corollary 2 (Compactness). *For all blocks $b \in reachable(start)$, if for some idx $b.array(idx) \in Elem$ at time t_0 then $b.array(idx - 1) \in Elem$ at time t_0. This follows directly from the Lemma 2.9 and Lemma 2.10, and the fact that the CAS in line 22 only writes to array entries idx for which it previously read the value from b.index.*

Definition 2.11 [Transition] If for a function $f(t)$ there exist times t_0 and t_1 such that $\forall t, t_0 < t < t_1, f(t) = v_0$ and $f(t_1) = v_1$, then we say that the function f goes through a **transition** at t_1. We denote this as:

$f : v_0 \overset{t_1}{\rightarrow} v_1$

Or, if we don't care about the exact time t_1, simply as:

$f : v_0 \rightarrow v_1$

Definition 2.12 [Monotonicity] A function of time $f(t)$ is said to be **monotonic**, if every value in its string of transitions occurs only once.

Lemma 2.13 [Freshness] For all blocks $b \in reachable(start)$, and for all $x \in b.array$, function x is monotonic.

Proof. CAS instruction in line 22 writes a value of type $Elem$. No CAS instruction has a value of type $Elem$ as the expected value, so this write occurs only once.

Trivial analysis of CAS instructions in lines 105 and 124, shows that their expected values are of type $Terminal$. Their new values are always freshly allocated.

The more difficult part is to show that CAS instruction in line 21 respects the statement of the lemma.

Since the CAS instructions in lines 105 and 124 are preceeded by a read of $idx = b.index$, from Lemma 2.10 it follows that $b.array(idx - 1)$ contains a value of type $Elem$. These are also the only CAS instructions which replace a $Terminal$ value with another $Terminal$ value. The new value is always unique, as shown above.

So the only potential CAS to write a non-fresh value to $idx + 1$ is the CAS in line 21.

A successful CAS in line 21 overwrites a value cb_0 at $idx + 1$ read in line 18 at t_0 with a new value cb_2 at time t_2. Value cb_2 was read in line 19 at t_1 from the entry idx. The string of transitions of values at idx is composed of unique values at least since t_1 (by Lemma 2.9), since there is a value of type $Elem$ at the index $idx - 1$.

The conclusion above ensures that the values read in line 19 to be subsequently used as new values for the CAS in line 21 form a monotonic function $f(t) = b.array(idx)$ at t.

Now assume that a thread T1 successfully overwrites cb_0 via CAS in line 21 at $idx + 1$ at time t_2 to a value cb_2 read from idx at t_1, and that another thread T2 is the **first** thread (since the FlowPool was created) to subsequently successfully complete the CAS in line 21 at $idx+1$ at time $t_{prev2} > t_2$ with some value cb_{prev2} which was at $idx + 1$ at some time $t < t_0$.

That would mean that $b.array(idx+1)$ does not change during $\langle t_0, t_2 \rangle$, since T2 was the first thread the write a non-fresh value to $idx + 1$, and any other write would cause the CAS in line 21 by T1 to fail.

Also, that would mean that the thread T2 read the value cb_{prev2} in line 19 at some time $t_{prev1} < t_1$ and successfully completed the CAS at time $t_{prev2} > t_2$. If the CAS was successful, then the read in line 18 by T2 occured at $t_{prev0} < t_{prev1} < t_1$. Since we assumed that T2 is the first thread to write a value cb_{prev2} to $idx+1$ at time t_{prev2} which was previously in $idx+1$ at some time $t < t_0$, then the CAS in line 21 at time t_{prev2} could not have succeeded, since its expected value is cb_{prev0} read at some time t_{prev0}, and we know that the value at $idx + 1$ was changed at least once in $\langle t_{prev0}, t_{prev2} \rangle$ because of the write of a fresh value by thread T1 at $t_2 \in \langle t_{prev0}, t_{prev2} \rangle$. This value is known to be fresh because $b.array(idx)$ is a monotonic function at least since t_{prev1}, and the read of the new value written by T1 occurred at $t_1 > t_{prev1}$. We also know that there is no other thread T3 to write the value cb_{prev0} during $\langle t_{prev0}, t_{prev2} \rangle$ back to $idx+1$, since we assumed that T2 is the first to write a non-fresh value at that position.

Hence, a contradiction shows that there is no thread T2 which is the **first** to write a non-fresh value via CAS in line 21 at $idx + 1$ for any idx, so there is no thread that writes a non-fresh value at all.

Lemma 2.14 [Lifecycle] For all blocks $b \in reachable(start)$, and for all $x \in b.array$, function x goes through and only through the prefix of the following transitions:
$null \to cb_1 \to \cdots \to cb_n \to elem$, where:
$cb_i \in Terminal, i \neq j \Rightarrow cb_i \neq cb_j, elem \in Elem$

Proof. First of all, it is obvious from the code that each block that becomes an element of $reachable(start)$ at some time t_0 has the value of all $x \in b.array$ set to $null$.

Next, we inspect all the CAS instructions that operate on entries of $b.array$.

The CAS in line 22 has a value $curo \in Terminal$ as an expected value and writes an $elem \in Elem$. This means that the only transition that this CAS can cause is of type $cb_i \in Terminal \to elem \in Elem$.

We will now prove that the CAS in line 21 at time t_2 is successful if and only if the entry at $idx + 1$ is $null$ or $nexto \in Terminal$. We know that the entry at $idx + 1$ does not change $\forall t, t_0 < t < t_2$, where t_0 is the read in line 18, because of Lemma 2.13 and the fact that CAS in line 21 is assumed to be successful. We know that during the read in line 19 at time t_1, such that $t_0 < t_1 < t_2$, the entry at idx was $curo \in Terminal$, by trivial analysis of the `check` procedure. It follows from corollary 2 that the array entry $idx + 1$ is not of type $Elem$ at time t_1, otherwise array entry idx would have to be of type $Elem$. Finally, we know that the entry at $idx + 1$ has the same value during the interval $\langle t_1, t_2 \rangle$, so its value is not $Elem$ at t_2.

The above reasoning shows that the CAS in line 21 always overwrites a one value of type $Terminal$ (or $null$) with another value of type $Terminal$. We have shown in Lemma 2.13 that it never overwrites the value cb_0 with a value cb_2 that was at $b.array(idx)$ at an earlier time.

Finally, note that the statement for CAS instructions in lines 105 and 124 also follows directly from the proof for Lemma 2.13.

Lemma 2.15 [Subsequence] Assume that for some block $b \in reachable(start)$ the transitions of $b.array(idx)$ are:

$$b.array(idx) : null \to cb_1 \to \cdots \to cb_n \overset{t_0}{\to} elem : Elem$$

Assume that the transitions of $b.array(idx + 1)$ up to time t_0 are:

$$b.array(idx + 1) : null \to cb'_1 \to \cdots \to cb'_m$$

The string of transitions $null \to cb'_1 \to \cdots \to cb'_m$ is a subsequence of $null \to cb_1 \to \cdots \to cb_n$.

Proof. Note that all the values written to $idx+1$ before t_0 by CAS in line 21 were previously read from idx in line 19. This means that the set of values occurring in $b.array(idx + 1)$ before t_0 is a subset of the set of values in $b.array(idx)$. We have to prove that it is actually a subsequence.

Assume that there exist two values cb_1 and cb_2 read by threads T1 and T2 in line 19 at times t_1 and $t_2 > t_1$, respectively. Assume that these values are written to $idx + 1$ by threads T1 and T2 in line 21 in the opposite order, that is at times t_{cas1} and $t_{cas2} < t_{cas1}$, respectively. That would mean that the CAS by thread T1 would have to fail, since its expected value cb_0 has changed between the time it was read in line 18 and the t_{cas1} at least once to a different value, and it could not have been changed back to cb_0 as we know from the Lemma 2.13.

Notice that we have actually prooved a stronger result above. We have also shown that the string of values written at $idx + 1$ by CAS in line 21 successfully is a subsequence of **all** the transitions of values at idx (not just until t_0).

Lemma 2.16 [Valid writes] Given a FlowPool in a valid state, all writes in all operations produce a FlowPool in a valid state.

Proof. A new FlowPool is trivially in a valid state.

Otherwise, assume that the FlowPool is in a valid state $\mathbb{S}$. In the rest of the proof, whenever some invariant is trivially unaffected by a write, we omit mentioning it. We start by noting that we already prooved the claim for atomic writes in lines 23, 55 and 86 (which only affect [INV5]) in Lemma 2.10. We proceed by analyzing each atomic CAS instruction.

