Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 187 (2007) 55–74
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Derivation of a Scalable Lock-Free Stack Algorithm

Lindsay Groves[1]

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, Wellington, New Zealand

Robert Colvin[2]

ARC Centre for Complex Systems,
School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, Australia

Abstract

We show how a sophisticated, lock-free concurrent stack implementation can be derived from an abstract specification in a series of verifiable steps. The algorithm is a simplified version of one described by Hendler, Shavit and Yerushalmi [6], which allows push and pop operations to be paired off and eliminated without affecting the central stack. This reduces contention on the stack compared with other implementations, and allows multiple pairs of push and pop operations to be performed in parallel.
Our derivation introduces an abstract model of the elimination process, which allows the basic algorithmic ideas to be separated from implementation details, and provides a basis for explaining and comparing different variants of the algorithm. We show that the elimination stack algorithm is linearisable by showing that any execution of the implementation can be transformed into an equivalent execution of an abstract model of a linearisable stack. At each step in the derivation, this transformation reduces an execution of an entire operation at a time of the model at that level, or two in the case of a successful elimination, rather than translating one atomic action at a time as is done in simulation proofs.

Keywords: Concurrent lock-free stack, abstract specification, linearisable stack

1 Introduction

Concurrent algorithms designed to provide good performance under a wide range of workloads typically do not use locks, and use a variety of mechanisms to reduce contention on shared memory and increase the potential for parallel execution. For example, Hendler, Shavit and Yerushalmi [6] present a lock-free stack implementation which allows processes that detect interference while operating on a shared

[1] Email: lindsay@mcs.vuw.ac.nz

[2] Email: robert@itee.uq.edu.au

doi:10.1016/j.entcs.2006.08.044

stack to pair off so that pairs of push and pop operations "eliminate" each other, leaving the central stack unchanged.

Although the basic idea underlying this elimination mechanism is quite simple, the description given in [6] is quite hard to follow as it is presented directly in terms of a concrete implementation. The lack of an abstract description of the elimination mechanism makes it hard to separate the essential ideas from the particular implementation, and to explore and compare alternative implementations. In previous work [3], we have verified a simplified version of Hendler, Shavit and Yerushalmi's algorithm which we discovered while attempting to verify their algorithm. Our proof is based on simulation between Input-Output Automata (IOAs) [9,10] and is fully mechanised in PVS [4], but it too is encumbered with low-level details which obscure the basic ideas underlying the algorithm.

In this paper, we attempt to give a more intelligible presentation of the elimination stack, by showing how our algorithm can be derived in several steps from an abstract specification. We are primarily concerned with showing that the algorithm is linearisable with respect to an abstract specification of a concurrent stack; we will also argue that our algorithm is lock-free. *Linearisability* [8] is the standard safety property for concurrent data structures, and requires that each operation appears to occur atomically at some point between its invocation and its response. *Lock-freedom* is a liveness property which ensures that the system as a whole makes progress, even though individual operations may never terminate. More precisely, a system is lock-free if some operation will always complete within a finite number of steps of the system. [3]

We express our algorithms in a language based on the refinement calculus, with procedures and type declarations [14], and parallel composition [1]. Our procedures use **in**, **out** and **in out** parameters, as in Ada; we also use value-returning procedures and name the return value so that it can be constrained in specification statements. In reasoning about linearisability we are not concerned with termination, so a specification statement $w\!:\![\, R \,]$ is required to establish postcondition R, modifying only variables in w, only if the statement terminates. In writing specifications and invariants, we use Z's mathematical notation [16]; in particular, $\mathrm{seq}\, T$ and $\mathrm{iseq}\, T$ are the sets of sequences and injective sequences over T, $\#s$ is the length of the finite sequence s, $A \nrightarrow B$ is the set of partial functions from A to B, $\mathrm{dom} f$ is the domain of function f, $f \oplus \{x \mapsto y\}$ is the function which is the same as f except at x where its value is y, and $u \lhd f$ is the function obtained from f by removing elements of u from its domain.

We assume a trace semantics similar to that used in [1], except that (following [9]) we define an *execution* to be an alternating sequence of states and actions, starting with a state, and a *trace* to be the sequence of observable actions in an execution. Our notion of refinement is *preservation of linearisability* — at each step

[3] Some authors call this property *nonblocking*; others use nonblocking as a more general term encompassing other progress conditions such as wait-freedom and obstruction-freedom [7].

in our development, we show that every execution of the lower level model can be transformed into an equivalent execution of the higher level model which preserves the order of non-concurrent operations. We justify these steps by reasoning about traces directly in a fairly informal way, rather than using a specific proof rule for refinement.

We begin in Section 2 by presenting a specification for a concurrent stack and taking the first step towards refining this to a lock-free stack, and then in Section 3 we refine this to a simple lock-free stack algorithm using a linked list representation — these sections serve to introduce some of the basic ideas underlying lock-free algorithms and linearisability, and to illustrate our trace-based derivation approach in the context of a simple algorithm. Section 4 then extends the simple algorithm by adding the elimination mechanism, starting with an abstract specification of elimination and deriving its implementation. Section 5 presents our conclusions.

2 Specifying an Abstract Lock-Free Stack

We consider a system consisting of a finite set of concurrent processes which access a shared stack with elements of some type T. Each process occasionally performs an operation on the stack, and otherwise performs actions which do not involve the stack. We can model such a system by abstracting away from its other behaviour and just considering its stack operations.

2.1 An abstract concurrent stack

At the most abstract level the behaviour of a system involving a concurrent stack is described by a set of processes each performing a non-deterministically chosen sequence of stack operations (see Figure 1). Since we have abstracted away from the rest of the program, which would otherwise provide values to be pushed and use values that are popped, the values to be pushed onto the stack are chosen non-deterministically, whereas values returned by *pop* are determined by the contents of the stack at the time and are then discarded. We write $||_{p \in \mathcal{P}}\ op_p$ for the parallel composition of processes drawn from a finite set $\mathcal{P}$, each executing an operation op_p, and usually omit the p subscript when the operation does not depend on p. Parallel composition is defined in terms of interleaving of atomic steps, as in [1].

$$\begin{array}{l} \mathit{STACK} \mathrel{\widehat{=}} \\ \quad \mathbf{var}\ s : \mathit{Stack} := \mathit{EmptyStack}\ ; \\ \quad \Big|\Big|_{p\in\mathcal{P}} \left(\begin{array}{l}\mathbf{var}\ y : T_\perp\ ; \\ \mathbf{do}\ \mathit{true} \rightarrow ([]_{x:T}\ \mathit{push}_p(x))\ []\ \mathit{pop}_p(y)\ \mathbf{od}\end{array}\right)\end{array}$$

Fig. 1. Abstract specification for a system involving a concurrent stack

All of our programs will have this high level structure, but will use different versions of *push* and *pop*. We use superscripts (e.g. $\mathit{push}^1(x)$, $\mathit{pop}^2(y)$) where we need to distinguish different versions of the stack operations, and we write $STACK^k$ to denote a version of $STACK$ using operations $push^k$ and pop^k. Unless stated otherwise, any operations not defined explicitly are assumed to be defined as in the previous version, but using new components from the current version. For example, in Figure 4, $push^3$ and pop^3 are the same as $push^2$ and pop^2, but using $tryPush^3$ and $tryPop^3$ instead of $tryPush^2$ and $tryPop^2$, respectively.

