Concurrent Datatype Verification

Verifying lock free data types using CSP and FDR

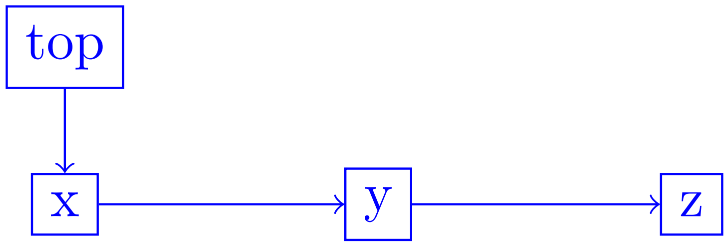
Jonathan Lawrence – jonathan.lawrence@cs.ox.ac.uk

Introduction

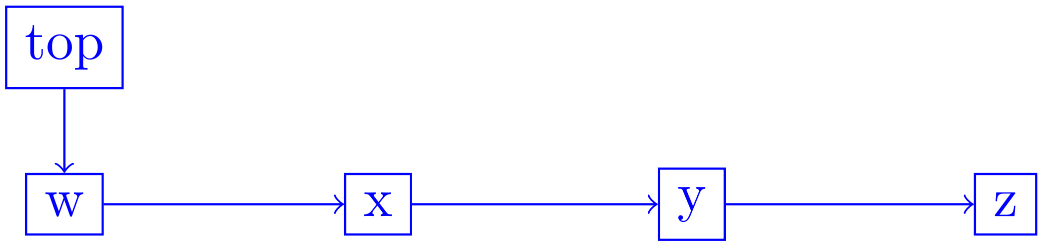
- Case study using CSP [2] model.
  - List-based stack [9].
- FDR (Failures-Divergences Refinement tool [8, 7]) for verification .
- Implementation requires unbounded stamps for correctness.
- Combines model-checking with state-based refinement.
- Lock freedom is also verified.

Case study: list based stack

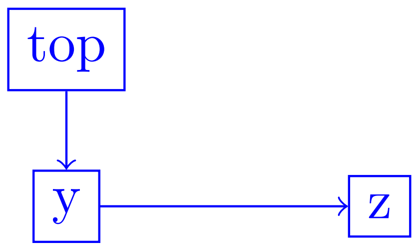
- Encapsulates a bounded total stack.
- Has operations push(value:T):Boolean and pop:Option[T].



after push(w):true :



or, after pop:Some(x) :



Specification (in CSP)

Provides an abstract model of the behaviour of the datatype.

```
datatype Value = A | B                     -- Type of data to be stored
datatype Option = None | Some.Value        -- Analog of Scala Option[Value]

channel push : Value.Bool
channel pop  : Option

Stack(stack) =                          -- stack is a sequence of Value

((#stack < capacity) & push?x!True -> Stack(<x>^stack) )
[]
((#stack + nthreads > capacity) & push?x!False -> Stack(stack))
[]
(if (#stack == 0) then pop!None -> Stack(stack)
 else pop!Some.head(stack) -> Stack(tail(stack)) )
```

Implementation

Scala code for push(value:T):Boolean and pop:Option[T]

```
def push(value: T): Boolean = {
  val node = allocate
  if (node == null) return false
  node.value = value
  while (true) {
    val top = t.get
    node.next = top
    val done = t.compareAndSet(top, node)
    if (done) return true
  }
  false  // Unreachable
}
```

Implementation...

```
def pop: Option[T] = {
  while (true) {
    val top = t.get
    if (top == null) return None
    else {
      val next = top.next
      val done = t.compareAndSet(top, next)
      if (done) {
        val value = top.value
        free(top)
        return Some(value)
      }
    }
  }
  None // Unreachable
}
```