CAS in line 63 at time t_1 maintains the invariant [INV1]. This is because its expected value is always *null*, which ensures that the lifecycle of *b.next* is $null \rightarrow b' : Block$, meaning that the function $reachable(start)$ returns a monotonically growing set. So if $current \in reachable(start)$ at t_0, then this also holds at $t_1 > t_0$. It also maintains [INV2] because the new value *nb* is always fresh, so $\forall b, b \notin following(b)$. Finally, it maintains [INV3] because it is preceeded with a bounds check and we know from corollary 2 and the Lemma 2.9 that all the values in $b.array(idx), idx < LASTELEMPOS$ must be of type *Elem*.

CAS in line 66 at time t_1 maintains the invariant [INV1], since the new value for the $current \neq null$ was read from *b.next* at $t_0 < t_1$ when the invariant was assumed to hold, and it is still there a t_1, as shown before.

For CAS instructions in lines 22, 124 and 105 that write to index *idx* we know from Lemma 2.10 that the value at $idx - 1$ is of type *Elem*. This immediately shows that CAS instructions in lines 124 and 105 maintain [INV3] and [INV4].

For CAS in line 22 we additionally know that it must have been preceeded by a successful CAS in line 21 which previously wrote a $Terminal$ value to $idx + 1$. From Lemma 2.14 we know that $idx + 1$ is still $Terminal$ when the CAS in line 22 occurs, hence [INV4] is kept.

Finally, CAS in line 21 succeeds only if the value at $idx + 1$ is of type $Terminal$, as shown before in Lemma 2.14. By the same lemma, the value at *idx* is either $Terminal$ or *Elem* at that point, since $idx - 1$ is known to be *Elem* by Lemma 2.10. This means that [INV4] is kept.

Lemma 2.17 [Housekeeping] Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, CAS instructions in lines 21, 63 and 66 do not change the abstract pool state $\mathbb{A}$.

Proof. Since none of the relations *hasElem*, *hasCallback*, *willBeCalled* and *sealedAt* are defined by the value of *current* CAS in line 66 does not change them, hence it does not change the abstract pool state.

No CAS changes the set of scheduled futures, nor is succeeded by a `future` construct so it does not affect the *willBeCalled* relation.

It is easy to see that the CAS in line 63 does not remove any elements, nor make any additional elements reachable, since the new block *nb* which becomes reachable does not contain any elements at that time. Hence the *hasElem* relation is not affected. It does change the value $last(start)$ to *nb*, but since $nb.array = t \cdot null^{BLOCKSIZE-1}$, where $t \in Terminal$ was previously the last non-null element in *b.array*, it does changes neither the *sealedAt* nor the *hasCallback* relation.

The CAS in line 21 does not make some new element reachable, hence the *hasElem* relation is preserved.

Note now that this CAS does not change the relations *hasCallback* and *sealedAt* as long as there is a value of type $Terminal$ at the preceeding entry *idx*. We claim that if the CAS succeeds at t_2, then either the value at *idx* is of type $Terminal$ (trivially) or the CAS did not change the value at $idx + 1$. In other words, if the value at *idx* at time t_2 is of type *Elem*, then the write by

CAS in line 21 does not change the value at $idx + 1$ at t_2. This was, in fact, already shown in the proof of Lemma 2.15.

The argument above proves directly that relations *hasCallback* and *sealedAt* are not changed by the CAS in line 21.

Lemma 2.18 [Append correctness] Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, a successful CAS in line 22 at some time t_0 changes the state of the FlowPool to $\mathbb{S}_0$ consistent with an abstract pool state $\mathbb{A}_0$, such that:

$\mathbb{A} = (elems, callbacks, seal)$

$\mathbb{A}_0 = (\{elem\} \cup elems, callbacks, seal)$

Furthermore, given a fair scheduler, there exists a time $t_1 > t_0$ at which the FlowPool is consistent with an abstract pool in state $\mathbb{A}_1$, such that:

$\mathbb{A}_1 = (elems_1, callbacks_1, seal_1)$, where:

$\forall (f, called_1) \in callbacks_1, (f, called) \in callbacks \Rightarrow elem \in called_1$

Proof. Assume that the CAS in line 22 succeeds at some time t_3, the CAS in line 21 succeeds at some time $t_2 < t_3$, the read in line 19 occurs at some time $t_1 < t_2$ and the read in line 19 occurs at some time $t_0 < t_1$.

It is easy to see from the invariants, *check* procedure and the corollary 1 that the CAS in line 22 can only occur if $b = last(start)$.

We claim that for the block $b \in reachable(start)$ such that $b = last(b)$ the following holds at t_2:

$b.array = elem^N \cdot cb_1 \cdot cb_2 \cdot null^{BLOCKSIZE-N-2}$

where $cb_1 = cb_2$, since there was no write to idx after cb_1, otherwise the CAS in line 22 at t_3 would not have been successful (by lemma Lemma 2.13).

Furthermore, $cb_1 = cb_2$ at t_3, as shown in the Lemma 2.15. Due to the same lemma, the entries of $b.array$ stay the same until t_3, otherwise the CAS in line 22 would not have been successful. After the successful CAS at t_3, we have:

$b.array = elem^N \cdot e \cdot cb_1 \cdot null^{BLOCKSIZE-N-2}$

where $e : Elem$ is the newly appended element– at t_3 the relation $hasElem(start, e)$ holds, and $sealedAt(start, s)$ and $hasCallback(start, f)$ did not change between t_2 and t_3.

It remains to be shown that $willBeCalled(start, e, f)$ holds at t_3. Given a fair scheduler, within a finite number of steps the future store will contain a request for an asynchronous computation that invokes f on e. The fair scheduler ensures that the future is scheduled within a finite number of steps.

Lemma 2.19 [Foreach correctness] Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, a successful CAS in line 124 at some time t_0 changes the state of the FlowPool to $\mathbb{S}_0$ consistent with an abstract pool state $\mathbb{A}_0$, such that:

$\mathbb{A} = (elems, callbacks, seal)$

$\mathbb{A}_0 = (elems, (f, \emptyset) \cup callbacks, seal)$

Furthermore, given a fair scheduler, there exists a time $t_1 \geq t_0$ at which the FlowPool is consistent with an abstract pool in state $\mathbb{A}_1$, such that:

$\mathbb{A}_1 = (elems_1, callbacks_1, seal_1)$, where:
$elems \subseteq elems_1$
$\forall (f, called_1) \in callbacks_1, elems \subseteq called_1$

Proof. From Lemma 2.13 and the assumption that the CAS is successful we know that the value at $b.array(idx)$ has not changed between the read in line 117 and the CAS in line 124. From Lemma 2.10 we know that the value at $idx - 1$ was of type $Elem$ since $b.index$ was read. This means that neither $hasElem(start, e)$ nor $sealedAt$ have changed after the CAS. Since after the CAS there is a $Terminal$ with an additional function f at idx, the $hasCallback(start, f)$ holds after the CAS. Finally, the $willBeCalled(start, e, f)$ holds for all elements e for which the $hasElem(e)$ holds, since the CAS has been preceeded by a call $f(e)$ in line 127 for each element. The Lemma 2.9 ensures that for each element f was called for stays in the pool indefinitely (i.e. is not removed).

Trivially, the time t_1 from the statement of the lemma is such that $t_1 = t_0$.

Lemma 2.20 [Seal correctness] Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, a successful CAS in line 105 at some time t_0 changes the state of the FlowPool to $\mathbb{S}_0$ consistent with an abstract pool state $\mathbb{A}_0$, such that:
$\mathbb{A} = (elems, callbacks, seal)$, where $seal \in \{-1\} \cup \{s\}$
$\mathbb{A}_0 = (elems, callbacks, \{s\})$

Proof. Similar to the proof of Lemma 2.19.

Definition 2.21 [Obstruction-freedom] Given a FlowPool in a valid state, an operation *op* is **obstruction-free** if and only if a thread T executing the operation *op* completes within a finite number of steps given that no other thread was executing the operation *op* since T started executing it.

We say that thread T executes the operation *op* **in isolation**.

Lemma 2.22 [Obstruction-free operations] All operations on FlowPools are obstruction-free.

Proof. By trivial sequential code analysis supported by the fact that the invariants (especially [INV2]) hold in a valid state.