In our initial model, we regard *push* and *pop* actions as being atomic, so a trace is a sequence of *push* and *pop* actions which is valid according to the semantics of these stack operations, and this stack is clearly linearisable. We specify the stack operations using a model-based approach, treating an abstract stack as a sequence of values of its component type (i.e. $Stack \mathrel{\widehat{=}} \text{seq } T$). Thus, *push* and *pop* are defined as shown in Figure 2. Since we wish to implement a lock-free stack, a *pop* on an empty stack cannot wait for the stack to become non-empty, but instead returns a distinguished value, $\perp \notin T$, indicating that the stack was empty, and the result type for *pop* is $T_\perp \mathrel{\widehat{=}} T \cup \{\perp\}$.

$$\begin{array}{ll} push^1(\mathbf{in}\ x : T) \mathrel{\widehat{=}} & \qquad pop^1(\mathbf{out}\ y : T_\perp) \mathrel{\widehat{=}} \\ \quad s \mathbin{\boldsymbol{:}} \left[\, s = \langle x \rangle \frown s_0 \,\right] & \qquad\quad s, y \mathbin{\boldsymbol{:}} \left[\, s = s_0 = \langle\rangle \wedge y = \perp \ \vee\ s_0 = \langle y \rangle \frown s \,\right] \end{array}$$

Fig. 2. Abstract specification for stack operations

2.2 An abstract lock-free stack

We are going to use a linked list representation for the stack, which means that in performing a stack operation, we need to read the value of the top of stack pointer, perform some other steps, and then update the top of stack pointer. Since this cannot be implemented atomically, we need to consider the possible effects that other processes may have between when we read the top of stack pointer and when we update it. Rather than taking the traditional approach of preventing interference by using synchronisation mechanisms such as locks or semaphores, we will obtain a lock-free implementation by instead detecting interference when it occurs, using a hardware CAS operation (see Section 3). We will then implement *push* and *pop* by repeatedly attempting to perform the operation until it is successfully performed without interference. Thus, we define operations *tryPush* and *tryPop* (see Figure 3), which attempt to perform a *push* or *pop*, respectively, and either "succeed"(returning *true*), or "fail" (returning *false*). [4]

In this model, we regard *tryPush* and *tryPop* as being atomic, so a trace of $STACK^2$ is a semantically valid sequence of these operations. We distinguish suc-

[4] We write **do** S **od** as an abbreviation for $[\![\mathbf{var}\ r : bool := false\ \mathbf{;}\ \mathbf{do}\ \neg\, r \rightarrow r := S\ \mathbf{od}]\!]$, where S assigns a value to a boolean variable r. We use similar abbreviations later when the loop body is a more complex statement assigning to r.

```
$push^2(\textbf{in}\ x : T) \mathrel{\widehat=}$
    do
        $tryPush^2(x)$
    od

$tryPush^2(\textbf{in}\ x : T)\ r : bool \mathrel{\widehat=}$
    $s, r \colon \left[\begin{array}{l} s = \langle x \rangle \frown s_0 \wedge r = true\ \vee \\ s = s_0 \wedge r = false \end{array}\right]$

$pop^2(\textbf{out}\ y : T_\perp) \mathrel{\widehat=}$
    do
        $tryPop^2(y)$
    od

$tryPop^2(\textbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat=}$
    $s, y, r \colon \left[\begin{array}{l} s = s_0 = \langle\rangle \wedge y = \perp \wedge r = true\ \vee \\ s_0 = \langle y \rangle \frown s \wedge r = true\ \vee \\ s = s_0 \wedge r = false \end{array}\right]$
```

Fig. 3. Abstract specification for lock-free stack operations

cessful and failed occurrences of these operations by appending "S" and "F", respectively.

To show that this is a valid refinement (i.e. that linearisability is preserved), we need to show that for any trace of $STACK^2$ there is an equivalent trace of $STACK^1$. Now, any execution of $STACK^2$ containing a completed *push* operation by process p has a trace consisting of zero or more failed occurrences of $tryPush^2_p(x)$, followed by one successful occurrence, interleaved with actions of other processes; i.e. the trace has the form $\alpha_1\ tryPushF^2_p(x)\ \alpha_2 \cdots tryPushF^2_p(x)\ \alpha_n\ tryPushS^2_p(x)\ \alpha_{n+1}$, for some $n \geq 1$, where $\alpha_1 \cdots \alpha_{n+1}$ are (possibly empty) sequences of *tryPush* and *tryPop* actions containing no p-actions, and any p-action in α_1 is part of a completed operation contained within α_1.[5]

Only the successful *tryPush* operation has any observable effect, so the others can be discarded. Ignoring the local assignment to r, a successful execution of $tryPush^2$ is equivalent to an execution of $push^1$. Thus, the above trace is equivalent to $\alpha_1\ \alpha_2 \cdots \alpha_n\ push^1_p(x)\ \alpha_{n+1}$; i.e. the observable effect is the same as if activation of the *push* had been delayed until after α_n and then completed without interference.

Similarly, any execution of $STACK^2$ containing a completed *pop* operation by process p has a trace of the form $\alpha_1\ tryPopF^2_p(y)\ \alpha_2 \cdots tryPopF^2_p(y)\ \alpha_n\ tryPopS^2_p(y)$ α_{n+1}, which is equivalent to $\alpha_1\ \alpha_2 \cdots \alpha_n\ pop^1_p(y)\ \alpha_{n+1}$.

Thus, by induction on the number of stack operations, any execution of $STACK^2$ can be transformed into an equivalent trace of $STACK^1$, which induces an equivalent execution of $STACK^1$. Each operation of $STACK^2$ is either discarded, or mapped to an atomic action of $STACK^1$ which lies between the first and last actions of this execution of $STACK^2$, so the ordering of non-current operations is preserved, as required for linearisability. Note that in performing this transformation one operation at a time, we construct intermediate executions combining actions of both $STACK^1$ and $STACK^2$.

In showing linearisability, we can assume that each "try" operation chooses nondeterministically whether to succeed or fail; we can also ignore the issue of whether the loops in *push* and *pop* terminate. In order to show lock-freedom, we need to show that it is not possible for all operations to always fail, since then no

[5] We will assume these constraints in justifying subsequent refinements without mentioning them explicitly.

operation would ever terminate. In Section 3 we will show that the implementation of a "try" operation only fails if it experiences interference; however, that property cannot (easily) be expressed at this level of abstraction.

3 Deriving a Simple Lock-Free Stack

We now introduce a linked list data structure to represent the stack, and rewrite the specifications for *tryPush* and *tryPop* to use this representation (see Figure 4). We use new types *Ptr* and *Node* to represent pointers and nodes, and model the heap explicitly as a partial function h from pointers to nodes. *Ptr* is assumed to contain a distinguished value *null*, which is never in the domain of h. We discuss the conjunct $h \approx h_0$ below.

$\mathbf{type}\ \mathit{Ptr}$

$\mathbf{type}\ \mathit{Node} = (\mathit{val} : T\ ;\ \mathit{next} : \mathit{Ptr})$

$$\mathit{tryPush}^3(\mathbf{in}\ x : T)\ r : \mathit{bool} \mathrel{\widehat=}$$
$$\begin{array}{r} \mathit{Top}, \\ h, \\ r \end{array} : \left[\begin{array}{l} (\ \exists\, n : \mathit{Ptr} \bullet \\ \quad n \notin \operatorname{dom} h_0 \wedge \mathit{Top} = n \wedge \\ \quad h = h_0 \oplus \{n \mapsto (x, \mathit{Top}_0)\} \wedge \\ \quad r = \mathit{true}\ )\ \vee \\ \mathit{Top} = \mathit{Top}_0 \wedge h \approx h_0 \wedge r = \mathit{false} \end{array}\right]$$

$\mathbf{const}\ \mathit{null} : \mathit{Ptr}$

$\mathbf{var}\ h : (\mathit{Ptr} \setminus \{\mathit{null}\}) \nrightarrow \mathit{Node} := \varnothing$