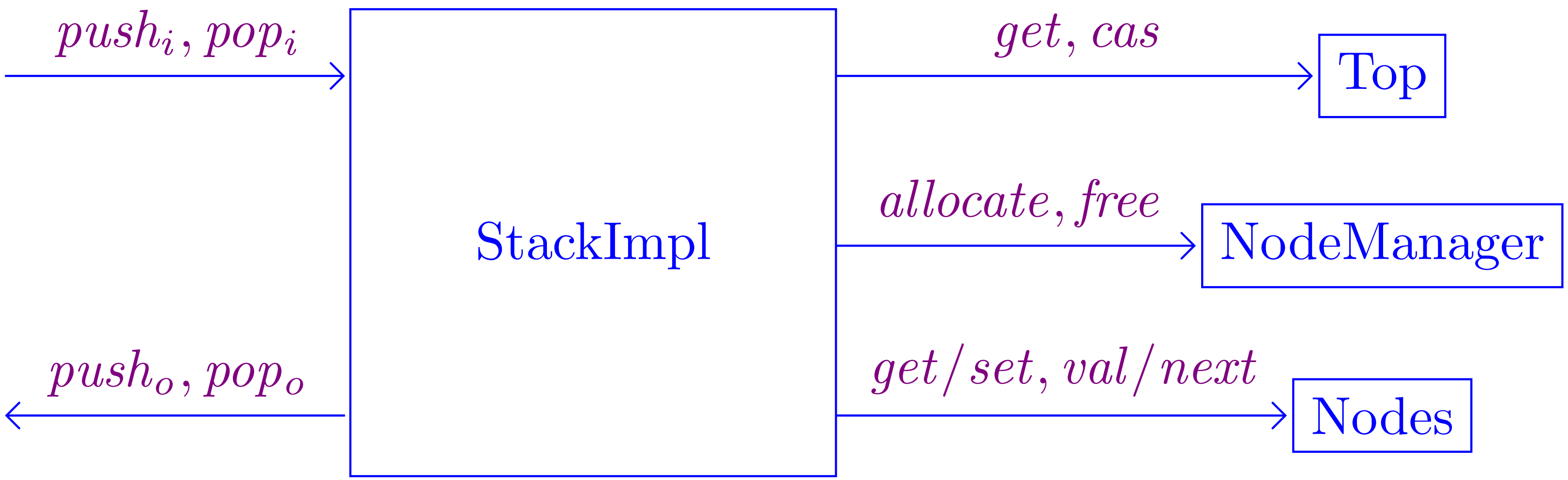
CSP implementation model

```
StackImpl = (push_i?value -> allocate?node ->
             if node==Null then push_o!False -> StackImpl
             else setval!node.value -> PushLoop(node) )
          [] pop_i -> PopLoop

PushLoop(node) = get?top -> setnext!node.top ->
                 cas!top.node?done ->
                 if (done) then push_o!True -> StackImpl
                 else PushLoop(node)

PopLoop = get?top ->
          if top==Null then pop_o!None -> StackImpl
          else getnext!top?next -> cas!top.next?done ->
            if (done) then getval!top?value -> free!top ->
              pop_o!Some.value -> StackImpl
            else PopLoop
```

CSP implementation structure



- `StackImpl` is replicated per thread.
- $push_i, pop_i, push_o, pop_o$ are labelled with thread ID.
- `Top`, `NodeManager` and `Nodes` are shared components.
- All operations on shared components are logically atomic.

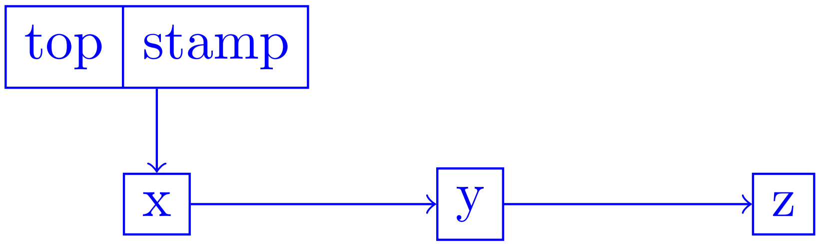
The ABA problem

- This naive implementation is actually incorrect.
- The so-called "ABA problem" [3, 1, pages 233–237] commonly arises in concurrent datatypes which use compare and set.
- It can occur when a thread has read the old value in a location and is then suspended, prior to performing a CAS operation.
- Other thread(s) then perform actions on the datastructure which return the location to the same value, such that performing the (successful) CAS now has an incorrect effect.