Proof (Safety). From Lemma 2.17, Lemma 2.31, Lemma 2.19 and Lemma 2.20 directly, along with the fact that all operations executing in isolation complete after a finite number of steps by Lemma 2.22.

Definition 2.23 [Linearizability] We say that an operation *op* is linearizable if every thread observers that it completes at some time t_0 after it was invoked and before it finished executing.

Theorem 2.24 [Linearizable operations] FlowPool operations `append` and `seal` are linearizable.

Proof (Linearizable operations). This follows directly from statements about CAS instructions in Lemma 2.17, Lemma 2.31 and Lemma 2.20, along with the fact that a CAS instruction itself is linearizable.

Note that `foreach` starts by executing an asynchronous computation and then returns the control to the caller. This means that the linearization point may happen outside the execution interval of that procedure – so, `foreach` is not linearizable.

Definition 2.25 [Lock-freedom] In a scenario where some finite number of threads are executing a concurrent operation, that concurrent operation is *lock-free* if and only if that concurrent operation is completed after a finite number of steps by some thread.

Theorem 2.26 [Lock-freedom] FlowPool operations `append`, `seal`, and `foreach` are lock-free.

We begin by first proving that there are a finite number of execution steps before a consistency change occurs.

By Lemma 2.31, after invoking `append`, a consistency change occurs after a finite number of steps. Likewise, by Lemma 2.34, after invoking `seal`, a consistency change occurs after a finite number of steps. And finally, by Lemma 2.35, after invoking `foreach`, a consistency change likewise occurs after a finite number of steps.

By Lemma 2.36, this means a concurrent operation `append`, `seal`, or `foreach` will successfully complete. Therefore, by Definition 2.25, these operations are lock-free.

Note. For the sake of clarity in this section of the correctness proof, we assign the following aliases to the following CAS and WRITE instructions:

- $CAS_{append-out}$ corresponds to the outer CAS in `append`, on line 21.
- $CAS_{append-inn}$ corresponds to the inner CAS in `append`, on line 22.
- $CAS_{expand-nxt}$ corresponds to the CAS on *next* in `expand`, line 63.
- $CAS_{expand-curr}$ corresponds to the CAS on *current* in `expand`, line 66.
- CAS_{seal} corresponds to the CAS on the $Terminal$ in `tryWriteSeal`, line 105.
- $CAS_{foreach}$ corresponds to the CAS on the $Terminal$ in `asyncFor`, line 124.
- $WRITE_{app}$ corresponds to the WRITE on the new *index* in `append`, line 23.
- $WRITE_{adv}$ corresponds to the WRITE on the new *index* in `advance`, line 55.
- $WRITE_{seal}$ corresponds to the WRITE on the new *index* in `seal`, line 86.

Lemma 2.27 After invoking an operation *op*, if non-consistency changing CAS operations $CAS_{append-out}$, $CAS_{expand-nxt}$, or $CAS_{expand-curr}$, in the pseudocode fail, they must have already been successfully completed by another thread since *op* began.

Proof. Trivial inspection of the pseudocode reveals that since $CAS_{append-out}$ makes up a check that precedes $CAS_{append-inn}$, and since $CAS_{append-inn}$ is the only operation besides $CAS_{append-out}$ which can change the expected value of $CAS_{append-out}$, in the case of a failure of $CAS_{append-out}$, $CAS_{append-inn}$ (and thus $CAS_{append-out}$) must have already successfully completed or $CAS_{append-out}$ must have already successfully completed by a different thread since *op* began executing.

Likewise, by trivial inspection $CAS_{expand-nxt}$ is the only CAS which can update the *b.next* reference, therefore in the case of a failure, some other thread must have already successfully completed $CAS_{expand-nxt}$ since the beginning of *op*.

Like above, $CAS_{expand-curr}$ is the only CAS which can change the *current* reference, therefore in the case of a failure, some other thread must have already successfully completed $CAS_{expand-curr}$ since *op* began. □

Lemma 2.28 [Expand] Invoking the *expand* operation will execute a non- consistency changing instruction after a finite number of steps. Moreover, it is guaranteed that the *current* reference is updated to point to a subsequent block after a finite number of steps. Finally, *expand* will return after a finite number of steps

Proof. From inspection of the pseudocode, it is clear that the only point at which *expand*(b) can be invoked is under the condition that for some block b, $b.index > LASTELEMPOS$, where $LASTELEMPOS$ is the maximum size set aside for elements of type *Elem* in any block. Given this, we will proceed by showing that a new block will be created with all related references *b.next* and *current* correctly set.

There are two conditions under which a non-consistency changing CAS instruction will be carried out.

- **Case 1:** if $b.next = null$, a new block *nb* will be created and $CAS_{expand-nxt}$ will be executed. From Lemma 2.27, we know that $CAS_{expand-nxt}$ must complete successfully on some thread. Afterwards recursively calling *expand* on the original block b.
- **Case 2:** if $b.next \neq null$, $CAS_{expand-curr}$ will be executed. Lemma 2.27 guarantees that $CAS_{expand-curr}$ will update *current* to refer to *b.next*, which we will show can only be a new block. Likewise, Lemma 2.27 has shown that $CAS_{expand-nxt}$ is the only state changing instruction that can initiate a state change at location *b.next*, therefore, since $CAS_{expand-nxt}$ takes place within Case 1, Case 2 can only be reachable after Case 1 has been executed successfully. Given that Case 1 always creates a new block, therefore, *b.next* in this case, must always refer to a new block.

Therefore, since from Lemma 2.27 we know that both $CAS_{expand-nxt}$ and $CAS_{expand-curr}$ can only fail if already completed guaranteeing their finite completion, and since $CAS_{expand-nxt}$ and $CAS_{expand-curr}$ are the only state changing operations invoked through *expand*, the *expand* operation must complete in a finite number of steps.

Finally, since we saw in Case 2 that a new block is always created and related references are always correctly set, that is both *b.next* and *current* are correctly updated to refer to the new block, it follows that *numBlocks* strictly increases after some finite number of steps. □

Lemma 2.29 [$CAS_{append-inn}$] After invoking `append(elem)`, if $CAS_{append-inn}$ fails, then some thread has successfully completed $CAS_{append-inn}$ or CAS_{seal} (or likewise, $CAS_{foreach}$) after some finite number of steps.

Proof. First, we show that a thread attempting to complete $CAS_{append-inn}$ can't fail due to a different thread completing $CAS_{append-out}$ so long as `seal` has not been invoked after completing the read of *currobj*. We address this exception later on.

Since after *check*, the only condition under which $CAS_{append-out}$, and by extension, $CAS_{append-inn}$ can be executed is the situation where the current object *currobj* with index location *idx* is the $Terminal$ object, it follows that $CAS_{append-out}$ can only ever serve to duplicate this $Terminal$ object at location $idx+1$, leaving at most two $Terminal$s in block refered to by *current* momentarily until $CAS_{append-inn}$ can be executed. By Lemma 2.27, since $CAS_{append-out}$ is a non-consistency changing instruction, it follows that any thread holding any element $elem'$ can execute this instruction without changing the expected value of *currobj* in $CAS_{append-inn}$, as no new object is ever created and placed in location *idx*. Therefore, $CAS_{append-inn}$ cannot fail due to $CAS_{append-out}$, so long as `seal` has not been invoked by some other thread after the read of *currobj*.

This leaves only two scenarios in which consistency changing $CAS_{append-inn}$ can fail:

- **Case 1:** Another thread has already completed $CAS_{append-inn}$ with a different element $elem'$.
- **Case 2:** Another thread completes an invocation to the `seal` operation after the current thread completes the read of *currobj*. In this case, $CAS_{append-inn}$ can fail because CAS_{seal} (or, likewise $CAS_{foreach}$) might have completed before, in which case, it inserts a new $Terminal$ object *term* into location *idx* (in the case of a `seal` invocation, $term.sealed \in \mathbb{N}_0$, or in the case of a `foreach` invocation, $term.callbacks \in \{Elem \Rightarrow Unit\}$).

We omit the proof and detailed discussion of $CAS_{foreach}$ because it can be proven using the same steps as were taken for CAS_{seal}. □

Lemma 2.30 [Finite Steps Before State Change] All operations with the exception of `append`, `seal`, and `foreach` execute only a finite number of steps between each state changing instruction.

Proof. The `advance`, `check`, `totalElems`, `invokeCallbacks`, and `tryWriteSeal` operations have a finite number of execution steps, as they contain no recursive calls, loops, or other possibility to restart.