$\mathbf{var}\ \mathit{Top} : \mathit{Ptr} := \mathit{null}$

$$\mathit{tryPop}^3(\mathbf{out}\ y : T_\perp)\ r : \mathit{bool} \mathrel{\widehat=}$$
$$\begin{array}{r} \mathit{Top}, \\ y, \\ r \end{array} : \left[\begin{array}{l} \mathit{Top} = \mathit{Top}_0 = \mathit{null} \wedge y = \perp \wedge r = \mathit{true}\ \vee \\ h(\mathit{Top}_0) = (y, \mathit{Top}) \wedge r = \mathit{true}\ \vee \\ \mathit{Top} = \mathit{Top}_0 \wedge r = \mathit{false} \end{array}\right]$$

Fig. 4. Concrete stack specification

We define a state invariant, *Inv*, which ensures that the linked list is well-formed by postulating the existence of a sequence of unique locations corresponding to the nodes in the linked list. $\mathit{Inv}(\mathit{Top}, h)$ is assumed as an implicit pre- and postcondition of $\mathit{tryPush}^3$ and tryPop^3 and all operations derived from them.

$$\begin{array}{lcl} \mathit{Inv}(\mathit{Top}, h) & \mathrel{\widehat=} & \exists f : \operatorname{iseq} \mathit{Ptr} \bullet \mathit{Rep}(f, \mathit{Top}, h) \\ \mathit{Rep}(f, \mathit{Top}, h) & \mathrel{\widehat=} & (\mathit{Top} = \mathit{null} \Rightarrow \# f = 0) \wedge \\ & & (\mathit{Top} \neq \mathit{null} \Rightarrow \# f > 0 \wedge h(f(\# f)).\mathit{next} = \mathit{null} \wedge \\ & & \quad (\forall\, i : 1 \mathrel{..} \# f - 1 \bullet h(f(i)).\mathit{next} = f(i+1))\,) \end{array}$$

To prove that this step is a valid refinement, we just need to show that any occurrence of $\mathit{tryPush}^3(x)$ or $\mathit{tryPop}^3(y)$ can be replaced by $\mathit{tryPush}^2(x)$ or $\mathit{tryPop}^2(y)$, respectively. We do this using standard data refinement techniques [5], using an abstraction relation, *Abs*, which maps the sequence of values stored in the linked list onto the abstract stack:

$$\begin{array}{lcl} \mathit{Abs}(s, \mathit{Top}, h) & \mathrel{\widehat=} & \exists f : \operatorname{iseq} \mathit{Ptr} \bullet \\ & & \quad \mathit{Rep}(f, \mathit{Top}, h) \wedge \\ & & \quad \# s = \# f \wedge (\forall\, i : 1 \mathrel{..} \# f \bullet s(i) = f(i).\mathit{val}\,) \end{array}$$

We now wish to refine $\mathit{tryPush}^3$ and tryPop^3 so that they can be implemented using atomic instructions on a standard computer architecture. This essentially means that tests and assignments should involve no more than one access (read or write) on a shared location, with the exception of the CAS (Compare And Swap) instruction. $CAS(loc, old, new)$ takes the address of a memory location (loc), an "expected" value (old), and a "new" value (new). If the location contains the expected value, the CAS *succeeds*, atomically storing the new value into the location and returning *true*; otherwise, the CAS *fails*, returning *false* and leaving the location unchanged. This is formally specified, for the case where all three arguments are pointers, on the right of Figure 5.

The crucial step in $tryPush^3$ is the update to Top. We have to allocate the new node and initialise its fields, and then set Top to point to the new node, provided that it has not changed since we read the value stored in the *next* field of the new node. To do this, we store a "snapshot" (ss) of Top and use it to initialise the new node, and then set Top to the new node provided that its value has not changed since we took the snapshot (see Figure 5 — the labels on the right will be used later to refer to these specification statements).

$$
\begin{array}{l}
tryPush^4(\textbf{in } x : T)\ r : bool \mathrel{\widehat{=}} \\
\quad \textbf{var } n, ss : Ptr\ ; \\
\quad n, ss, h : \left[\begin{array}{l} n \notin \operatorname{dom} h_0 \wedge ss = Top\ \wedge \\ h = h_0 \oplus \{n \mapsto (x, ss)\} \end{array} \right] ; \qquad (A) \\
\quad Top, r : \left[\begin{array}{l} Top_0 = ss \wedge Top = n \wedge r = true\ \vee \\ Top = Top_0 \wedge r = false \end{array} \right] \quad (B)
\end{array}
$$

$$
\begin{array}{l}
CAS(\textbf{in out } loc : Ptr\ ; \\
\qquad \textbf{in } old, new : Ptr)\ r : bool \mathrel{\widehat{=}} \\
\quad loc, r : \left[\begin{array}{l} loc_0 = old \wedge loc = new \wedge r = true\ \vee \\ loc_0 \neq old \wedge loc = loc_0 \wedge r = false \end{array} \right]
\end{array}
$$

Fig. 5. Refining *tryPush* and defining CAS

The conjunct $h \approx h_0$ in $tryPush^3$ is defined to hold when h and h_0 represent the same abstract stack, i.e. $\forall\, s \bullet Abs(s, Top, h) \Leftrightarrow Abs(s, Top, h_0)$. This means that a failed $tryPush^3$, or $tryPush^4$, may modify the heap in a way that does not affect the abstract stack. In particular, it allows us to allocate the new node, n, in A, irrespective of the outcome of B.

Now, any execution of $STACK^4$ containing a completed *tryPush* operation by process p has a trace of the form $\alpha\, A_p\, \beta\, B_p\, \gamma$. We will assume (i) that α, β and γ preserve $Inv(Top, h)$, and (ii) that it is not possible for β to change Top from the value observed in A_p to another value and back again (i.e. β cannot take the system from state σ through σ' to σ'', where $\sigma(Top) = \sigma''(Top) \neq \sigma'(Top)$). Assumption (i) is an implicit postcondition of *tryPush* and *tryPop* and can easily be verified later; we will return to assumption (ii) below.

If B_p succeeds, then Top has the same value when p executes B as it had when p executed A, which has been saved in ss. In this case, $A_p\, \beta = \beta\, A_p$, since by assumption (ii) above, β does not change Top, and changes to h in either A_p or β do not affect the other. Thus, the above trace is equivalent to $\alpha\, \beta\, A_p\, B_p\, \gamma$, which in turn is equivalent to $\alpha\, \beta\, tryPushS^3_p(x)\, \gamma$. If B_p fails, then $\beta\, B_p = B_p\, \beta$ and the above trace is equivalent to $\alpha\, A_p\, B_p\, \beta\, \gamma$, which is equivalent to $\alpha\, tryPushF^3_p(x)\, \beta\, \gamma$.