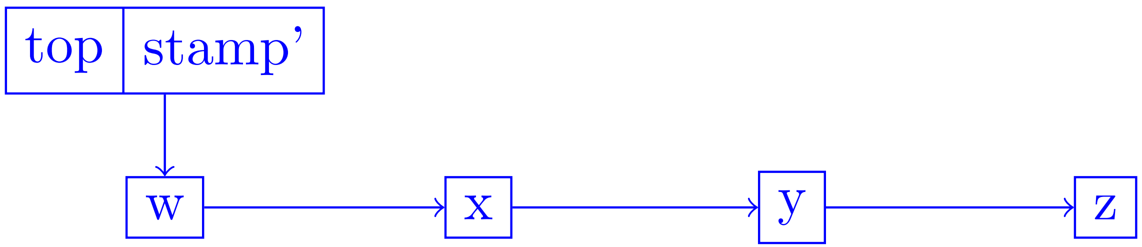
FDR Demo

Stamped references

- A solution to the ABA problem uses stamped values.
- A single-use value (stamp) is associated with each location prone to ABA updates.
- The stamp is updated to a fresh value each time the location is modified.



after push(w):true :



Stamp reuse

- FDR can only check finite state systems.
- This is a problem if we require a fresh stamp for every update.
- Solutions:
  1. Allow an error if a stamp is reused (realistic).
  2. Model an unbounded set of stamps with a finite set.
- In both cases a stamp can validly be reused if it is not currently held by another thread, to be used in a CAS.
- The trouble with (1) is that it *might* conceal other errors.

StampManager

- Represents an "angelic" watchdog process.
- Z[6, 10] presentation is easier to understand:

$inc?stamp$ →

$issue?stamp!newstamp?swap$ →

$$\begin{array}{|l|}\hline StampManager \\ usecount : Stamp \longrightarrow \mathbb{N} \\ current : Stamp \\ total\ usecount \leq MAXCOUNT \\ \hline \end{array}$$

- $[Stamp]$ is a finite, data-independent set.
- Initially, no stamps are in use:

$$InitStampManager \mathrel{\widehat{=}} [StampManager \mid usecount = Stamp \times \{0\}]$$

StampManager operations

inc

$$\Delta StampManager; stamp? : Stamp$$

$$stamp? = current \land total\ usecount < MAXCOUNT$$
$$usecount' = usecount \oplus \{stamp? \mapsto usecount(stamp?) + 1\}$$
$$current' = current$$

