DOI 10.1007/s00165-008-0092-5
BCS © 2008
Formal Aspects of Computing (2009) 21: 187–223

Formal Aspects of Computing

Trace-based derivation of a scalable lock-free stack algorithm

Lindsay Groves[1] and Robert Colvin[2]

[1] School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand. E-mail: lindsay@mcs.vuw.ac.nz
[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 based on the scalable stack algorithm of Hendler et al. (Proceedings of the sixteenth annual ACM symposium on parallel algorithms, 27–30 June 2004, Barcelona, Spain, pp 206–215), which allows push and pop operations to be paired off and eliminated without affecting the central stack, thus reducing contention on the stack, and allowing multiple pairs of push and pop operations to be performed in parallel. Our algorithm uses a simpler data structure than Hendler, Shavit and Yerushalmi's, and avoids an ABA problem. We first derive a simple lock-free stack algorithm using a linked-list implementation, and discuss issues related to memory management and the ABA problem. We then add an abstract model of the elimination process, from which we derive our elimination algorithm. This 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. Each step in the derivation is either a data refinement which preserves the level of atomicity, an operational refinement which may alter the level of atomicity, or a refactoring step which alters the structure of the system resulting from the preceding derivation. We verify our refinements using an extension of Lipton's reduction method, allowing concurrent and non-concurrent aspects to be considered separately.

Keywords: Refinement; Derivation; Lock-free algorithms; Concurrency; Stack; Elimination; Reduction, Linearisability; Atomicity

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. This is illustrated clearly in the case of concurrent implementations of stacks. Michael and Scott [MS98] describe a simple linked-list implementation of a concurrent stack, attributed to Treiber [Tre86], which relies only on a compare and swap (CAS) instruction to synchronise access to the stack. Hendler et al. [HSY04] extend this implementation by allowing processes that detect interference while accessing the stack to pair off so that push and pop operations can "eliminate" each other, leaving the central stack unchanged. The

Correspondence and offprint requests to: L. Groves, E-mail: lindsay@mcs.vuw.ac.nz

resulting implementation is scalable, since under high workloads, multiple pairs of push and pop operations can be performed in parallel without increasing contention on the central stack.

Although the basic idea underlying this elimination mechanism is quite simple, the description given in [HSY04] 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 [CG07], we have verified a simplified version of Hendler, Shavit and Yerushalmi's algorithm which we discovered while attempting to verify their algorithm. That proof was based on simulation between input–output automata (IOAs) [Lyn96, LV95] and is fully mechanised in PVS [COR+95], but it too is encumbered with low-level details, related more to the IOA formalism than to the algorithm being verified.

Our aim in this paper is to give a more illuminating presentation of the elimination stack algorithm, showing how our simplified version can be derived 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 the algorithm is lock-free. *Linearisability* [HW90] 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, and is thus equivalent to a legal sequential execution. *Lock-freedom* is a progress 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.[1]

By taking a constructive approach, we are able to clearly separate essential ideas from implementation detail, and identify points at which alternative implementation approaches might be considered, leading to a deeper understanding of the algorithm. We begin with a highly non-deterministic model, which allows us to focus on the essential behaviour of the algorithm, and gradually introduce the algorithmic decisions that ensure that the algorithm can be implemented in a lock-free manner on the intended architecture. We make a clear separation between the underlying stack implementation and the elimination mechanism, allowing elimination to be used with any concurrent stack implementation. We also use a simpler data structure for the elimination mechanism than the one in [HSY04], and thereby avoid an ABA problem which arises with their implementation.

Whereas formalisms such as IOAs and Action Systems [BKS88] require algorithms to be expressed as sets of actions or transitions, our derivation works with algorithmic descriptions in a more familiar procedural form, making it easier to relate correctness conditions to the original algorithm. Our derivation approach separates reasoning about the sequential behaviour of each process, which is based on standard refinement techniques [Mor94, BvW98, dRE98], and the interaction between processes, which uses a trace reduction approach similar to that of Lipton [Lip75] and Lamport and Schneider [LS89]. This approach appears to lead to proofs that are closer to the way we think about lock-free algorithms, and to give greater insight into why they are correct, than simulation proofs. To keep the paper to a reasonable length, we take a semi-formal approach, describing each model formally, giving semi-formal justifications for the correctness of each step, and identifying correctness conditions that would need to be verified in order to obtain a more formal proof.