We now further refine A to a sequence of assignments, and implement B as a CAS. We write the resulting program in a more concrete pseudocode, as shown in Figure 6, by further defining Ptr to be **pointer to** $Node$, omitting explicit reference to the heap, and assuming that **new** $Node()$ allocates a new heap location.

```
**type** $Ptr$ = **pointer to** $Node$

$tryPush^5$(**in** $x : T$) $r : bool \mathrel{\widehat{=}}$
    **var** $n, ss : Ptr$ ;
    $n :=$ **new** $Node()$ ;          $(P1)$
    $n.val := x$ ;                    $(P2)$
    $ss := Top$ ;                     $(P3)$
    $n.next := ss$ ;                  $(P4)$
    $r := CAS(Top, ss, n)$            $(P5)$

$tryPop^5$(**out** $y : T_\perp$) $r : bool \mathrel{\widehat{=}}$
    **var** $ss, ssn : Ptr$ ;
    $ss := Top$ ;
    **if** $ss = null$ **then**
        $y := \perp$ ;
        $r := true$
    **else**
        $ssn := ss.next$ ;
        $y := ss.val$ ;
        $r := CAS(Top, ss, ssn)$
    **fi**
```

Fig. 6. Concrete stack implementation

To show that this is a valid refinement, we need to show that for any trace of $STACK^5$ there is an equivalent trace of $STACK^4$. Now, any execution of $STACK^5$ containing a completed *tryPush* operation by process p has a trace t of the form $\alpha\, P1_p\, \beta\, P2_p\, \gamma\, P3_p\, \delta\, P4_p\, \epsilon\, P5_p\, \zeta$. Since $P2$ and $P4$ only access local variables, they can be moved relative to γ and δ. We will also assume that the effect of $P1$ cannot be seen by other processes (i.e. no process can see what pointers have been allocated but are not yet part of the linked list), so we can also move $P1$. Trace t is thus equivalent to $\alpha\, \beta\, \gamma\, P1_p\, P2_p\, P3_p\, P4_p\, \delta\, \epsilon\, P5_p\, \zeta$, which is equivalent to $\alpha\, \beta\, \gamma\, A_p\, \delta\, \epsilon\, B_p\, \zeta$, which contains a completed $tryPush^4$. Since the linked list can only be altered by the CAS in $P5$, it is easy to see that *tryPush* preserves $Inv(Top, h)$: if f is the sequence guaranteed by Inv beforehand, then $\langle n \rangle \frown f$ is the sequence required by Inv afterwards.

We likewise refine $tryPop^4$ to pseudocode (see Figure 6). The details are similar, though complicated a little by the need to introduce an **if** statement, and are omitted for brevity.

We can improve the resulting code by changing the interfaces to *tryPush* and *tryPop*, so that their arguments are pointers to nodes, rather than values. We can then allocate a new node and initialise it with x in *push* rather than in *tryPush*, and pass a pointer to it to *tryPush*, so that only one node is allocated per *push*. We can also extract the return value in *pop*, so it is only done once, rather than being done in every execution of *tryPop*. We omit this version due to space restrictions, however, the modifications to *push* and *pop* are embodied in Figure 12.

This algorithm is *lock-free*, provided that **new** is lock-free, since a "try" operation will only fail if another process successfully executes a CAS. Since we assume that there are a finite number of processes, this can only occur an infinite number of times if an infinite number of operations are completed.

To show that *Inv* is maintained when new nodes are added to the linked list, and to justify assumption (ii) above, we assume that when *push* acquires a new node, there is no pointer to it already in the linked list or in a local variable of another process. This can be ensured if every *push* allocates a new node and storage is never released (note that *pop* cannot modify h). If we allow storage to be reused, it is possible that between a *push* taking a snapshot and its CAS, the node at the top of the stack could be popped by another process and then pushed again by another process, after the rest of the stack has changed. This is known as the *ABA problem*.

One solution to the ABA problem is to assume that the implementation language provides automatic garbage collection. Another solution is for the implementation to maintain its own free list and for every pointer variable to have a *modification count* which is incremented every time the variable is modified. This way, if a node is released to the free list and later returned to the head of the linked list, the snapshot stored in *ss* will be different from *Top* because its modification count has changed. Both of these solutions can be introduced as a further data refinement, so that h is an abstraction of the memory management system actually used.

The "solution" using a free list and modification counts is only strictly correct if modification counts are unbounded. To be practical, we must assume that a pointer and its modification count can be tested and assigned atomically using a CAS — e.g. if a pointer requires 32 bits and a CAS operates on 64-bit values. In that case, however, we only have 32 bits for the modification count, so it is still possible for the modification count to wrap around and return to the same value as the snapshot. The chance of this actually occurring can be shown to be extremely small [12], and is generally assumed to be small enough to make this solution acceptable for practical purposes. This approach is adopted in Treiber's stack [15], and in many other lock-free algorithms (e.g. Michael and Scott's queue [11]).

4 The Elimination Stack

The stack implementation presented in Section 3 works well at medium loads, but does not scale well [6]. When a large number of processes access the stack concurrently, they all compete to read and update the shared *Top* location, resulting in a large amount of interference. Also, since all operations must update a single shared location, *Top*, all stack operations must be performed in a strictly sequential fashion — there is no possibility of operations running on separate processors actually being performed in parallel. To obtain better performance under high loads, while maintaining good performance under low to medium loads, Hendler, Shavit and Yerushalmi [6] propose an algorithm which incorporates two key ideas.

Firstly, if a process fails in its attempt to apply an operation, it waits before trying again. This kind of "backoff" mechanism is a common way to reduce contention in concurrent systems. For example, with exponential backoff, a process doubles its delay time each time it retries its operation. This reduces contention, and can improve throughput in many cases; but it can also result in processes waiting too long, which then reduces throughput.

Secondly, a *push* and a *pop* can be paired and eliminated, passing the pushed value to the *pop* operation, and leaving the stack unchanged. The elimination does not create interference with any operations on the central stack, and multiple eliminations may occur in parallel. This can be combined with the backoff mechanism described above, so that a process that is waiting to retry an operation looks for a complementary operation with which to eliminate.

In [6], a process p performing a stack operation first attempts its operation on the stack, as described in Section 3. If this attempt fails, instead of immediately retrying, p attempts to match up with a complementary operation so that both operations can be eliminated. If the elimination attempt fails, p tries its operation on the stack again. This strict alternation between attempting the operation on the stack and attempting to eliminate may not give optimal performance under all conditions, so it may be better to use an adaptive scheme to determine, for each attempt, whether to try on the stack or to try to eliminate (this approach is taken in [13] to implement a scalable lock-free queue).

4.1 An abstract model of elimination

The basic idea of elimination is that, instead of immediately retrying a failed operation on the central stack, a process may try to find another process performing a complementary operation. This can be likened to a service, such as an employment or accommodation service, where customers who are either offering or seeking some resource are normally served by a clerk. [6] However, if the clerk is busy, instead of waiting to be served, customers may resort to some other way of meeting their requirements, for example by posting or inspecting notices on a noticeboard. We will use this analogy in developing an abstract model of elimination.

In proving linearisability, we will simply treat elimination as an alternative way in which an operation may satisfy its specification. Thus, we obtain $STACK^6$ (see Figure 7) from $STACK^2$ (Figure 3), by refining the loop bodies in *push* and *pop* so that at each attempt an operation makes a nondeterministic choice between trying on the stack and trying to eliminate. The new operations, $tryPushElim^6$ and $tryPopElim^6$, have the same specifications as $tryPush^6$ and $tryPop^6$ (except that $tryPopElim^6$ cannot return $\perp$), but will be implemented differently.

```
$push^6(\textbf{in}\ x : T) \mathrel{\widehat=}$
    do
        $tryPush^6(x)\ [\!]\ tryPushElim^6(x)$
    od

$tryPushElim^6(\textbf{in}\ x : T)\ r : bool \mathrel{\widehat=}$
    $s, r : \left[\begin{array}{l} s = \langle x \rangle \frown s_0 \wedge r = true \vee \\ s = s_0 \wedge r = false \end{array}\right]$

$pop^6(\textbf{out}\ y : T_\perp) \mathrel{\widehat=}$
    do
        $tryPop^6(y)\ [\!]\ tryPopElim^6(y)$
    od

$tryPopElim^6(\textbf{out}\ y : T)\ r : bool \mathrel{\widehat=}$
    $s, y, r : \left[\begin{array}{l} s_0 = \langle y \rangle \frown s \wedge r = true \vee \\ s = s_0 \wedge r = false \end{array}\right]$
```

Fig. 7. Introducing elimination ($tryPush^6 \equiv tryPush^2$ and $tryPop^6 \equiv tryPop^2$)

This modification is clearly a valid refinement, since at this level *tryPushElim* and *tryPopElim* are equivalent to *tryPush* and *tryPop*, respectively, so occurrences

[6] To make the analogy work for a stack, we assume that all resources are interchangeable, and that the clerk simply keeps a pile of (descriptions of) available resources, adding resources offered to the top of the pile and always handing out the resource at the top of the pile.

of *tryPushElim* and *tryPopElim* are transformed in the same way that occurrences of *tryPush* and *tryPop* were transformed in Section 2.2. In order to prove lock-freedom, we would need to show that an operation cannot continually try to eliminate without ever trying to perform its operation on the stack, since *tryPushElim* and *tryPopElim* can fail without another stack operation being completed.