While the `expand` operation contains a recursive call following a CAS instruction, it was shown in Lemma 2.28 that an invocation of `expand` is guaranteed to execute a state changing instruction after a finite number of steps.

□

Lemma 2.31 [Append] After invoking `append(elem)`, a consistency changing instruction will be completed after a finite number of steps.

Proof. The `append` operation can be restarted in three cases. We show that in each case, it's guaranteed to either complete in a finite number of steps, or leads to a state changing instruction:

– **Case 1:** The call to `check`, a finite operation by Lemma 2.30, returns $false$, causing a call to `advance`, also a finite operation by Lemma 2.30, followed by a recursive call to `append` with the same element *elem* which in turn once again calls `check`.
  We show that after a finite number of steps, the `check` will evaluate to *true*, or some other thread will have completed a consistency changing operation since the initial invocation of `append`. In the case where `check` evaluates to *true*, Lemma 2.29 applies, as it guarantees that a consistency changing CAS is completed after a finite number of steps.
  When the call to the finite operation `check` returns $false$, if the subsequent `advance` finds that a $Terminal$ object is at the current block index *idx*, then the next invocation of `append` will evaluate `check` to *true*. Otherwise, it must be the case that another thread has moved the Terminal to a subsequent index since the initial invocation of append, which is only possible using a consistency changing instruction.
  Finally, if `advance` finds that the element at *idx* is an *Elem*, *b.index* will be incremented after a finite number of steps. By $INV1$, this can only happen a finite number of times until a $Terminal$ is found. In the case that `expand` is meanwhile invoked through `advance`, by Lemma 2.28 it's guaranteed to complete state changing instructions $CAS_{expand-nxt}$ or $CAS_{expand-curr}$ in a finite number of steps. Otherwise, some other thread has moved the $Terminal$ to a subsequent index. However, this latter case is only possible by successfully completing $CAS_{append-inn}$, a consistency changing instruction, after the initial invocation of append.

– **Case 2:** $CAS_{append-out}$ fails, which we know from Lemma 2.27means that it must've already been completed by another thread, guaranteeing that $CAS_{append-inn}$ will be attempted. If $CAS_{append-inn}$ fails, after a finite number of steps, a consistency changing instruction will be completed. If $CAS_{append-inn}$ succeeds, as a consistency changing instruction, consistency will have clearly been changed.

– **Case 3:** $CAS_{append-inn}$ fails, which, by Lemma 2.29, indicates that either some other thread has already completed $CAS_{append-inn}$ with another element, or another consistency changing instruction, CAS_{seal} or $CAS_{foreach}$ has successfully completed.

Therefore, `append` itself as well as all other operations reachable via an invocation of `append` are guaranteed to have a finite number of steps between *consistency* changing instructions. □

Lemma 2.32 [CAS_{seal}] After invoking `seal(size)`, if CAS_{seal} fails, then some thread has successfully completed CAS_{seal} or $CAS_{append-inn}$ after some finite number of steps.

Proof. Since by Lemma 2.29, we know that $CAS_{append-out}$ only duplicates an existing $Terminal$, it can not be the cause for a failing CAS_{seal}. This leaves only two cases in which CAS_{seal} can fail:

- **Case 1:** Another thread has already completed CAS_{seal}.
- **Case 2:** Another thread completes an invocation to the *append*(*elem*) operation after the current thread completes the read of $currobj$. In this case, CAS_{seal} can fail because $CAS_{append-inn}$ might have completed before, in which case, it inserts a new *Elem* object *elem* into location *idx*. □

Lemma 2.33 [$WRITE_{adv}$ and $WRITE_{seal}$] After updating *b.index* using $WRITE_{adv}$ or $WRITE_{seal}$, *b.index* is guaranteed to be incremented after a finite number of steps.

Proof. For some index, *idx*, both calls to $WRITE_{adv}$ and $WRITE_{seal}$ attempt to write $idx + 1$ to *b.index*. In both cases, it's possible that another thread could complete either $WRITE_{adv}$ or $WRITE_{seal}$, once again writing *idx* to *b.index* after the current thread has completed, in effect overwriting the current thread's write with $idx + 1$. By inspection of the pseudocode, both $WRITE_{adv}$ and $WRITE_{seal}$ will be repeated if *b.index* has not been incremented. However, since the number of threads operating on the FlowPool is finite, p, we are guaranteed that in the worst case, this scenario can repeat at most p times, before a write correctly updates *b.index* with $idx + 1$. □

Lemma 2.34 [Finite Steps Before Consistency Change] After invoking `seal(size)`, a consistency changing instruction will be completed after a finite number of steps, or the initial invocation of `seal(size)` completes.

Proof. The `seal` operation can be restarted in two scenarios.

- **Case 1:** The check $idx \leq LASTELEMPOS$ succeeds, indicating that we are at a valid location in the current block b, but the object at the current index location *idx* is of type *Elem*, not $Terminal$, causing a recursive call to `seal` with the same size *size*.
  In this case, we begin by showing that the atomic write of $idx + 1$ to *b.index*, required to iterate through the block b for the recursive call to `seal`, will be correctly incremented after a finite number of steps.
  Therefore, by both the guarantee that, in a finite number of steps, *b.index* will eventually be correctly incremented as we saw in Lemma 2.33, as well

as by $INV1$ we know that the original invocation of `seal` will correctly iterate through b until a $Terminal$ is found. Thus, we know that the call to `tryWriteSeal` will be invoked, and by both Lemma 2.30 and Lemma 2.31, we know that either `tryWriteSeal`, will successfully complete in a finite number of steps, in turn successfully completing `seal(size)`, or $CAS_{append-inn}$, another consistency changing operation will successfully complete.

- **Case 2:** The check $idx \leq LASTELEMPOS$ fails, indicating that we must move on to the next block, causing first a call to `expand` followed by a recursive call to `seal` with the same size *size*.
  We proceed by showing that after a finite number of steps, we must end up in Case 1, which we have just showed itself completes in a finite number of steps, or that a consistency change must've already occurred.
  By Lemma 2.28, we know that an invocation of `expand` returns after a finite number of steps, and *pool.current* is updated to point to a subsequent block.
  If we are in the recursive call to `seal`, and the $idx \leq LASTELEMPOS$ condition is $false$, trivally, a consistency changing operation must have occurred, as, the only way for the condition to evaluate to *true* is through a consistency changing operation, in the case that a block has been created during an invocation to `append`, for example.
  Otherwise, if we are in the recursive call to `seal`, and the $idx \leq LASTELEMPOS$ condition evaluates to *true*, we enter Case 1, which we just showed will successfully complete in a finite number of steps.

□

Lemma 2.35 [Foreach] After invoking `foreach(fun)`, a consistency changing instruction will be completed after a finite number of steps.

We omit the proof for `foreach` since it proceeds in the exactly the same way as does the proof for `seal` in Lemma 2.34.

Lemma 2.36 Assume some concurrent operation is started. If some thread completes a consistency changing CAS instruction, then some concurrent operation is guaranteed to be completed.

Proof. By trival inspection of the pseudocode, if $CAS_{append-inn}$ successfully completes on some thread, then that thread is guaranteed to complete the corresponding invocation of `append` in a finite number of steps.

Likewise by trivial inspection, if CAS_{seal} successfully completes on some thread, then by Lemma 2.30, `tryWriteSeal` is guaranteed to complete in a finite number of steps, and therefore, that thread is guaranteed to complete the corresponding invocation of `seal` in a finite number of steps.