We begin in Sect. 2 by outlining the language we use and our derivation approach. In Sect. 3, we present an abstract specification for a concurrent stack and take an initial step towards refining this to a lock-free stack implementation, and then in Sect. 4 we refine this to a simplified version of Treiber's algorithm using a linked list representation, and consider issues to do with reuse of heap space, including the ABA problem—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 5 then extends the simple stack implementation by adding the elimination mechanism, starting with an abstract specification of elimination and deriving its implementation. Section 6 presents our conclusions, and discusses related and future work.

This paper is a revised and extended version of [GC07]. In this version, our derivation approach is explained in more detail, the derivations are given in more detail, and a simpler abstract model of the elimination mechanism is used in Sect. 5.

[1] Early papers often called this property *non-blocking*. Current usage treats "non-blocking" as a more general term encompassing other progress properties such as wait-freedom [Her91] and obstruction-freedom [HLM03]. This should not be confused with the (non)blocking interpretation of preconditions, as used for example in [DB03].

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

Fig. 1. Specification of CAS

2. Preliminaries

We begin by describing the basic features of the language we use for our derivation—some other notation will be introduced later as needed. We then outline the basis for the approach taken in our derivation.

2.1. A wide-spectrum language

We wish to start with an abstract specification of a concurrent stack, and derive a lock-free algorithm that can be executed on a typical modern architecture which supports atomic read and write operations on shared variables, and also allows shared variables to be conditionally updated using a CAS instruction. To do this, we express our algorithms in a wide-spectrum language based on the refinement calculus [Mor94, MV93, BvW98], with a number of extensions and variations.

Our stack specification and implementation are assumed to be packaged as modules (e.g. as in [Mor94, Chap. 16]) which export procedures implementing the stack operations and encapsulate the variables used to represent the stack so that they can only be modified by the procedures declared in that module. Thus, we can regard the stack as a closed system where we know what actions can be performed by other processes.

We assume procedures and type declarations essentially as in [Mor94], but we use **in**, **out** and **in out** to describe parameter modes, rather than Morgan's more verbose **value**, **result** and **value result**; for example, see Fig. 1. As in Morgan, parameters are treated as local variables within the procedure, whose initial value is provided in the call (in the case of **in** and **in out**) and whose final value is copied to the caller at the end of execution of the procedure body (in the case of **out** and **in out**). Call by reference parameters could be used instead, but may introduce the possibility of processes being able to see intermediate states that should not be observable if shared variables are passed as parameters.

We also use value-returning procedures and name the return value so that it can be constrained in specification statements. It is straightforward to transform a value-returning procedure into an equivalent procedure with an additional **out** parameter, so we can reason about value-returning procedures in the same was as non-value-returning procedures. For example, in the definition of CAS in Fig. 1, r can be declared as an **out** parameter, and an assignment $r := CAS(c, a, b)$, as in the example in Sect. 2.2, can be replaced by $CAS(c, a, b, r)$. Similarly, an **if** statement of the form **if** $CAS(c, a, b)$ **then** S **else** T **fi** (as in Figs. 22, 24) can be replaced by $CAS(c, a, b, r)$**;** **if** r **then** S **else** T **fi**, where r is a new local Boolean variable. We will see in Sect. 2.3 why these transformations are safe.

We use specification statements of the form $w : [\, P/R \,]$ with their usual meaning (postcondition R must be established, altering only variables in the frame w, assuming that precondition P holds initially), and write x_0 in the postcondition to refer to the initial value of a variable x. All statements are required to preserve the relevant state invariant, however, we usually leave invariants in preconditions implicit and write $w : [\, R \,]$ when there is no precondition other than the state invariant (cf. [Mor94]). Parts of a postcondition that only refer to initial variables act as guards, and if they fail cause the program to block (i.e. the statement is not enabled). For example, $c : [\, c = c_0 + 1 \,]$ specifies an operation that increments a variable c, and is always enabled, while $c : [\, c = c_0 + 1 \wedge c_0 > 0 \,]$ specifies an operation that decrements c provided that it is positive, and is disabled when c is not positive. Since blocking must be avoided in lock-free algorithms, in a demonic choice (written as []) we must always ensure that at least one branch will be enabled whenever the choice is executed. The only non-determinism in our language is demonic, so all programs are conjunctive.