To describe elimination in more detail, we need to consider the rôles of the two processes involved. We will assume (as in [6]) that an elimination is initiated by one of these processes. [7] Thus, a process attempting to find a matching process to eliminate with may proceed in either of two ways:

- A *passive* approach in which it places a request on the noticeboard describing the resource it is offering or seeking, waits for a while, then removes its request from the noticeboard and checks to see if its request has been fulfilled.
- An *active* approach in which it looks at the requests on the noticeboard to see if there is one that matches its requirements, and if so, marks that request as being completed and transfers the offered resource to the seeking process.

Thus, the noticeboard can be viewed as a set of *requests*, each describing an operation which is either waiting to be performed or has already been performed. Since there can be at most one request associated with each process, we will model the noticeboard (N) as a partial function from processes to requests. [8]

A waiting request must describe the operation to be performed, i.e. whether it is a *push* or a *pop*, and for the former the value to be pushed. A completed *pop* request must specify the value to be returned. We will model requests as pairs of the form (op, val), where op is either $PUSH$ or POP, and val is the value to be pushed for a waiting *push* request, the value to be returned for a completed *pop* request, and $\perp$ for a completed *push* or waiting *pop* request.

A process attempting an elimination chooses whether to take an active approach or a passive approach; at this level, we can regard this as a nondeterministic choice (see Figure 8).

In $tryActivePush^7$ and $tryActivePop^7$, the first disjunct describes active elimination as outlined above: q is some process attempting a complementary operation, and its request is modified to show that it has been fulfilled. To obtain the intended implementation, we have to allow a failing active eliminator to remove another process's unfulfilled request from the noticeboard, and allow an active elimination attempt to fail even when there is a potential partner available — these provisions are embodied in the second and third disjuncts.

To model a passive eliminator p waiting, we split a *tryPassive* operation into two actions, so that an arbitrary number of actions of other processes can occur

[7] In a more expressive semantic model, we could treat elimination as an atomic action involving two processes. Alternatively, we could use a separate set of dedicated "match maker" processes to select pairs of processes for elimination.

[8] To avoid lots of domain membership tests, we assume that a statement about $N(p)$ can only be true if p is in the domain of N.

$$\textbf{type}\ OP = PUSH \mid POP \qquad\qquad \textbf{var}\ N : \mathcal{P} \nrightarrow Request := \varnothing$$

$$\textbf{type}\ Request = \{op : OP\ ;\ val : T_\perp\}$$

$$\begin{array}{l} tryPushElim^7(\textbf{in}\ x : T)\ r : bool \mathrel{\widehat=} \\ \qquad r := tryActivePush^7(x) \\ \quad [] \quad r := tryPassivePush^7(x) \end{array} \qquad \begin{array}{l} tryPopElim^7(\textbf{out}\ y : T)\ r : bool \mathrel{\widehat=} \\ \qquad r := tryActivePop^7(y) \\ \quad [] \quad r := tryPassivePop^7(y) \end{array}$$

$$\begin{array}{l} tryActivePush^7(\textbf{in}\ x : T)\ r : bool \mathrel{\widehat=} \\ \begin{array}{c} N, \\ r \end{array} : \left[\begin{array}{l} (\ \exists\, q : \operatorname{dom} N_0 \bullet \\ \quad N_0(q) = (POP, \perp) \wedge \\ \quad N = N_0 \oplus \{q \mapsto (POP, x)\} \wedge \\ \quad r = true\ ) \\ \qquad \vee \\ (\ \exists\, q : \operatorname{dom} N_0 \bullet \\ \quad N = \{q\} \lhd N_0 \wedge r = false\ ) \\ \qquad \vee \\ N = N_0 \wedge r = false \end{array}\right] \end{array} \qquad \begin{array}{l} tryActivePop^7(\textbf{out}\ y : T)\ r : bool \mathrel{\widehat=} \\ \begin{array}{c} N, \\ y, \\ r \end{array} : \left[\begin{array}{l} (\ \exists\, q : \operatorname{dom} N_0, v : T \bullet \\ \quad N_0(q) = (PUSH, v) \wedge \\ \quad N = N_0 \oplus \{q \mapsto (PUSH, \perp)\} \wedge \\ \quad y = v \wedge r = true\ ) \\ \qquad \vee \\ (\ \exists\, q : \operatorname{dom} N_0 \bullet \\ \quad N = \{q\} \lhd N_0 \wedge r = false\ ) \\ \qquad \vee \\ N = N_0 \wedge r = false \end{array}\right] \end{array}$$

$$\begin{array}{l} tryPassivePush^7_p(\textbf{in}\ x : T)\ r : bool \mathrel{\widehat=} \\ N : [\,N = N_0 \cup \{p \mapsto (PUSH, x)\}\,]\ ; \qquad (C) \\ \begin{array}{c} N, \\ r \end{array} : \left[\begin{array}{l} N_0(p).val = \perp \wedge \\ N = \{p\} \lhd N_0 \wedge r = true \\ \qquad \vee \\ N_0(p).val = x \wedge \\ N = \{p\} \lhd N_0 \wedge r = false \\ \qquad \vee \\ p \notin \operatorname{dom} N_0 \wedge N = N_0 \wedge r = false \end{array}\right] \quad (D) \end{array} \qquad \begin{array}{l} tryPassivePop^7_p(\textbf{out}\ y : T)\ r : bool \mathrel{\widehat=} \\ N : [\,N = N_0 \cup \{p \mapsto (POP, \perp)\}\,]\ ; \\ \begin{array}{c} N, \\ y, \\ r \end{array} : \left[\begin{array}{l} N_0(p).val = y \wedge \\ N = \{p\} \lhd N_0 \wedge r = true \\ \qquad \vee \\ N_0(p).val = \perp \wedge \\ N = \{p\} \lhd N_0 \wedge r = false \\ \qquad \vee \\ p \notin \operatorname{dom} N_0 \wedge N = N_0 \wedge r = false \end{array}\right] \end{array}$$

Fig. 8. Abstract description of elimination

between them — there is no need to model the delay itself. In $tryPassivePush^7$ and $tryPassivePop^7$, the first specification statement adds p's request to the noticeboard, while the second describes p's behaviour after its delay. The elimination attempt will succeed if the val field of p's request has changed (first disjunct), and will fail if the request is unchanged (second disjunct) or has been removed (third disjunct).