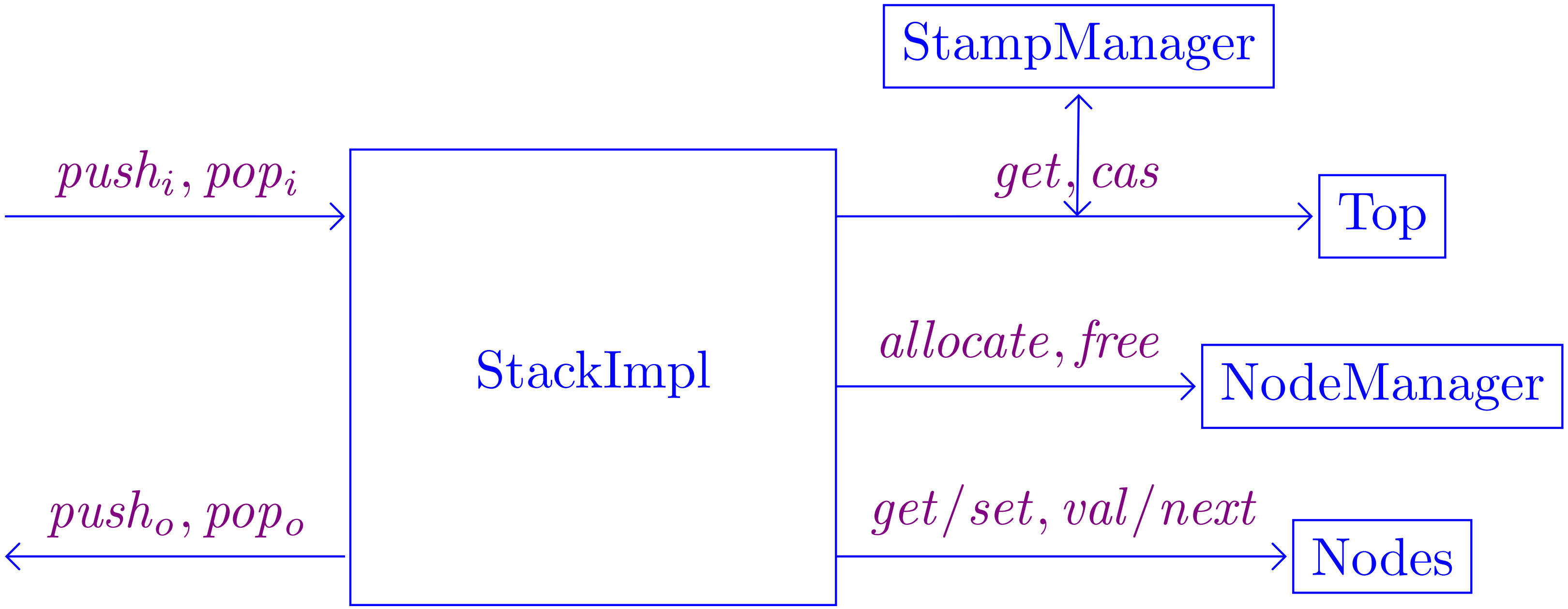
issue

$$\Delta StampManager$$
$$stamp?, newstamp! : \mathbb{N}; swap? : Boolean$$

$$usecount\ stamp? > 0 \land (swap? = True \Rightarrow stamp? = current)$$
$$usecount' = usecount \oplus \{stamp? \mapsto usecount(stamp?) - 1\}$$
$$stamp? = current \Rightarrow usecount'\ newstamp! = 0$$
$$current' = \{True \mapsto newstamp!, False \mapsto current\} swap?$$

Enhanced implementation

- *StackImpl* is replicated per thread.
- *StampManager* observes and controls stamp values.



- FDR verification checks now succeed.
- However – relies on “angelic” *StampManager*.

Data-independence

Often, there is a *threshold* size N_T for a type T such that:

$\forall\, T, T' \bullet |T|, |T'| \geq N_T \Rightarrow P(T) \sqsubseteq Q(T) \Leftrightarrow P(T') \sqsubseteq Q(T')$
for CSP processes P, Q both data-independent w.r.t. T [4, §15.2.2].

In our case, *Stamp* is data-independent and $N_{Stamp} = nthreads$.

FDR verification for $|Stamp| = nthreads$ therefore implies correctness for any $|Stamp| \geq nthreads$ and in particular for $Stamp = \mathbb{N}$.

$Stack(Stamp) \sqsubseteq_{FD} StackImpl(Stamp) \Leftrightarrow Stack(\mathbb{N}) \sqsubseteq_{FD} StackImpl(\mathbb{N})$

NB Data type $Value(A|B)$ is also data-independent with $N_{Value} = 2$.

StampManager refinement

- There is a stateless data refinement of *StampManager* for $Stamp = \mathbb{N}$.
- The retrieve relation is simply an additional invariant on the abstract state:

$$StampManagerRetr \mathrel{\widehat{=}} [StampManager \mid \forall n : \mathbb{N} \bullet n > current \Rightarrow usecount\ n = 0]$$

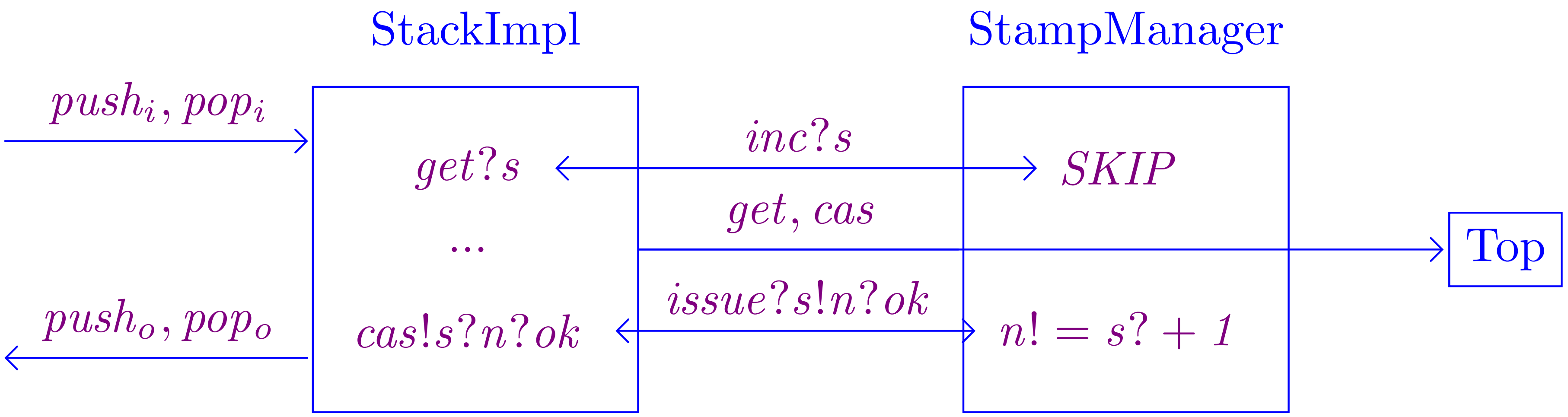
No *Stamp* value above *current* is used.