In writing specifications and invariants, we mostly use Z's mathematical notation [Spi92]; in particular, seq T and iseq T are the sets of finite sequences and injective finite sequences (i.e. sequences containing no repeated elements) over T, $\#s$ is the length of the finite sequence s, $A \nrightarrow B$ is the set of partial functions from A to B, 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

$$
\begin{array}{ll}
inc() \mathrel{\widehat{=}} & tryInc()\ r : bool \mathrel{\widehat{=}} \\
\quad \textbf{do} & \quad \textbf{var}\ a, b : int\ ;\ r : bool\ ; \\
\quad\quad tryInc() & \quad a := c\ ; \qquad (C1) \\
\quad \textbf{od} & \quad b := a + 1\ ; \qquad (C2) \\
 & \quad r := CAS(c, a, b) \qquad (C3)
\end{array}
$$

Fig. 2. A lock-free algorithm to increment a shared counter

is y, $s \lhd f$ is the function obtained from f by restricting its domain to elements of set s, and $s \lhd\!\!\!- f$ is the function obtained from f by restricting its domain to elements not in set s. We also use labelled tuple types, of the form $(f_1 : T_1\ ;\ \cdots\ ;\ f_n : T_n)$, and access their components using the field names (f_i) as selectors; so if $r = (v_1, \ldots, v_n)$ is a tuple of the aforementioned type, then $r.f_i = v_i$.

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, where assignment statements, unrefined specification statements, and tests are assumed to be atomic. See, for example, the abstract stack specification given in Fig. 3.

We assume a trace semantics similar to that used in [BvW94], except that (following [Lyn96]) 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 actions in an execution, rather than a sequence of states. We take this approach so that we can define refinement in terms of sequences of actions, as it is for IOAs [Lyn96, LV93], which is consistent with the way that linearisabiity is defined in [HW90].

2.2. The target language

As usual, our target language is an executable subset of the wide-spectrum language—with the additional requirement that assignments and tests can reasonably be treated as atomic at the hardware level. To this end, we ensure that assignments and tests contain at most one reference to a shared variable. Also, we use initialisations in declarations only where variables are initialised to constants.

We use **if-then-else** statements (as in [BvW98]) rather than Dijkstra's guarded commands (as in [Mor94]), as this suits the style of algorithm we are deriving. Since most of our loops involve repeating some action until it succeeds, we write $\textbf{do}\ A\ \textbf{od}$ as an abbreviation for $|[\textbf{var}\ r : bool := false\ ;\ \textbf{do}\ \neg\ r \rightarrow r := A\ \textbf{od}]|$, where A assigns a value to a Boolean variable r; for example, see Fig. 2. We use similar abbreviations when the loop body is a more complex statement assigning to r. Thus, $A\ []\ B$ in the body of such a loop is an abbreviation for $r := A\ []\ r := B$, and $A\ ;\ B$ is short for $A\ ;\ r := B$. Again, we will see in Sect. 2.3 why this is safe.

The only synchronisation mechanism used is a CAS 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 an arbitrary type T, in Fig. 1.

Compare and swap is typically used to update a shared variable, provided that it has the same value that it was observed to have at some earlier point. For example, the $tryInc$ operation in Fig. 2 will update a shared integer counter c provided it has the same value when the CAS is executed at $C3$ as it had when its value was read at $C1$, and set r to indicate whether the update was successful. We can obtain a lock-free algorithm to increment c by repeating this piece of code until r is $true$, as illustrated in the inc operation in Fig. 2.

In practice, at hardware level, CAS is untyped and just operates on words or double words. Thus, to obtain a practical implementation, we have to ensure that CAS is only used on data types that can be represented using single or double words. For example, it is common to assume that a CAS can operate on integers and pointers, but not on arbitrary record types. We will discuss this issue further in Sects. 4.4 and 5.7.

When a CAS is used to update an array element, the specification must show the effect on the entire array. Thus, $r := CAS(A[k], old, new)$ is equivalent to the specification:

$$
A, r : \left[\begin{array}{l} A_0[k] = old \wedge A = A_0 \oplus \{k \mapsto new\} \wedge r = true \vee \\ A_0[k] \neq old \wedge A = A_0 \wedge r = false \end{array} \right]
$$

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

Fig. 3. Abstract specification for a system involving a concurrent stack

$$\textbf{type } Stack \mathrel{\widehat{=}} \text{seq } T \qquad \textbf{const } EmptyStack \mathrel{\widehat{=}} \langle\rangle$$

$$push^1(\textbf{in } x : T) \mathrel{\widehat{=}} \quad s : \left[s = \langle x \rangle \frown s_0 \right]$$

$$pop^1(\textbf{out } y : T_\perp) \mathrel{\widehat{=}} \quad s, y : \left[\begin{array}{l} s = s_0 = \langle\rangle \wedge y = \perp \ \vee \\ s_0 = \langle y \rangle \frown s \end{array} \right]$$