The case for $CAS_{foreach}$ is omitted since it follows the same steps as for the case of CAS_{seal}

3 Proof of Determinism

3.1 Reduction rules

$$\frac{t = \texttt{create}\ p\ ;\ t' \quad p = (\emptyset, \emptyset, -1)}{t, T \mid P \ \longrightarrow\ t', T \mid p, P} \tag{Create}$$

$$\frac{t = p << v \ ; \ t' \quad p = (vs, cbs, -1) \quad p' = (\{v\} \cup vs, cbs, -1)}{t, T \mid p, P \longrightarrow t', T \mid p', P} \quad (\textsc{Append1})$$

$$\frac{\begin{array}{c} t = p << v \ ; \ t' \quad p = (vs, cbs, n) \quad |\{v\} \cup vs| \leq n \\ p' = (\{v\} \cup vs, cbs, n) \quad T' = \{f(v) \mid f \in cbs\} \end{array}}{t, T \mid p, P \longrightarrow t', T, T' \mid p', P} \quad (\textsc{Append2})$$

$$\frac{t = p \ \texttt{foreach} \ f \ ; \ t' \quad p = (vs, cbs, -1) \quad p' = (vs, \{f\} \cup cbs, -1)}{t, T \mid p, P \longrightarrow t', T \mid p', P} \quad (\textsc{Foreach1})$$

$$\frac{\begin{array}{c} t = p \ \texttt{foreach} \ f \ ; \ t' \quad p = (vs, cbs, n) \\ T' = \{g(v) \mid g \in \{f\} \cup cbs, v \in vs\} \quad p' = (vs, \{f\} \cup cbs, n) \end{array}}{t, T \mid p, P \longrightarrow t', T, T' \mid p', P} \quad (\textsc{Foreach2})$$

$$\frac{\begin{array}{c} t = p \ \texttt{seal} \ n \ ; \ t' \quad p = (vs, cbs, -1) \\ T' = \{g(v) \mid g \in cbs, v \in vs\} \quad p' = (vs, cbs, n) \end{array}}{t, T \mid p, P \longrightarrow t', T, T' \mid p', P} \quad (\textsc{Seal1})$$

$$\frac{t = p \ \texttt{seal} \ n \ ; \ t' \quad p = (vs, cbs, n)}{t, T \mid p, P \longrightarrow t', T \mid p, P} \quad (\textsc{Seal2})$$

3.2 Definitions

Definition 3.1 [Termination] A term t terminates with result P if its reduction ends in execution state $\{t : t = \{\epsilon\}\} \mid P$.

Definition 3.2 [Interleaving] Consider the reduction of a term t: $T_1 \mid P_1 \longrightarrow T_2 \mid P_2 \longrightarrow \ldots \longrightarrow \{t : t = \{\epsilon\}\} \mid P_n$. An *interleaving* is a reduction of t starting in execution state $T_1 \mid P_1$ in which reduction rules are applied in a different order.

Definition 3.3 [Valid Interleaving] An interleaving S', c, S'' of a reduction sequence S is *valid iff* for any reduction step c using rule (Create), if c creates pool p, p is not used in any reduction step in S'.

Definition 3.4 [Determinism] The reduction of a term t is *deterministic iff* either (a) t does not terminate for any valid interleaving, or (b) t always terminates with the same result for all valid interleavings.

3.3 Theorems and Lemmas

Theorem 3.5 [Determinism] The reduction of terms t is deterministic.

To prove this, we show that the reduction steps of every valid interleaving can be reordered in a way that does not change the termination, nor the result. The proof is based on the commutativity of transitions in the reduction.

Lemma 3.6 [Commutativity] Consider the reduction R of a term t and the two subsequent reduction steps based on reduction rules R_1 and R_2:

$$\ldots \longrightarrow T_n \mid P_n \xrightarrow{R_1} T_{n+1} \mid P_{n+1} \xrightarrow{R_2} T_{n+2} \mid P_{n+2} \longrightarrow \ldots$$

We list the reduction rules for which it is true that switching these two reduction steps yields the following reduction R':

$$\ldots \longrightarrow T_n \mid P_n \xrightarrow{R'_2} T'_{n+1} \mid P'_{n+1} \xrightarrow{R'_1} T_{n+2} \mid P_{n+2} \longrightarrow \ldots$$

where the prefix and suffix of the reduction R' are the same as in R.

Any pair of rules R_1 and R_2 applies above if the reduction steps operate on different pools. If they operate on the same pool, then for any two (not necessarily different) threads executing these reduction steps, the following table of rules applies.

R_1	R_2	R'_1	R'_2
Append1	Append1	Append1	Append1
Append1	Foreach1	Foreach1	Append1
Append1	Seal1	Seal1	Append2
Append2	Foreach2	Foreach2	Append2
Append2	Seal2	Seal2	Append2
Foreach1	Append1	Append1	Foreach1
Foreach1	Foreach1	Foreach1	Foreach1
Foreach1	Seal1	Seal1	Foreach2
Foreach2	Append2	Append2	Foreach2
Foreach2	Seal2	Seal2	Foreach2
Seal1	Append2	Append1	Seal1
Seal1	Foreach2	Foreach1	Seal1
Seal1	Seal2	Seal1	Seal2
Seal2	Append2	Append2	Seal2
Seal2	Foreach2	Foreach2	Seal2

Proof. By straightforward inspection of each pair of rules.

As an example, we pick the third row from the table. Consider the following execution schedule fragment:

$$\begin{aligned}
& p << v \ ; \ t', p \ \mathit{seal} \ n \ ; \ t'', T_k \mid p = (vs, cbs, -1), P_k \\
\xrightarrow{\textsc{Append1}} \ & t', p \ \mathit{seal} \ n \ ; \ t'', T_k \mid p = (\{v\} \cup vs, cbs, -1), P_k \\
\xrightarrow{\textsc{Seal1}} \ & \{cbs(w) : w \in \{v\} \cup vs\}, t', t'', T_k \mid p = (\{v\} \cup vs, cbs, n), P_k
\end{aligned}$$

If we reorder these two reduction steps, we get:

$$\begin{aligned}
&p << v\ ;\ t', p\ seal\ n\ ;\ t'', T_k \mid p = (vs, cbs, -1), P_k \\
&\overset{\textsc{Seal1}}{\longrightarrow}\ \{cbs(w) : w \in vs\}, p << v\ ;\ t', t'', T_k \mid p = (vs, cbs, n), P_k \\
&\overset{\textsc{Append2}}{\longrightarrow}\ cbs(v), \{cbs(w) : w \in vs\}, t', t'', T_k,\ \mid p = (\{v\} \cup vs, cbs, n), P_k
\end{aligned}$$

The final execution states in these two execution fragments are the same, hence the rest of the interleaving stays the same.

Note that if the two execution steps above had been executed by the same thread, the proof would proceed in the same manner.

Definition 3.7 [Canonical interleaving] Assume, without the loss of generality, that each Create rule assigns a unique identifier to each created pool. Assume any ordering on this set of identifiers.

We define the following ordering **comes-before** between threads as follows. There exists one **main thread** defined by the term t being reduced. The main thread comes before any other thread. For all other threads T_p and T_q let p and q be the terms with which the threads first appear in some execution state. Thread T_p comes before T_q *iff* p comes before q in the lexicographic ordering.

Note that in any interleaving we can identify the first reduction step in which an arbitrary thread T_p appears. We can recursively find the rest of its reduction steps. This means that we can assign the index p to every thread in the set of threads in every execution state later[3], and thus uniquely identify every thread in any execution state.

We define the **canonical order of the reduction steps of a thread** as follows:

1. First apply all the Create steps in the order of the pool identifiers.
2. Then apply all the Append steps in the order of the pool identifiers and some ordering of the elements being appended[4].
3. Then apply all the Foreach steps in the order of the pool identifiers and the lexicographic ordering on the callback term.
4. Finally, apply all the Seal steps in the order of the pool identifiers and the sizes of the seal.

Or, more formally:

[3] We claim that in our programming model this choice is unique, but even if this was not the case, we could pick either thread in the case of an ambiguity and do so without the loss of generality.