- The *inc* operation is refined by *SKIP*.
- The *issue* operation returns its input, incremented by 1:

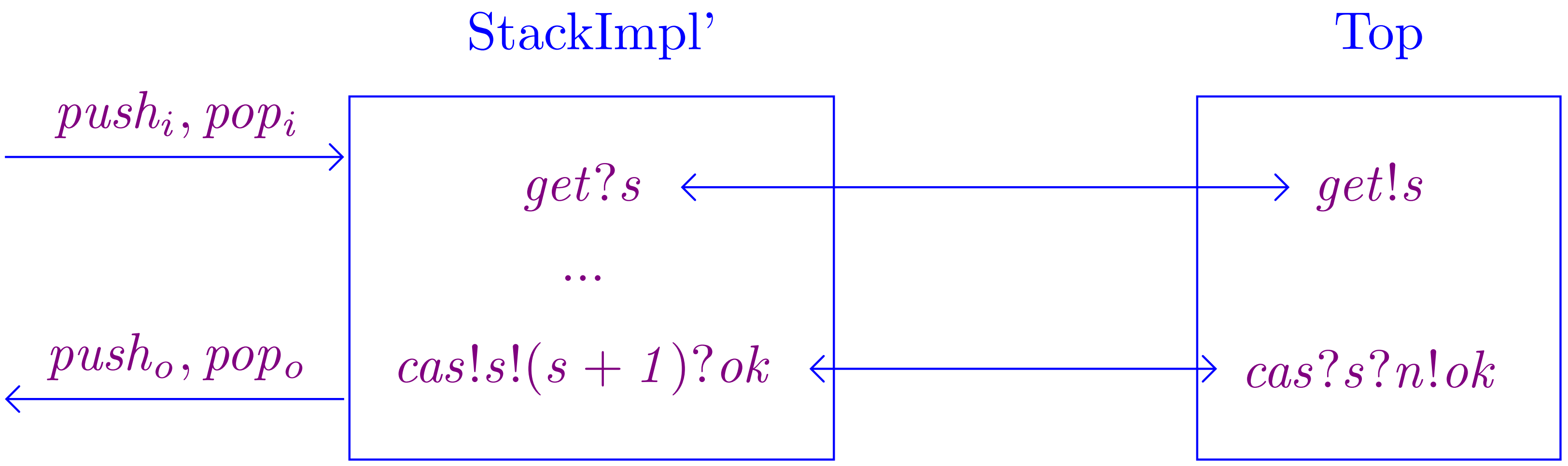
$$issueImpl \mathrel{\widehat{=}} [stamp?, newstamp! : \mathbb{N} \mid newstamp! = stamp? + 1]$$

- Correctness proofs are not included here.

StampManager elimination



Merge *StackImpl* with *StampManager* to become *StackImpl'*:



Final implementation (of pop())

```
def pop: Option[T] = {
  while (true) {
    val (top,∗stamp∗) = t.get  <<<
    if (top == null) return None
    else {
      val next = top.next
      val done = t.compareAndSet((top,∗stamp∗), (next,∗stamp+1∗}))  <<<
      if (done) {
        val value = top.value
        free(top)
        return Some(value)
      }
    }
  }
  None  // Unreachable
}
```

Verified correct for *nthreads* – assuming infinite stamps are available.