Fig. 4. Abstract specification for stack operations

2.3. Derivation approach

The informal notion of linearisability of a concurrent object $\mathcal{O}$ (such as a shared data structure) introduced in Sect. 1 is formalised in [HW90] by adding invocation and response actions marking the beginning and end of each execution of an operation on $\mathcal{O}$. A *history* is defined as the sequence of invocation and response actions occurring in some execution, and is *well-formed* if each process alternately performs matching invocations and responses (so a process only ever performs one operation on the object at a time), *sequential* if every response is immediately preceded by a matching invocation by the same process (and *concurrent* otherwise), and *legal* if each matching invocation–response pair satisfies the sequential semantics for $\mathcal{O}$. An implementation of $\mathcal{O}$ is then defined to be *linearisable* if for every concurrent history there is an equivalent well-formed legal sequential history which preserves the order of non-concurrent operations (i.e. if operation op_1 ends before operation op_2 begins in the concurrent history, this ordering is preserved in the sequential one).

A common way to demonstrate linearisability is to identify, for each operation on the concurrent object, a point, called a *linearisation point*, at which that operation can be understood to take place. We can then construct the required sequential history by inserting the required invocation–response pair at the linearisation point. In many cases, the linearisation point is a step in the code for that operation at which a shared data structure is updated or read (in the case of operations that do not alter the data structure). In some cases, however, the linearisation point for one operation may be a step of another process—indeed, this is the case for the elimination stack algorithm.

We will derive our stack algorithms by constructing a sequence of models of a system in which a finite set of processes operate on a shared stack. The atomic actions for a given model are the tests, assignments, and unrefined specification statements. In the initial model, each stack operation is an atomic action (see Figs. 3, 4). Subsequent models each add more implementation detail, until we arrive at the final implementation model, in which each test and assignment can be implemented in hardware as an atomic action. Typically, each successive model will provide implementations of operations that were treated as atomic in the previous model, sometimes introducing new data structures for this purpose, and in some cases, the new model redefines operations that have already been defined in the previous model.

The initial model is designed to be trivially linearisable, since each of its executions can be mapped directly to an equivalent well-formed legal sequential history by replacing each atomic action by the obvious invocation–response pair. We then wish to show that each subsequent model preserves linearisability, and thus that the final implementation is linearisable. Since the initial model already requires that each operation should appear to take place atomically and be correct with respect to the sequential semantics, we only need to ensure that in all executions of subsequent models, each process performs the same sequence of stack operations and that the order of non-concurrent operations is preserved.

Since linearisability is a safety property, we only need to consider finite executions (or prefixes of executions), and do not need to consider termination. Also, we only need to consider executions consisting of completed executions of the operations used in the current model ($push$ and pop in the initial model). Incomplete executions

can be handled in standard ways in which they are either completed or discarded, according to whether their linearisation point is present in the history, as described in [HW90]. We therefore ignore incomplete operations in the rest of this paper.

Thus, we say that a model M is *refined by* model M', or that M' is a *valid refinement* of M, if for every finite execution of M', there is an equivalent execution of M in which each process performs the same sequence of stack operations and the order of non-concurrent operations is preserved.

When we are discussing the abstract model, two executions are equivalent if, when started in the same state, they can end in the same set of final states. When discussing more concrete models, we regard two executions as equivalent if the final states are equivalent under a suitable abstraction relation; i.e. they represent the same abstract state.

Our derivations involve three kinds of transformation between successive models: data refinement, which introduces a more concrete data representation; algorithmic refinement, which refines one or more specification statements to a more concrete form; and refactoring, which alters the structure of the system by changing the interface to a procedure, combining two procedures into one, or moving an operation out of a loop. Some refactoring steps also modify data representations, so refactoring steps use a combination of the techniques used for reasoning about data refinement and algorithmic refinement steps.

Data refinement is performed in such a way that each action on the abstract type is replaced by a single action on the concrete type. This means that data refinement can be performed using standard data refinement techniques for sequential programs (e.g. [Mor94, MV93, BvW00, dRE98]), without considering the effects of concurrency. We use the following definition, from [MG90]:

Program P is *data refined* by program P', using abstraction invariant AI, abstract variables a, and concrete variables c, if for any formula ψ not containing free occurrences of c, we have:

$$(\exists a \bullet AI \wedge wp(P, \psi)) \Rightarrow wp(P', \exists a \bullet AI \wedge \psi)$$

Since all of our programs have the same structure (as shown in Fig. 3), this usually amounts to showing that each operation specified in P is correctly implemented (i.e. refined) in P', and implies refinement in the sense defined above.