To show that this is a valid refinement, we show that any trace of $STACK^7$ can be transformed into an equivalent trace of $STACK^6$ in a way that preserves the order of non-concurrent operations. We first consider successful elimination. A successful $tryPassivePush$ by process p consists of two actions: C_p, which adds a request to N, and D_p, which later finds that p's request has been modified. Assuming that a process can only add its own request and can only modify the val field of another process's request (which is easy to verify), this must be the same request that was posted by C_p, and the modification can only have occurred because a successful $tryActivePop$ by another process, say q, has occurred between C_p and D_p. Thus, an execution containing a successful $tryPassivePush_p$ has a trace t of the form $\alpha\, C_p\, \beta\, tryActivePopS_q\, \gamma\, D_p\, \delta$. The fact that $tryActivePop_q$ and D_p both succeed implies that β and γ do not affect $N(p)$, so t is equivalent to $\alpha\, \beta\, C_p\, tryActivePopS_q(x)\, D_p\, \gamma\, \delta$. The combined effect of these three actions is the same as pushing a value onto the stack and then immediately popping it off the stack, so t is equivalent to $\alpha\, \beta\, tryPushElimS^6(x)_p\, tryPopElimS^6(x)_q\, \gamma\, \delta$, which is equivalent to to $\alpha\,\beta\,push^6(x)_p\; pop^6(x)_q\,\gamma\,\delta$. Placing these operations in the position of the successful *tryActivePop* ensures that both actions occur during the execution of both eliminating operations, and thus preserves the order of non-concurrent operations. Conversely, a successful *tryActivePop* must occur between the C_p and D_p actions of a successful *tryPassivePush*, so is covered by this translation. We can similarly replace a successful $tryActivePush^7$ and a successful $tryPassivePop^7$ by a $tryPushElim^6$ followed by a $tryPopElim^6$.

We next consider unsuccessful elimination. An unsuccessful *tryActivePush* or *tryActivePop* may delete another process's waiting request, causing that process's elimination attempt to fail, but otherwise has no observable effect, and can be mapped to a *tryPushF* or *tryPopF*, respectively (or discarded). An unsuccessful *tryPassivePush* or *tryPassivePop* adds a request to N in C, but ensures that this request is no longer present in D, so can be mapped to a $tryPushElimF^6$ or $tryPopElimF^6$, respectively (or discarded).

4.2 Combining push and pop elimination

At this point we observe that there is considerable similarity between *tryActivePush* and *tryActivePop*, and between *tryPassivePush* and *tryPassivePop*. We thus combine them by introducing procedures *tryActive* and *tryPassive* which take an additional parameter indicating the kind of operation to be performed, as shown in Figure 9. The second parameter of *tryActive* and *tryPassive* is now an **in out** parameter, which is used as an **in** parameter when the first argument is *PUSH* and as an **out** parameter when the first argument is *POP*.

$$
\begin{array}{l}
tryPushElim^8(\mathbf{in}\ x : T)\ r : bool \mathrel{\widehat=}\\
\qquad\quad r := tryActive^8(PUSH, x)\\
\quad [\,]\quad r := tryPassive^8(PUSH, x)
\end{array}
\qquad
\begin{array}{l}
tryPopElim^8(\mathbf{out}\ y : T)\ r : bool \mathrel{\widehat=}\\
\quad y := \bot\ ;\\
\quad \left(\begin{array}{l} \quad\ r := tryActive^8(POP, y)\\ [\,]\ \ r := tryPassive^8(POP, y)\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
tryActive^8(\mathbf{in}\ op : OP\ ;\ \mathbf{in\ out}\ v : T_\bot)\ r : bool \mathrel{\widehat=}\\
\begin{array}{c} N,\\ v,\\ r\end{array} \mathbf{:} \left[\begin{array}{l}
(\ \exists\, q : \mathrm{dom}\, N_0 \bullet N_0(q).op \neq op\ \wedge\\
\quad (N_0(q).val = \bot \Leftrightarrow N_0(q).op = POP)\ \wedge\\
\quad N = N_0 \oplus \{q \mapsto (N_0(q).op, v_0)\}\ \wedge\\
\quad v = N_0(q).val \wedge r = true\ )\\
\qquad \vee\\
(\ \exists\, q : \mathrm{dom}\, N_0 \bullet\\
\quad N = \{q\} \lhd N_0 \wedge r = false\ )\\
\qquad \vee\\
N = N_0 \wedge r = false
\end{array}\right]
\end{array}
$$

$$
\begin{array}{ll}
tryPassive^8_p(\mathbf{in}\ op : OP\ ;\ \mathbf{in\ out}\ v : T_\bot)\ r : bool \mathrel{\widehat=} & \\
N \mathbf{:} \left[\, N = N_0 \cup \{p \mapsto (op, v)\}\,\right]\ ; & (TP1)\\
\begin{array}{c} N,\\ v,\\ r\end{array} \mathbf{:} \left[\begin{array}{l}
N_0(p).val = v \neq v_0\ \wedge\\
N = \{p\} \lhd N_0 \wedge r = true\\
\qquad \vee\\
N_0(p).val = v_0\ \wedge\\
N = \{p\} \lhd N_0 \wedge r = false\\
\qquad \vee\\
p \notin \mathrm{dom}\, N_0\ \wedge\\
N = N_0 \wedge r = false
\end{array}\right] & (TP2)
\end{array}
$$

Fig. 9. Combining *push* and *pop* elimination

To show that this step is a valid refinement, we must show that we can replace any occurrence of $tryActive^8$ or $tryPassive^8$ by an equivalent operation of $STACK^7$. It is straightforward to show that $tryActive^8(POP, v)$ and $tryPassive^8(POP, v)$ are equivalent to $tryActivePop^7(v)$ and $tryPassivePop^7(v)$, respectively. To show that the *push* cases can be handled similarly, we note that a $tryActive^8(PUSH, v)$ or *tryPassive*$^8(PUSH, v)$ will copy the $\perp$ value from the matching *pop* request into the local variable x of the calling *tryPushElim*, which has no externally visible effect.

4.3 Combining active and passive elimination

We have so far described active and passive elimination as separate actions, because this allows us to define these rôles clearly. However, there is no reason why a process should have to choose at the beginning of an elimination attempt which approach to try. A process may post its request on the noticeboard, and then, instead of waiting idly before checking to see if its request has been fulfilled, proceed to look at other requests in the manner of an active eliminator. If it finds a matching request, the process checks to see if its request has already been fulfilled before proceeding with the elimination. This essentially means that the process tries both approaches, and goes with whichever of them (if either) succeeds. We will thus combine *tryActive* and *tryPassive*, to give a single *tryEliminate* procedure (see Figure 10).

```
$tryPushElim^9(\textbf{in}\ x : T)\ r : bool \;\widehat{=}$
    $r := tryEliminate^9(PUSH, x)$

$tryPopElim^9(\textbf{out}\ y : T)\ r : bool \;\widehat{=}$
    $y := \perp$ ;
    $r := tryEliminate^9(POP, y)$

$tryEliminate^9_p(\textbf{in}\ op : OP\ ;\ \textbf{in out}\ v : T_\perp)\ r : bool \;\widehat{=}$
    var $q : \mathcal{P}$ ;
    $N : [\, N = N_0 \cup \{p \mapsto (op, v)\} \,]$ ;                                  $(TE1)$
    $q : [\, q \in \mathcal{P} \,]$ ;                                                  $(TE2)$
    if $N_0(q).op \neq op$ then                                                        $(TE3)$
        $N, v, r : \left[\begin{array}{l} N_0(p).val = v \neq v_0 \wedge N = \{p\} \lhd N_0 \wedge r = true \\ \quad \vee \\ N_0(p).val = v_0 \wedge (N_0(q).val = \perp \Leftrightarrow N_0(q).op = POP) \wedge \\ N = (\{p\} \lhd N_0) \oplus \{q \mapsto (N_0(q).op, v_0)\} \wedge \\ v = N_0(q).val \wedge r = true \\ \quad \vee \\ N_0(p).val = v_0 \wedge N = \{p, q\} \lhd N_0 \wedge r = false \\ \quad \vee \\ p \notin \mathrm{dom}\, N_0 \wedge N = N_0 \wedge r = false \end{array}\right]$     $(TE4)$
    else
        $N, v, r : \left[\begin{array}{l} N_0(p).val = v \neq v_0 \wedge N = \{p\} \lhd N_0 \wedge r = true \\ \quad \vee \\ N_0(p).val = v_0 \wedge N = \{p\} \lhd N_0 \wedge r = false \\ \quad \vee \\ p \notin \mathrm{dom}\, N_0 \wedge N = N_0 \wedge r = false \end{array}\right]$     $(TE5)$
    fi
```

Fig. 10. Combining active and passive elimination

Following [6], we will assume that an active eliminator only tries one potential elimination partner (q) before giving up, so we can move the selection of q out of the specification statement in *tryActive*, and use the test $N_0(q).op \neq op$ to decide whether to continue with an active elimination attempt or revert to the passive approach. The active elimination case ($TE4$) is now a little more complicated, since a process's request may be fulfilled by another process before it makes its attempt at active elimination. Also, if a process succeeds in active elimination, it must remove the request it posted, so that no other process attempts to eliminate with it.