[4] Any countable set has some total ordering.

$$
\begin{aligned}
&T_0 \mid P_0 \xrightarrow{\textsc{Create}} T_0 \mid p_{i_1}, P_0 \xrightarrow{\textsc{Create}} \ldots \xrightarrow{\textsc{Create}} T_0 \mid p_{i_n}, p_{i_{n-1}}, \ldots, p_{i_1}, P_0 \\
&\xrightarrow{\textsc{Append}} T_1 \mid (\{v_{j_1}\} \cup vs_1, cbs_1, n_1)_{j_1}, P_{j_1} \xrightarrow{\textsc{Append}} \ldots \\
&\xrightarrow{\textsc{Append}} T_m \mid (\{v_{j_m}\} \cup vs_m, cbs_m, n_m)_{j_n}, P_{j_n} \\
&\xrightarrow{\textsc{Foreach}} T_{m+1} \mid (vs_{m+1}, \{t_{k_1}\} \cup cbs_{m+1}, n_{m+1})_{k_1}, P_{k_1} \xrightarrow{\textsc{Foreach}} \ldots \\
&\xrightarrow{\textsc{Foreach}} T_{m+f} \mid (vs_{m+f}, \{t_{k_f}\} \cup cbs_{m+f}, n_{m+f})_{k_f}, P_{k_f} \\
&\xrightarrow{\textsc{Seal}} T_{m+f+1} \mid (vs_{m+f+1}, cbs_{m+f+1}, n_{m+f+1})_{l_1}, P_{l_1} \xrightarrow{\textsc{Seal}} \ldots \\
&\xrightarrow{\textsc{Seal}} T_{m+f+s} \mid (vs_{m+f+s}, cbs_{m+f+s}, n_{m+f+s})_{l_s}, P_{l_s}
\end{aligned}
$$

where (with $\overset{a}{<}$ being the lexicographic ordering and $\dot{<}$ being some ordering on the set of elements):

$$
\begin{aligned}
&i_1 < i_2 < \ldots < i_n \\
&x < y \Rightarrow v_{j_x} \dot{<} v_{j_y} \lor (v_{j_x} = v_{j_y} \land j_x < j_y) \\
&x < y \Rightarrow t_{k_x} \overset{a}{<} t_{k_y} \lor (t_{k_x} = t_{k_y} \land k_x < k_y) \\
&x < y \Rightarrow n_x < n_y \lor (n_x = n_y \land l_x < l_y)
\end{aligned}
$$

We define the **canonical order of the reduction steps** as follows:

1. First apply all the reduction steps of the main thread t in the canonical order.
2. Identify all the threads created in the previous step. Take each of these threads in the above defined comes-before ordering and apply all their reduction steps in the canonical order.
3. Repeat the last step until there are no more applicable reduction steps.

Or, more formally:

$$
\begin{aligned}
&\{t\} \mid \emptyset \xrightarrow{S_C(\mathrm{t})} \{\epsilon, T_1, T_2, \ldots, T_n\} \mid P(\mathrm{t}) \\
&\xrightarrow{S_C(T_1)} \{\epsilon, \epsilon, T_2, \ldots, T_n, T_{1,1}, \ldots, T_{1,n_1}\} \mid P(T_1), P(\mathrm{t}) \\
&\xrightarrow{S_C(T_2)} \ldots \xrightarrow{S_C(T_n)} \\
&\xrightarrow{S_C(T_n)} \{\epsilon, \ldots, \epsilon, T_{1,1}, \ldots, T_{1,n_1}, T_{2,1}, \ldots, T_{n-1,n_{n-1}}, T_{n,1}, \ldots, T_{n,n_n}\} \\
&\mid P(T_1), \ldots, P(T_n), P(\mathrm{t}) \\
&\xrightarrow{S_C(T_{1,1})} \ldots \xrightarrow{S_C(T_{1,n})} \ldots \xrightarrow{S_C(T_{2,1})} \ldots \xrightarrow{S_C(T_{n-1,n_{n-1}})} \ldots \xrightarrow{S_C(T_{n,1})} \ldots
\end{aligned}
$$

where $S_C(T)$ is the sequence of reduction steps of thread T in the canonical order and t is the term being reduced, i.e. the main thread. For each T_p and T_q if $p \overset{a}{<} q$, then $T_p \overset{cb}{<} T_q$, where $\overset{cb}{<}$ is the comes-before ordering.

A **canonical interleaving** S_c is an interleaving of the reduction of a term t, such that the reduction steps are applied in the canonical order.

Proposition 2. *Every canonical interleaving S_c is a valid interleaving.*

Lemma 3.8 [Canonicity] Every valid interleaving of the reduction of a term t can have its execution steps reordered to the canonical interleaving S_c so that neither the termination nor the result is changed.

Proof. Consider an arbitrary valid interleaving S. We know that the starting execution state is $\{t\} \mid \emptyset$, where t is the term being reduced. Here, t is the main thread. As argued before, we can recursively identify all the reduction steps in the interleaving S in which the main thread executes.

We start by identifying all the Create steps of the main thread. Relying on the results from Lemma 3.6 we create a new interleaving in which the Create steps appear at the beginning in the canonical order. We know that the state after the last Create step in the original interleaving does not change by reordering the steps. Therefore, neither the termination nor the result are changed in the new interleaving. The new interleaving is trivially valid.

We then identify all the Append steps of the main thread. We know from our programming model that each of these steps can only refer to FlowPools from the Create steps we have already moved to the beginning of the interleaving – we cannot append to a pool created by some other subsequently created thread, due to scoping rules. For this reason we can reorder the Append steps of the main thread so that they appear after the Create steps and maintain a valid interleaving. Again, from Lemma 3.6 the new interleaving has the same termination and the result. We repeat the same for the Foreach and Seal steps.

At this point the new interleaving is the prefix of a canonical interleaving. From the commutativity rules we know that the main thread still creates the same set of threads as in the original interleaving. We order the newly created threads according to the comes-before ordering introduced earlier and identify each of these threads in the original interleaving. We recursively apply the same reordering of steps for the newly created threads, traversing them in the comes-before ordering.

We claim that every reduction in our programming model has a finite number of steps, so this procedure will be completed eventually, yielding a canonical interleaving.

More generally, if our programming model supported recursion, we could have infinite reductions. In that case, every time we produce a new interleaving in the process above, we extend the prefix of the interleaving which is the same as in the canonical interleaving. Furthermore, every time we produce a new interleaving in the process above, there is an execution state after the last reordered reduction step which is the same in both the new and the old interleaving, meaning that all the subsequent execution states are the same in both interleavings (by Lemma 3.6). This means that for any reduction of a term t we can produce a new interleaving with a prefix of an arbitrary length which corresponds to the prefix of the canonical interleaving, in the same time not changing the non-termination. Hence, every reduction of a term t does not terminate.

Proof (Determinism). Directly from the definition of determinism and Lemma 3.8.

4 Syntax and examples

This section describes the used syntax in more detail and presents a range of programming examples. The syntax is based on languages such as Groovy, Scala and Ruby. For conciseness and reasons of space, we omit the block braces and use indentation to denote block boundaries.

Methods. We use the `def` keyword to declare methods. After declaring the method name, the parameter list follows. Each parameter is given a name and a type behind a colon. Here is an example of declaring a `max` method which returns the greater of the two integers:

```
def max(a: Int, b: Int)
  if a > b a else b
```

Each method may either be standalone or defined within some object, in our case a FlowPool. If the method is defined within the object, it can refer to the object instance using the `this` keyword. It can additionally call the methods of the `this` object without prefixing their names with a `this` and a dot. Otherwise, invoking methods belonging to object instances must be prefixed with the object instance name and a dot, as in most object-oriented languages. Methods can be generic in their types and this is expressed with a list of type parameters in square brackets behind the method name. The following generic method just returns its parameter.

```
def id[T](x: T)
  x

id[Int](0)
id(0)
id(true)
id("Ok!")
```

Notice that above we did not have to put a type parameter value `T` when invoking the method – we assume that type parameters are infered from the types of regular method parameters.

Methods can also nest, as in the following example of a method which computes the sum of first `n` numbers:

```
def sum(n: Int)
  def subsum(i: Int)
    if i == n n
    else i + sum(i + 1)
  subsum(0)
```

Finally, methods can have multiple parameter lists, which provides a nice syntax for methods such as `aggregate`.

Values. Values are declared using the `val` keyword. Once declared, the value does not change. The following method creates an empty FlowPool:

```
def empty[T]
  val fp = new FlowPool[T]
  fp.builder.seal(0)
  fp
```

First-class functions. First-class functions can be simulated with anonymous classes in most object-oriented languages, but we provide special syntax for reasons of conciseness. The type of the function which takes parameters of type `T1` to `TN` and returns a value of type `R` is denoted as `(T1, T2, ..., TN) => R`. The function values are expressed by writing the name of the parameter, followed by a `=>` keyword and the method body.

Here is an example of a generic method which takes a value of type `T` and applies a custom function on it twice:

```
def twice[T](x: T)(f: T => T)
  f(f(x))
```

Since `twice` has multiple parameter lists, we can invoke with either of the following two notations:

```
twice(0)(x => x + 1)
twice(0) {
  x => x + 1
}
```

Additionally, we can omit the `x =>` prefix when defining the function value to make notation even more concise:

```
twice(0) {
  _ + 1
}
```

We can do this only if the parameter occurs in the method body only once. Each subsequent occurrence of a `_` keyword denotes a different parameter.