Algorithmic refinement is typically more complex because it introduces the possibility of interference between processes. We perform these refinement steps in two parts. We first refine the specifications to their more concrete form without formally considering interference, so this can be done using the standard refinement calculus for sequential programs, and we generally do not justify these steps formally. Next, we show that the refined version works correctly in the presence of concurrency. This is usually done by showing that any concurrent execution of the resulting system can be transformed into an equivalent execution in which the actions of the refined operation are executed without interruption—following [LS89, FQ03, FQ05, FF04, WS06] and others, we call this property *atomicity* (note, however, that some authors, e.g. [Lyn96], use atomicity as a synonym for linearisability). The sequential refinement result then implies that this is equivalent to an execution of the initial system. Of course, we know that many of the rules of the sequential refinement calculus, such as introducing sequential composition, will not always work in a concurrent setting, and we take potential interference into account when we apply them, just as we take performance into account in an informal way in deriving sequential programs.

In most cases, the equivalent sequential execution is obtained by using commutativity properties to rearrange the actions of the concurrent execution. As a simple example, suppose we have a system S, in which operation A is treated as atomic, and we wish to refine this to a system S', in which A is implemented as a sequential composition B ; C, where B and C are regarded as atomic. We first show that $A \sqsubseteq B$; C holds in the sequential refinement calculus. We then consider the effects of concurrency. An execution of S' in which (the implementation of) A is executed by a process p, will have a trace t of the form $\alpha\ B_p\ \beta\ C_p\ \gamma$, where α, β and γ are sequences of actions of S' and β does not contain any actions performed by p (which we call "p-actions"). We thus show that any execution with this trace can be transformed into an equivalent execution in which B_p and C_p are executed without interruption. We typically do this by showing that B_p can move right over steps of other processes, so we obtain an execution whose trace is $\alpha\ B_p\ C_p\ \beta\ \gamma$, or that C_p can move left over steps of other processes, giving an execution whose trace is $\alpha\ \beta\ B_p\ C_p\ \gamma$. Since we know that $A \sqsubseteq B$; C, it follows that there is an equivalent execution with trace $\alpha\ A_p\ \beta\ \gamma$ or $\alpha\ \beta\ A_p\ \gamma$. Repeating this transformation for all completed executions of A, we can transform any trace of S' into an equivalent trace of S. Also note that the position of A_p in the transformed trace lies between the first and last actions (inclusive) of the implementation of A, which ensures that the order of non-concurrent operations is preserved. It thus follows that S' is linearisable with respect to S.

The main technique used in this approach is to show that certain atomic actions in the body of a procedure commute (either left, or right, or in both directions) with atomic actions that may be performed by other processes. An action ϕ *right commutes* with ψ, and ψ *left commutes* with ϕ, if for any sequences of actions α and β (executions with), traces $\alpha\ \phi\ \psi\ \beta$ and $\alpha\ \psi\ \phi\ \beta$ are equivalent. An action that left (right) commutes with all actions of the current model is called a *left* (*right*) *mover*; an action which is both a left mover and a right mover is called a *both mover*. If a given execution of an operation Op consists of atomic actions $a_1, \ldots, a_n$ where, for some k with $1 \leq k \leq n$, $a_1, \ldots, a_{k-1}$ are right movers and $a_{k+1}, \ldots, a_n$ are right movers, then this execution is equivalent to an execution in which actions $a_1, \ldots, a_n$ are executed without interruption, and a_k can be taken as the linearisation point.

In reasoning about commutativity of actions, we can use general properties that hold for all programs, such as:

- An action that only accesses local variables of process p, or heap locations to which p holds a unique pointer, commutes in both directions with any action of another process.
- An action that reads a shared variable commutes in both directions with any action that does not assign to that variable.
- An action that assigns to a shared variable commutes in both directions with any action that does not refer to that variable.

For example, in the shared counter in Fig. 2, $b := a + 1$ is a local action and can always be moved over actions of other processes so that it is executed immediately before the CAS; for this reason we can omit the assignment to b and write $C3$ as $r := CAS(c, a, a + 1)$, even though we know that this is not actually an atomic hardware action. Similarly, our treatment of value returning procedures and our convention of ignoring loop exit tests can be justified in the same way, because they only involve a local variable, r.

We may also appeal to assumptions about the effects that other processes may have, for example that they preserve state invariants or that they can only modify shared variables in particular ways. In the stack implementation, we are able to verify such properties because we know that the only actions that can affect the variables used in the stack implementation are other stack operations.