To show that this step is a valid refinement, we must show that any trace of $STACK^9$ can be transformed into an equivalent trace of $STACK^8$. Now, any execution containing a completed *tryEliminate* by process p has a trace t of the form: $\alpha\ TE1_p\ \beta\ TE2_p\ \gamma\ TE3S_p\ \delta\ TE4_p\ \epsilon$ or $\alpha\ TE1_p\ \beta\ TE2_p\ \gamma\ TE3F_p\ \delta\ TE5_p\ \epsilon$.

In the first disjunct of $TE4_p$, p's request has been changed by another process (in β or γ), and $TE2$ and $TE3$ have no observable effect, so trace t is equivalent to $\alpha\ TP1_p\ \beta\,\gamma\,\delta\ TP2S_p\ \epsilon$, which contains a successful $tryPassive^8$.

In the second disjunct of $TE4_p$, p's request has not been changed and p completes an elimination with process q. The request that $TE1$ added to N is still there when p executes $TE4$, which then removes it, so β, γ and δ are indifferent to its presence. Therefore, since $TE2$ only has local effect, trace t is equivalent to $\alpha\,\beta\,\gamma\,\delta\ TE2_p\ TE3S_p\ TE4S_p\ \epsilon$, which is equivalent to $\alpha\,\beta\,\gamma\,\delta\ tryActiveS_p^8\ \epsilon$.

The third and fourth disjuncts of $TE4_p$ correspond to the second and third disjuncts of *tryActive*. By similar reasoning, in these cases, trace t is equivalent to $\alpha\,\beta\,\gamma\,\delta\ TE2_p\ TES3_p\ TEF4_p\ \epsilon$, which is equivalent to $\alpha\,\beta\,\gamma\,\delta\ tryActiveF_p^8\ \epsilon$.

In the case where $TE3$ fails, since $TE5$ does not depend on the outcomes of $TE2$ and $TE3$, trace t is equivalent to $\alpha\ TP1_p\ \beta\,\gamma\,\delta\ TP2_p\ \epsilon$, and so contains a completed $tryPassive_p^8$, which succeeds iff $TE5$ succeeds.

4.4 A concrete elimination mechanism

To implement the elimination mechanism described above, we must introduce a data structure to represent the noticeboard, which allows the required operations to be implemented in a lock-free fashion. As in the lock-free stack implementation in Section 3, we will use CAS to update shared variables, using local snapshots to detect interference, and allow operations to fail when interference is detected.

We first modify *push* and *pop* as described in Section 3, so that *tryPush* is passed a pointer to a node which *push* has allocated and initialised with x, and a successful *tryPop* returns *null* if the stack was empty and otherwise a pointer to a node containing the popped value (see Figure 11). We then modify *tryEliminate* in the same way, so that its second argument is a pointer.

We represent the noticeboard using an array, *opInfos*, of requests, indexed by process identifiers of type *ProcId*. Requests are again represented by records, however, the *op* field may now assume a third value, *NONE*, indicating that the process is not attempting to perform an operation. This is required because arrays are total functions, and is also used to detect when a request has been fulfilled by another process. We also replace the *val* field by a *node* field, which is a pointer to a *Node*, used in the same way as in *tryPush* and *tryPop*.[9] For a *pop* operation, *node* is initially *null*, and following a successful elimination holds a pointer to a *Node* containing the returned value. We assume that a *Request* value can be stored atomically using a CAS. This is at least as practical as using modification counts, as discussed in Section 3, since we only need to store a two bit *op* field along with the pointer value.

We still have to determine how a process p selects another process q as a potential elimination partner. We could, for instance, choose q to be an arbitrary value in $\mathcal{P}$ and inspect $opInfos[q]$ to see if q is attempting to perform a complementary operation, or we could search through *opInfos* to find a suitable candidate. The approach taken in [6] uses a second array, called *collision*, which is indexed by integers and whose values are process ids. A process advertises that it is available for elimination and selects a potential elimination partner, by choosing an arbitrary location in *collision*. It reads the process id in that location and stores its own id in its place. This is done using a CAS (in a retry loop), so there is no possibility of another process overwriting q before p gets to do so. The size of the *collision* array does not affect the correctness of the algorithm, but will affect its performance, and may be adjusted dynamically to optimise performance under varying workloads [6]. We leave unspecified how the value of *pos* is selected, and assume just that it is in the correct range.

We can now state precisely the relationship between this representation and that used in the previous section. A request (op, v) is associated with process p in N if $opInfos[p] = (op, n)$ and $n.val = v$. The request is waiting if $op = PUSH$ and $v \neq \bot$ or $op = POP$ and $v = \bot$, and p occurs in *collision*; otherwise it is completed. Initially, $opInfos[p].op = NONE$ for all p, and *collision* contains arbitrary values — for example, all elements of *collision* may initially be the same.

```
$push^{10}$(in $x : T$) $\widehat{=}$
    var $n$ : pointer to $Node$ ;
    $n :=$ new $Node()$ ;
    $n.val := x$ ;
    do
            $tryPush^{10}(n)$
        [] $tryEliminate^{10}(PUSH, n)$
    od

$pop^{10}$(out $y : T_\bot$) $\widehat{=}$
    var $n$ : pointer to $Node := null$ ;
    do
            $tryPop^{10}(n)$
        [] $tryEliminate^{10}(POP, n)$
    od ;
    $y :=$ if $n = null$ then $\bot$ else $n.val$ fi
```

Fig. 11. Elimination stack implementation (i)

We finally implement *tryEliminate* as shown in Figure 12, where M is the size of the *collision* array, and *opp* is a function defined so that $opp(PUSH) = POP$, $opp(POP) = PUSH$ and $opp(NONE) = NONE$.

[9] This is convenient since the *push* operation has already constructed this *Node* value. However, it also creates a linkage between the implementation of the elimination mechanism and the underlying stack. Thus, if we want to use the elimination mechanism with a stack implementation using a different data structure, we would need to modify this part of the elimination mechanism.

```
type $OP = PUSH \mid POP \mid NONE$

type $Request = \{op : OP$ ; $node :$ pointer to $Node\}$

var $opInfos$ : array $[ProcId]$ of $Request$

var $collision$ : array $[1 \mathinner{..} M]$ of $ProcId$

initially $\forall\, p : ProcId \bullet opInfos[p] = (NONE, null) \wedge$
          $\forall\, i : 1 \mathinner{..} M \bullet collision[i] \in 1 \mathinner{..} M$

$tryEliminate_p^{10}$(in $op : OP$ ; in out $n$ : pointer to $Node$) $r : bool \mathrel{\widehat{=}}$
   var $pos : int$ ; $q : ProcId$ ; $pinfo, qinfo : Request$ ;
   $pinfo := (op, n)$ ;
   $opInfos[p] := pinfo$ ;
   $pos : [\, pos \in 1 \mathinner{..} M \,]$ ;
   do
       $q := collision[pos]$ ;
       $CAS(collision[pos], q, p)$
   od ;
   $qinfo := opInfos[q]$ ;
   if $qinfo.op = opp(op)$ then
       if $CAS(opInfos[p], pinfo, (NONE, n))$ then                    (CAS1)
           if $CAS(opInfos[q], qinfo, (NONE, n))$ then                (CAS2)
               $n := qinfo.node$ ;
               $r := true$                                            (1)
           else
               $r := false$                                           (2)
           fi
       else
           $n := opInfos[p].node$ ;
           $r := true$                                                (3)
       fi
   else
       $delay$ ;
       if $\neg\, CAS(opInfos[p], pinfo, (NONE, n))$ then              (CAS3)
           $n := opInfos[p].node$ ;
           $r := true$                                                (4)
       else
           $r := false$                                               (5)
       fi
   fi
```