Summary

1. **Translate:** Convert from the imperative program to CSP model.
2. **Finitize:** Model the *Stamp* type as a finite set, introducing an angelic *StampManager* watchdog process to control reuse.
3. **Verify:** Perform necessary FDR verification check(s).
4. **Promote:** Apply data-independence to replace the finite *Stamp* type with an infinite one ($\mathbb{N}$). Preserves verification results.
5. **Refine:** Use state-based data refinement to reduce *StampManager* to a stateless implementation.
6. **Eliminate:** Merge the refined *StampManager* into the threads' behaviour and remove it from the model.

Results and conclusion

- 3 datatypes using stamps have been modelled and verified using this approach:
  1. List based stack (this example)
  2. List based queue
  3. Array based stack
- Similar modelling techniques, with subtle differences.
- One error found in publication [1, Fig 10.16] (queue).
- More automation needed – particularly translations.

Acknowledgements to...

- my supervisors Gavin Lowe and Bill Roscoe for advice and encouragement.
- the FDR refinement checker for CSP [8, 7].
- the Fuzz typechecker for Z [5].

References

[1] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised First Edition. Morgan Kaufmann (2012)

[2] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)

[3] IBM Corporation: Z/Architecture Principles of Operation. IBM Knowledge Center (1975)

[4] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1997)

[5] Spivey, J.M.: Fuzz typechecker for Z - Spivey's Corner. https://spivey.oriel.ox.ac.uk/corner/Fuzz_typechecker_for_Z

[6] Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, 2nd revised edn. (1992)

[7] Thomas Gibson-Robinson: FDR4 - The CSP Refinement Checker. https://www.cs.ox.ac.uk/projects/fdr/

[8] Thomas Gibson-Robinson, Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 187–201. Springer (2014)

[9] Treiber, R.K.: Systems Programming: Coping with Parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center (1986)

[10] Woodcock, J., Davies, J.: Using Z : Specification, Refinement, and Proof. Prentice Hall (1996)

FDR verification checks

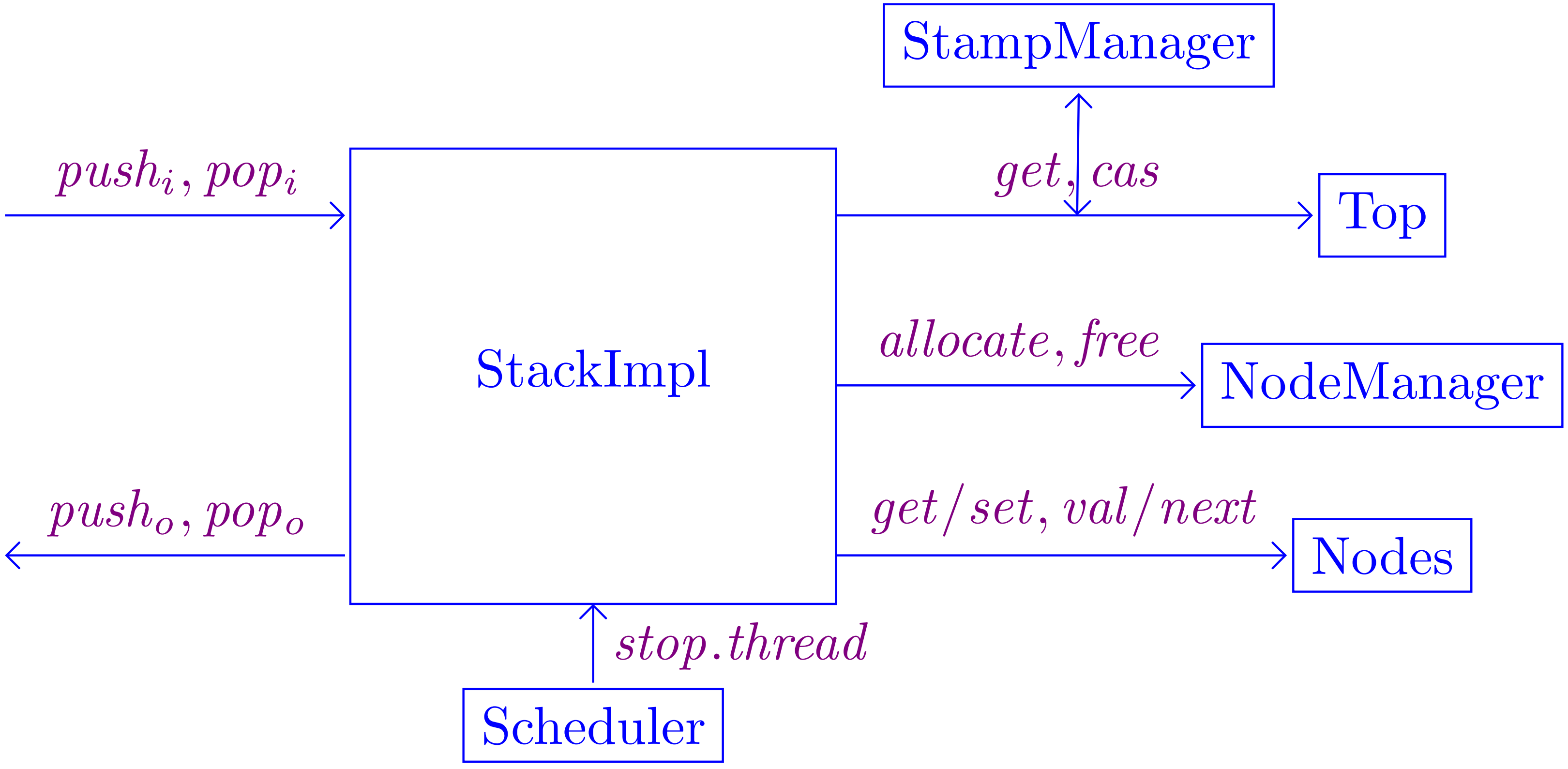
Checks required to validate a lock-free concurrent datatype:

1. **Conformance:** For any thread, the abstract operation is consistent with the interface events.
2. **Linearizability:** All possible executions are consistent with the abstract specification.
3. **Lock freedom:** The system is deadlock free irrespective of arbitrary thread suspensions.
4. **Divergence freedom:** If the specification is livelock free, so is the implementation.

These checks can be performed as a single FDR refinement check, or for efficiency, split into checks for each aspect separately.

Enhanced implementation structure

- StackImpl is replicated per thread.
- StampManager observes and controls stamp values.
- Scheduler may permanently suspend all except one thread.



Refinement theorem for *incImpl*

$$\forall\, StampManager;\ stamp? : \mathbb{N} \bullet$$
$$\quad \mathrm{pre}\ inc \land StampManagerRetr \land incImpl \Rightarrow$$
$$\quad (\exists\, StampManager' \bullet inc \land StampManagerRetr')$$

$$\forall\, usecount : \mathbb{N} \longrightarrow \mathbb{N};\ current : \mathbb{N};\ stamp? : \mathbb{N} \mid$$
$$\quad total\ usecount \leq MAXCOUNT \bullet$$
$$\quad stamp? = current \land total\ usecount < MAXCOUNT \land$$
$$\quad (\forall\, n : \mathbb{N} \bullet n > current \Rightarrow usecount\ n = 0) \Rightarrow$$
$$\quad (\exists\, usecount' : \mathbb{N} \longrightarrow \mathbb{N};\ current' : \mathbb{N} \mid total\ usecount' \leq MAXCOUNT \bullet$$
$$\quad usecount' = usecount \oplus \{stamp? \mapsto usecount(stamp?) + 1\} \land$$
$$\quad current' = current \land (\forall\, n : \mathbb{N} \bullet n > current' \Rightarrow usecount'\ n = 0))$$

$$
\begin{array}{l}
\forall\, usecount : \mathbb{N} \longrightarrow \mathbb{N};\ current : \mathbb{N} \bullet total\ usecount < MAXCOUNT \land \\
\quad (\forall\, n : \mathbb{N} \bullet n > current \Rightarrow usecount\ n = 0) \Rightarrow \\
\quad total(usecount \oplus \{current \mapsto usecount(current) + 1\}) \leq MAXCOUNT \\
\quad (\forall\, n : \mathbb{N} \bullet n > current \Rightarrow \\
\qquad (usecount \oplus \{current \mapsto usecount(current) + 1\})n = 0)
\end{array}
$$