The most interesting cases involve reasoning about CAS operations, where we will use the outcome of the CAS and other assumptions about the possible effects of other processes to show that a CAS can commute with other actions. Consider, for example, the shared counter in Fig. 2. If we know that the counter is only ever increased, then if the CAS succeeds, we can infer that the counter has not been modified since it was read in the first assignment. We can therefore obtain an equivalent trace in which the three assignments are executed without interruption, by moving the first two assignments right over any intervening actions to the position of the third assignment. That is, if there is a trace $\alpha\ C1_p\ \beta\ C2_p\ \gamma\ C3S_p\ \delta$ in which $C3$ succeeds, then this trace is equivalent to $\alpha\ \beta\ \gamma\ C1_p\ C2_p\ C3S_p\ \delta$.[2]

If the CAS fails, however, we know that the counter has been modified since it was read in the first assignment, so we cannot move the first two assignments right. Nor can we move the CAS left over the actions that modified c, since then the CAS will no longer fail, so we will not get an equivalent trace. In this case we observe that the actions of a failed attempt to update the counter have no observable effect, and can be deleted. Thus, in an operation that repeatedly attempts to update the counter until it succeeds in doing so without interference, only the final, successful attempt needs to be considered in constructing an equivalent sequential execution. That is, a trace of the form $\alpha_1 C1_p \beta_1 C2_p \gamma_1 C3F_p \ldots \alpha_n C1_p \beta_n C2_p \gamma_n C3F_p \alpha_{n+1} C1_p \beta_{n+1} C2_p \gamma_{n+1} C3S_p \delta$, for $n \geq 0$, is equivalent to $\alpha_1 \beta_1 \gamma_1 \ldots \alpha_n \beta_n \gamma_n \alpha_{n+1} C1_p \beta_{n+1} C2_p \gamma_{n+1} C3S_p \delta$.

In some cases, we are not able to rearrange the steps of an operation so that they are performed without interruption; for example, because it is essential that a step of another process occurs between two steps of the operation. In this case, we are still able to apply the trace–reduction approach by showing that a trace containing a particular pattern is equivalent to one containing the corresponding abstract operation.

Our trace-reduction approach is similar to the "reduction" method described in [Lip75, LS89], except that where they are concerned with showing that interference does not occur, and provide reductions that apply for all executions, we show that interference does not lead to incorrect behaviour, and provide reductions that work for various classes of executions, according to the outcomes of CAS operations and other tests. Also, in some

[2] When describing actions in traces, we append "S" or "F" to indicate whether a test succeeded or failed. For this purpose, an assignment such as $C3$, which assigns a Boolean variable to be tested subsequently, is treated as a test since the test is not shown explicitly. Thus, $C3S_p$ here denotes a successful execution of $C3$ by process p.

cases, we cannot simply rearrange the steps of the original execution, but may need to insert, delete or modify steps to obtain the required execution. This form of reduction is discussed in more detail in [Gro07], where it is used to verify a shared counter and the stack algorithm shown in Fig. 10.

This approach is also reminiscent of Owicki–Gries logic [OG76], where "local correctness" and "global correctness" (freedom from interference) are checked separately, however, we are not attempting to show that there is no interference, but rather that any interference that occurs does not lead to incorrect behaviour. Our approach also has something of the flavour of rely guarantee reasoning [XdRH97], since in proving trace reductions, we may make assumptions (like rely conditions) about the behaviours of other processes that can occur between the steps of the operation in question, which must in turn satisfy the assumptions made in designing other operations (like guarantee conditions).

3. An abstract concurrent stack

In this section, we present an abstract specification for a linearisable concurrent stack. We then take an initial step towards refining this specification to a lock-free stack implementation by allowing stack operations to fail, and introducing loops which repeatedly attempt to perform a stack operation until it is performed successfully. We introduce this refinement before introducing the data representation (see Sect. 4.1) because this pattern is common to a large number of lock-free algorithms, and it allows us to illustrate our derivation approach in a simple setting. It also forms the basis for the elimination algorithm discussed in Sect. 5, which is (mostly) independent of the representation used for the stack.

3.1. Specifying an abstract concurrent stack ($STACK^1$)

We wish to consider a system consisting of a finite set, $\mathcal{P}$, 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. Processes regard stack operations as being atomic and wait for each stack operation to complete before proceeding with any other action—in particular, a process will never invoke another stack operation before its previous stack operation is completed. In practical terms, a stack operation can be thought of as being executed in the same thread of control as its calling process.