For-comprehensions. We define syntactic sugar for element traversal, best described through a couple of examples. The following for-loop which is supposed to print numbers from 0 until 10:

```
for (i <- 0 until 10) {
  println(i)
}
```

is desugared into the following expression:

```
(0.until(10)).foreach {
  i => println(i)
}
```

Above, the expression following the `<-` keyword must be an object containing the `foreach` method. This method must take a single parameter function value

– in this case the block that prints a given number. We assume that the object produced by the expression `0.until(10)` is predefined for integer values.

This mechanism allows us to use the same for-loop notation both for traversing numbers, installing callback handlers on future value or asynchronously traversing FlowPool elements.

In some cases, given a set of values being traversed, we want to produce a new set of values. The syntax for this involves the `yield` keyword, as in the following example:

```
val fp = new FlowPool[Int]()
fp.builder << 1
fp.builder << 2
fp.builder << 3

for (i <- fp) yield {
  i * i
}
```

The for-loop above is translated into a call to the `map` method on FlowPools:

```
fp.map {
  i => i * i
}
```

The `map` method must take a single argument function. The `map` method on FlowPools will return a new FlowPool with every element mapped.

A similar method called `flatMap` takes a single argument function which returns another traversable object. It can be used to compose traversals over several objects within one for-loop. The following for-loop which traverses two flow pools to produce a Cartesian product of their elements:

```
for (x <- fp1; y <- fp2) yield (x, y)
```

is translated into the following calls:

```
fp1.flatMap {
  x => fp2.map {
    y => (x, y)
  }
}
```

Futures. Futures are values which will become available at some point in the future. We distinguish between a value of type `Future[T]`, where `T` is the type of the value which becomes available (for example, an integer – `Int`) and an asynchronous computation which completes a future value. This asynchronous computation may be started using the `future` construct:

```
val p = new FlowPool[Int]
val f = future {
  p << 1
}
p.seal(1)
```

Above, the last line may be executed before or after appending the element `1` at runtime. Also, the `future` construct returns a future completed with the value its associated block computes, which is the last expression in the block.

Futures can be used in for-comprehensions, i.e. they define `foreach`, `map` and `flatMap` methods. A `foreach` loop on a future is an asynchronous computation which executes once the value of the future becomes available – it is a for-loop which traverses only a single value, and only once it becomes available. More complex patterns can also arise. Given two futures `f` and `g`, we can produce a third future which whose value becomes available only after values of `f` and `g` are available:

```
val f = future {
  sum(10)
}
val p = new FlowPool[Int]
val g = p.aggregate(0)(_ + _)(_ + _)
for (i <- 0 until 10) p.builder << i * i
p.seal(10)

val h: Future[Double] = for (x <- f; y <- g) yield {
  sqrt(y / 10 - x * x / 100)
}
```

Above, the real-valued future `h` contains the standard deviation of the first ten integers. The for-comprehension on futures is desugared into:

```
f.flatMap {
  x => g.map {
    y => sqrt(y / 10 - x * x / 100)
  }
}
```

Examples. We start off with a couple of methods that create new FlowPools. The `iterate` method iteratively applies a function to the starting value and puts it into a FlowPool. The `range` method creates a fixed size FlowPool from a range of numbers. The `fill` method creates a fixed size FlowPool filled with exactly the same element. Given the syntax primer above, it should be straightforward to grasp their contents.

```
def iterate[T]
  (s: T, f: T => T)
  val p = new FlowPool[T]
  val b = p.builder
  def recurse(x: T) {
    b << x
    recurse(f(x))
  }
  future { recurse(s) }
  p
```

```
def range
  (from: Int, end: Int)
  val p = new FlowPool[T]
  val b = p.builder
  future {
    for (i <- start to end)
      b << i
    b.seal(n)
  }
  p
```

```
def fill[T]
  (n: Int, elem: =>T)
  val p = new FlowPool[T]
  val b = p.builder
  future {
    for (i <- 1 to n)
      b << elem
    b.seal(n)
  }
  p
```

We now show that `aggregate` can be used to implement a range of other methods based on reducing the values. Notice that the `fold` we define is less general than the `aggregate`, since it places a type bound on the type parameter, meaning it can only operate on supertypes of the values in the pool[5].

[5] We have not precisely defined what type bounds and inheritance are, but it suffices to say that in a language with subtyping an unrestricted fold cannot be expressed in terms of aggregate we have defined.

```
def exists
  (pred: T => Boolean)
  aggregate(false)(_ ∨ _) {
    (acc, x) =>
    acc ∨ pred(x)
  }
```

```
def forall
  (pred: T => Boolean)
  aggregate(true)(_ ∧ _) {
    (acc, x) =>
    acc ∧ pred(x)
  }
```

```
def min()
  def min(a: Int, b: Int)
    if a < b a else b
  aggregate(0)(min) {
    min
  }
```

```
def sum()
  aggregate(0)(_ + _) {
    _ + _
  }
```

```
def product()
  aggregate(1)(_ * _) {
    _ * _
  }
```

```
def count
  (pred: T => Boolean)
  aggregate(0)(_ + _) {
    (acc, x) =>
    if pred(x) 1 else 0
  }
```

```
def fold[U >: T]
  (zero: U, op: (U, U) => U)
  aggregate(zero)(op)(op)
```

```
def flatten[S]
  (q: FlowPool[FlowPool[S]])
  val p = new FlowPool[S]
  val b = p.builder
  aggregate(future(0))(add) {
    (af, r) =>
    val sf = for (x <- q) b << x
    add(af, sf)
  } map { sz => b.seal(sz) }
  p
```

The `flatten` method turns a FlowPool whose elements are FlowPools themselves into a FlowPool containing all the elements of the nested FlowPools. It could also have been expressed in terms of the `flatMap` method described earlier.

Sometimes there is a need to convert between futures and FlowPools. Lets assume we have a list of futures and we wish to have a FlowPool of their values instead. The method `toFlowPool` does this by calling a `foreach` on every future. We assume the existence of a traversable `List` datatype.

The converse is not so easy, since we do not know how many elements will there be available in the FlowPool. This means we cannot create a list of futures, because we do not know how many future there will be. We can instead create a future which holds the final list of values. This is what the method `toList` does. Notice that the aggregate invocation does not form a commutative monoid, so the results will be nondeterministic, which is reflected in the order of elements in the resulting list. We could still maintain determinism given that we use a `Set` datatype instead of a `List`.

```
def toFlowPool[T]
  (fs: List[Future[T]])
  val p = new FlowPool[T]
  val b = p.builder
  for (f <- fs; x <- f)
    b << x
  p
```

```
def toList[T]
  (p: FlowPool[T])
  p.aggregate(∅)(_ ::: _) {
    (acc, x) => x :: acc
  }
```

```
def toList2[T]
  (p: FlowPool[T], n: Int)
  type LFT = List[Future[T]]
  val fs = List.fill(n)
    (new Future[T]())
  def complete(l: LFT, x: T)
    if ¬l.head.tryComplete(x)
      complete(l.tail, x)
    else l
  p.builder.seal(n)
  p.aggregate(fs) {
    (gs, hs) => gs
  } {
    (acc, x) =>
    complete(acc, x)
  }
  fs
```

```
def intersect
  (that: FlowPool[T])
  val p = new FlowPool[T]
  val b = p.builder
  for (x <- this) {
    for (y <- that) {
      if (x = y) b << x
    }
  }
  p
```

However, given that we assume the number of elements in the FlowPool, we can do better than that. The method `toList2` starts by sealing the pool to the expected number of elements. If successful, the predefined futures are completed as the elements arrive into the FlowPool. We have to use the `tryComplete` method on futures, which completes the future only given that it has not already been completed. This potentially yields nondeterministic computations, but we cannot avoid this without more expressive abstractions, such as pools for which we know that all of the elements they hold are the same and some sort of a `forany` construct which invokes a callback on any element of the pool, roughly speaking.

The `intersect` method above produces a new FlowPool with elements that appear both in the current FlowPool `this` and another FlowPool `that`. It is not very efficient, however – a more efficient implementation would require a more expressive single-assignment abstraction such as a single-assignment map.

5 Additional Evaluation

In this section, some additional evaluation results are presented.

Map and Reduce The *Reduce* benchmark starts P threads which concurrently insert a total of N elements. The `aggregate` operation is used to reduce the set of values inserted into the pool. Note that in the FlowPool implementation there may be as many threads computing the aggregation as there are different lanes – elements from different lanes are batched together once the pool is sealed.