Fig. 12. Elimination stack implementation (ii)

To show that this is a valid refinement, we show that any trace of $STACK^{10}$ can be transformed into an equivalent trace of $STACK^{9}$. We first observe that the first few statements of *tryEliminate*, after some preliminary setup, implement $TE1$ (Figure 10), adding a request to N, and $TE2$, selecting a potential elimination partner.[10] Next, we consider the various execution paths that can assign a value to r (these assignments are numbered in Figure 12).

[10] We are glossing over some details here, which we do not have space to explain further — for example, a new request will become visible to other processes either when p stores its request in $opInfos[p]$ or when it writes its process id into *collision*, and overwriting q in *collision* may prevent any other process from selecting it as an elimination partner.

- Path (1) corresponds to the second disjunct of $TE4$, where p performs a successful active elimination.
- Path (2) corresponds to the third disjunct of $TE4$, where p's active elimination attempt fails because q's request has been fulfilled since it was checked at $TE3$.
- Path (3) corresponds to the first disjunct of $TE4$, where p performs a successful passive elimination, detecting the successful elimination just as it is about to attempt an active elimination.
- Path (4) corresponds to the first disjunct of $TE5$, where p performs a successful passive elimination, detecting the successful elimination abandoning its active elimination attempted and waiting.
- Path (5) corresponds to either the second or third disjunct of $TE5$, where p's passive elimination attempt fails because its request is unfulfilled or has been removed from the noticeboard.

As signalled earlier, to show that the implementation is lock-free, we need to show that an operation cannot continually try to eliminate without ever trying to perform its operation on the stack, since *tryPushElim* and *tryPopElim* can fail without another stack operation being completed. To do this, we need to either assume that the nondeterministic choice in *push* and *pop* is implemented as a fair choice, or replace it with an **if** statement which invokes a function to decide which alternative to choose (as is done in [13]) which must then be shown to have the desired property.

5 Conclusions

We have shown how a sophisticated concurrent stack implementation can be derived from an abstract specification in a series of verifiable steps. Although we have not given formal proofs for these steps, and have glossed over some details, we have stated the correctness conditions that need to be established and given what we hope are convincing informal arguments.

In doing this, we have provided an abstract description of the elimination mechanism which makes it easier to describe our algorithm, and to compare it with that in [6]. We have restructured their algorithm in a way that we believe makes it easier to understand — we did that before attempting the derivation presented here, and it is pleasing to see that the same structure could be arrived at in this derivation. This structure also allowed the algorithm to be simplified — for example, handling elimination for *push* and *pop* in a uniform way. More importantly, we have avoided an ABA problem by using a simpler data structure. In [6], *opInfos* is an array of pointers to *Request* nodes (which they call *ThreadInfo*), and they remove p's request from the noticeboard by setting the $opInfos[p]$ to *null*. Since these nodes must then be allocated dynamically, their algorithm is susceptible to another form of the ABA problem, which they do not mention explictly. Adding modification counts in this case proved to be more complex than for the pointers used in the stack representation, and we instead avoided the problem by making *opInfos* an array of *Request* nodes and allowing the *op* field to take on a third value (*NONE*).

We have previously verified this algorithm (for the three different memory management regimes discussed in Section 3) using simulation between IOAs [3]. That proof also involved showing that any trace of the implementation can be transformed into an equivalent trace of an abstract specification. The main differences between that proof and the one outlined here are: that we did the simulation proof in just two steps (one for the central stack and one for the elimination mechanism); and, more importantly, that while a simulation proof between a concrete machine C and an abstract machine A translates one step of C at a time (i.e. it uses induction on the length of the concrete execution), the proof here translates the entire execution of an operation of A at a time (i.e. it uses induction on the number of completed operations).

In future work, we intend to complete the formal proof of the derivation presented here and mechanise it using PVS, apply this derivation approach to other concurrent algorithms, such as those described in [11,2], explore other implementations that can be derived from our abstract model, and attempt to use our abstract model to derive elimination algorithms for other data structures, such as the queue algorithm described in [13].

Acknowledgements

We are grateful to Sun Microsystems Laboratories for financial support, and to Mark Moir for helpful discussion relating to this work.

References

[1] Back, R.-J. and J. von Wright, *Trace refinement of action systems*, in: *International Conference on Concurrency Theory*, 1994, pp. 367–384.

[2] Colvin, R. and L. Groves, *Formal verification of an array-based nonblocking queue*, in: *ICECCS '05: Proceedings of the Internation Conference on Engineering of Complex Computer Systems* (2005), pp. 92–101.

[3] Colvin, R. and L. Groves, *Verification of a scalable lock-free stack algorithm*, Computer Science Technical Report CS-TR-06-14, Victoria University of Wellington (2006).
URL http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-06-14.abs.html

[4] Crow, J., S. Owre, J. Rushby, N. Shankar and M. Srivas, *A tutorial introduction to PVS*, in: *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, 1995.
URL http://www.csl.sri.com/papers/wift-tutorial/

[5] de Roever, W.-P. and K. Engelhardt, "Data Refinement Model-Oriented Proof methods and their Comparison," Cambridge University Press, 1998, (With the assistance of J. Coenen and K.-H. Buth and P. Gardiner and Y. Lakhnech and F. Stomp).

[6] Hendler, D., N. Shavit and L. Yerushalmi, *A scalable lock-free stack algorithm*, in: *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain*, 2004, pp. 206–215.

[7] Herlihy, M., V. Luchangco and M. Moir, *Obstruction-free synchronization: Double-ended queues as an example*, in: *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems* (2003), p. 522.

[8] Herlihy, M. P. and J. M. Wing, *Linearizability: a correctness condition for concurrent objects*, TOPLAS **12** (1990), pp. 463–492.

[9] Lynch, N. A., "Distributed Algorithms," Morgan Kaufmann, 1996.

[10] Lynch, N. A. and F. W. Vaandrager, *Forward and backward simulations – part I: untimed systems.*, in: *135*, Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 1993 p. 35.

[11] Michael, M. M. and M. L. Scott, *Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors*, J. Parallel Distrib. Comput. **51** (1998), pp. 1–26.

[12] Moir, M., *Practical implementations of non-blocking synchronization primitives*, in: *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, Santa Barbara, CA.*, 1997, pp. 219–228.

[13] Moir, M., D. Nussbaum, O. Shalev and N. Shavit, *Using elimination to implement scalable and lock-free fifo queues*, in: *Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005)* (2005), pp. 253–262.

[14] Morgan, C., "Programming from Specifications," Prentice Hall, 1994, second edition.

[15] R.K.Treiber, *Systems Programming: Coping with Parallelism. RJ5118*, Technical report, IBM Almaden Research Center (1986).
URL http://domino.watson.ibm.com/library/CyberDig.nsf/Home

[16] Spivey, M., "The Z Notation: A Reference Manual," Prentice-Hall International, 1988, second edition, 1992.