We can model such a system by abstracting away from its other behaviour and just considering its stack operations. Thus, we consider a system in which each process performs a non-deterministically chosen sequence of stack operations, as shown in Fig. 3, where 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. Clearly, this program can generate any sequence of stack operations that can occur in any application. 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 sequence of stack operations is considered to constitute the observable behaviour of the program.[3]

All of our programs will have this high level structure, but will use different versions of $push$ and pop, and may use different data structures to represent the stack. We use superscripts (e.g. $push^1(x)$, $pop^2(y)$) 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 in a given model are assumed to be defined as in the previous model, but use new components from the current model. For example, in Fig. 7, $push^3$ 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, $STACK^1$, we treat an abstract stack as a sequence of values of its component type, and we define the stack operations as shown in Fig. 4. Since we wish to implement a lock-free stack, a pop on an empty stack (described in the first disjunct) cannot wait for the stack to become non-empty, but instead returns a distinguished value, $\bot \notin T$, indicating that the stack was empty, and the result type for pop is $T_\bot \mathrel{\widehat{=}} T \cup \{\bot\}$. Note that in the second disjunct, the type of s implies $y \neq \bot$, so the specification entails $y = \bot \iff s_0 = \langle\rangle$.

[3] Alternatively, we could store values of x and take the sequence of values of x and y to be observable, or we could consider the pushed values to be inputs and the popped values to be outputs, and take inputs and outputs to be observable.

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

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

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

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

Fig. 5. Abstract specification for lock-free stack

In this model, *push* and *pop* actions are atomic, so a trace is a sequence of *push* and *pop* actions which is valid according to the semantics of these stack operations. This stack is clearly linearisable, since *push* and *pop* are already atomic, so we can show that subsequent versions are linearisable by showing that their traces are equivalent to traces of $STACK^1$.

3.2. Allowing operations to fail ($STACK^2$)

In any realistic implementation of a concurrent stack, a *push* or *pop* operation will require more than one access to a shared location. For example, in the linked list implementation described in Sect. 4, a stack operation needs 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 on the intended architecture, we need to consider the possible effects that other processes may have between when the top of stack pointer is read and when it is updated.

Rather than taking the traditional approach of preventing interference by using mutual exclusion mechanisms based on locks or semaphores, lock-free algorithms must operate correctly in the presence of interference. A common way to achieve this is to detect interference, using a CAS, and retry the operation. Following this approach, we will implement *push* and *pop* by repeatedly attempting to perform the operation until it is performed successfully. Thus, we define *push* and *pop* in terms of new operations *tryPush* and *tryPop*, as shown in Fig. 5, which attempt to perform a *push* or *pop*, respectively, and either "succeed"(returning *true*) or "fail" (returning *false*). Introducing *tryPush* and *tryPop* as separate operations is helpful for expository purposes—it is also important in developing the elimination algorithm in Sect. 5, since it allows a process attempting to perform a stack operation to chose an alternative action if the operation fails. At this stage we do not impose any progress requirement, so every invocation of *tryPush* or *tryPop* could fail—we will address the lock-freedom requirement in the way that subsequent refinements resolve non-determinism.

In this model, we regard *tryPush* and *tryPop* as being atomic, so a trace of $STACK^2$ is a sequence of these operations. As indicated in Sect. 2, we distinguish successful and failed occurrences of these operations by appending "S" and "F", respectively.[4]

To show that $STACK^2$ is a valid refinement of $STACK^1$, we show that for any trace t of $STACK^2$ there is an equivalent trace of $STACK^1$. Now, every trace of $STACK^2$ is produced by running a set of processes, each executing a sequence of stack operations, and each stack operation consists of a sequence of $tryPush^2$ and $tryPop^2$ actions, possibly interleaved with $tryPush^2$ and $tryPop^2$ actions performed by other processes. We will show, firstly, that every execution of $STACK^2$ is equivalent to one in which each $push^2$ and pop^2 operation is executed without interruption, and secondly, that the trace of this sequential execution is equivalent to a trace consisting of $push^1$ and pop^1 operations.

From the semantics of loops, any execution of $STACK^2$ containing a completed $push^2(x)$ operation by process p has a trace containing zero or more failed occurrences of $tryPush_p^2(x)$, followed by one successful occurrence, interleaved with actions of other processes; i.e. trace t has the form $\alpha_1\ tryPushF_p^2(x)\,\alpha_2\ \ldots\ tryPushF_p^2(x)$

[4] Recall that we are eliding the assignment and test on r, which we can do because r is a local variable. Thus, $tryPushS^2(x)$ is equivalent to $s, r : [s = \langle x \rangle \frown s_0 \wedge r = true]$, and $tryPushF^2(x)$ is equivalent to $s, r : [s = s_0 \wedge r = false]$.