The *Map* benchmark is similar to the *Reduce* benchmark, but instead of reducing a value, each element is mapped into a new one and added to a second pool.

Scaling in Input Size In figure 4 we can see that the Input, Map, Reduce and Histrogram benchmark all scale linearly in the input size with any parallelism level. The Comm benchmark has not been tested for different sizes.

Multi-Lane Scaling By default, the number of lanes is set to the parallelism level P, corresponding to the number of used CPUs. However, since the implementation has to use hashing on the thread IDs instead of the real CPU index, we tested whether varying the number of lanes to $1.5P$, $2P$, $3P$ and $4P$ results in performance gain due to fewer collisions. Benchmarks have shown (see fig 3) that this yields no observable gain – in fact, this sometimes even decreased performance slightly.

Performance Gain As stated in the abstract, FlowPools – or more precisely multi-lane FlowPools – may reduce execution time by $49 - 54\%$ on 4-core i7. These figures have been obtained by comparing medians of execution times for insertions between multi-lane FlowPools and concurrent linked queues (which were always faster than linked transfer queues), where each structure was evaluated on its optimal parallelization level. The resulting data is shown in table 1.

Methodology All the presented configurations have been measured 20 times, where the 5 first values have been discarded to let the JIT stabilize. Aggregated values are always medians. The benchmarks have been written using `scala.testing.Benchmark` and executed through SBT[6] using the following flags

[6] Simple Build Tool

Architecture	Elements	FlowPool $t[ms]$	P	Queue $t[ms]$	P	decr.
4-core i7	2M	17	8	35	1	51%
4-core i7	5M	44	8	87	1	49%
4-core i7	15M	118	8	258	1	54%
UltraSPARC T2	1M	23	32	111	4	79%
UltraSPARC T2	2M	34	64	224	4	84%
UltraSPARC T2	5M	62	64	556	4	88%
UltraSPARC T2	15M	129	64	1661	4	92%
32-core Xeon	2M	30	8	50	1	40%
32-core Xeon	5M	46	64	120	1	61%
32-core Xeon	15M	126	64	347	1	63%

Table 1. Execution times for insert benchmark for multi-lane FlowPool and concurrent linked queues, including execution time decrease percentage.

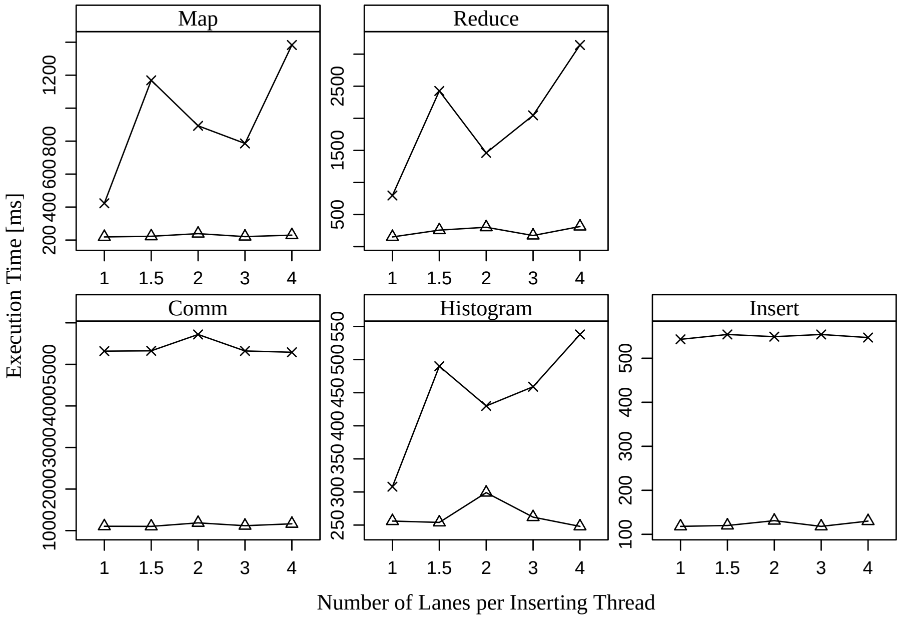

Fig. 3. Execution times vs lane-factor for multi-lane FlowPools. × UltraSPARC T2, △ 4-core i7

for the JavaVM: `-Xmx2048m -Xms2048m -XX:+UseCondCardMark -verbose:gc -XX:+PrintGCDetails -server`.

References

1. *FlowPools: a lock-free deterministic concurrent dataflow abstraction*, Tokyo, Japan, Sept. 2012.

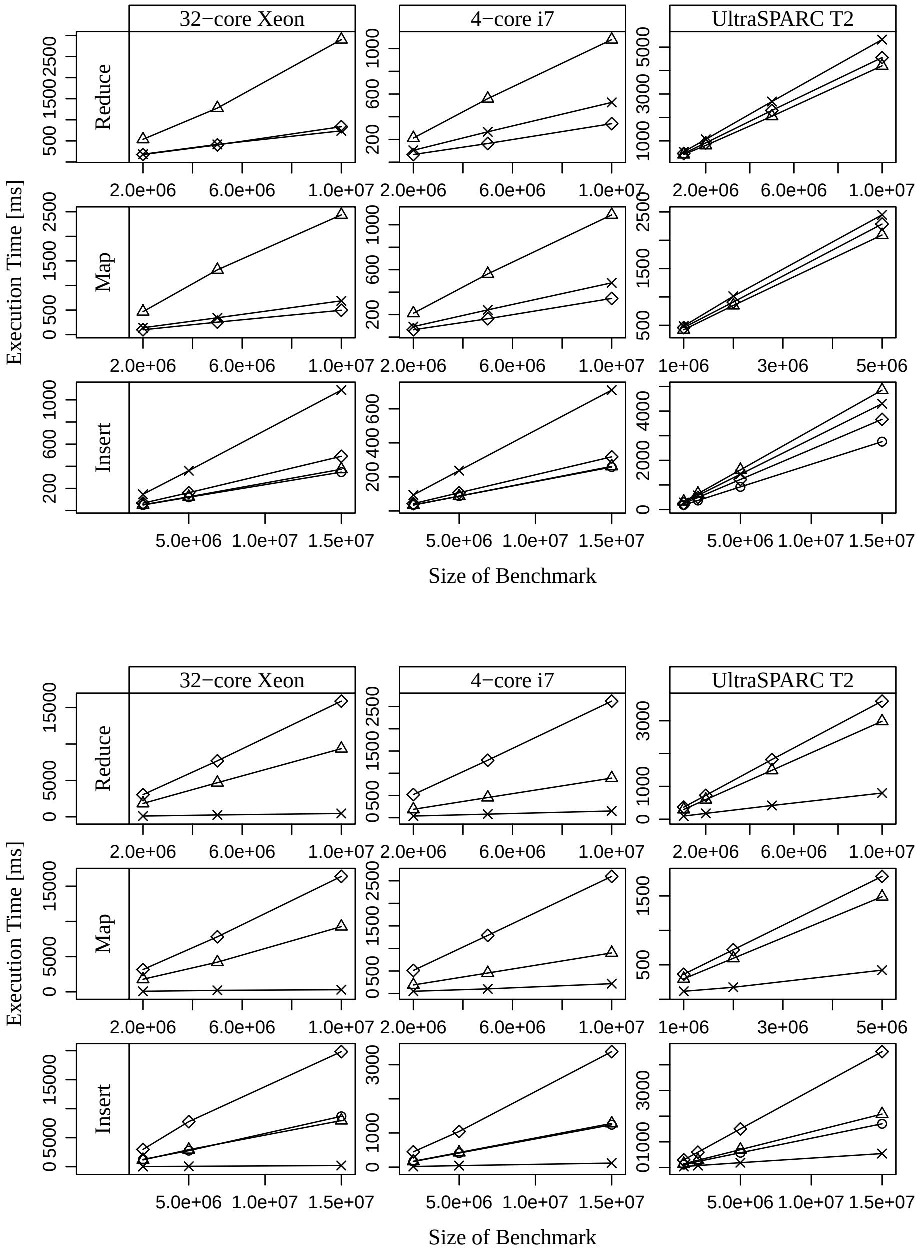

Fig. 4. Execution time vs benchmark size ($P = 1, 8$). $\Diamond$ single-lane FlowPool, $\times$ multi-lane FlowPool, $\triangle$ linked transfer queue, $\circ$ concurrent linked queue