$\alpha_n\ tryPushS_p^2(x)\,\alpha_{n+1}$, for some $n \geq 1$, where $\alpha_1, \ldots, \alpha_{n+1}$ are (possibly empty) sequences of $tryPush^2$ and $tryPop^2$ actions containing no p-actions, and any p-action in α_1 is part of a completed operation contained within α_1.[5]

The unsuccessful $tryPush$ operations have no effect on s, so they can be deleted. Thus, trace t is equivalent to $\alpha_1 \ldots \alpha_n\ tryPushS_p^2(x)\,\alpha_{n+1}$. But $tryPushS_p^2(x)$ is precisely the trace obtained when $push^2(x)$ is executed without interruption, so this demonstrates that $push^2$ is atomic. Moreover, $push(x)^2$ is equivalent to $push^1(x)$ in a sequential context, so we have sequential correctness. Thus, t is equivalent to $\alpha_1\,\alpha_2 \ldots \alpha_n\ push_p^1(x)\,\alpha_{n+1}$; 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 interruption.

Similarly, any execution of $STACK^2$ containing a completed $pop(y)^2$ operation by process p has a trace t of the form $\alpha_1\ tryPopF_p^2(y)\,\alpha_2 \ldots\ tryPopF_p^2(y)\,\alpha_n\ tryPopS_p^2(y)\,\alpha_{n+1}$, for some $n \geq 1$ and $\alpha_1, \ldots, \alpha_{n+1}$ as above. Again, the unsuccessful operations have no effect and can be deleted, so t is equivalent to $\alpha_1 \ldots \alpha_n\ tryPopS_p^2(y)\ \alpha_{n+1}$. But $tryPopS_p^2(x)$ is precisely the trace obtained when pop^2 is executed without interruption, so this demonstrates that pop^2 is atomic. Moreover, pop^2 is equivalent to pop^1 in a sequential context, so we have sequential correctness. Thus, t is equivalent to $\alpha_1\,\alpha_2 \ldots \alpha_n\ pop_p^1(x)\,\alpha_{n+1}$; i.e. the observable effect is the same as if activation of the pop had been delayed until after α_n and then completed without interruption.

We have shown that any execution containing a complete execution of an operation of $STACK^2$ (i.e. either $push^2$ or pop^2) is equivalent to an execution in which that operation is executed without interruption, and that sequential execution is equivalent to the corresponding operation of $STACK^1$. By induction on the number of completed stack operations, any trace of $STACK^2$ can be transformed into an equivalent trace in which each stack operation is executed without interruption, which is equivalent to a trace of $STACK^1$. Moreover, this reduction preserves the order of non-concurrent operations, as required for linearisability.

In showing linearisability, we can assume that each "try" operation chooses non-deterministically whether to succeed or fail; we can also ignore the issue of whether the loops in $push$ and pop terminate. At this stage, however, our algorithm is not lock-free because it is possible for $tryPush^2$ and $tryPop^2$ to always fail, in which case no operation would ever complete. In Sect. 4, we will refine these specifications so that a "try" operation only fails if it detects interference, which will then allow us to show that the resulting implementation is lock-free.

4. Deriving a simple lock-free stack

We now introduce a linked list data structure to represent the stack, and derive implementations of the stack operations using this representation. We first describe the data representation used, then data refine the specifications for $tryPush^2$ and $tryPop^2$ using this representation, and further refine these to implementations that can be executed on an architecture supporting atomic load, store and CAS operations. Our initial implementation does not recycle popped heap nodes. We then discuss memory management issues and perform a further data refinement which gives us an implementation which is identical to Treiber's lock-free stack, as described in [MS98].

4.1. Data representation ($STACK^3$)

To describe a linked list representation for a stack, we define two new types: Ptr, which models a set of pointers (or heap locations) containing a distinguished value called $null$; and $Node$, which models a set of labelled pairs (or records) each comprising a val field of type T and a $next$ field of type Ptr. We model the heap, as seen by the stack implementation, explicitly as a partial function h from non-null pointers (denoted by $NPtr$) to nodes, and use a pointer variable, Top, to record the top of the stack. The relevant declarations are shown in Fig. 6.

[5] In justifying subsequent refinements, we will usually assume these constraints without mentioning them explicitly. The constraint on α_1 ensures that we are reducing a complete execution of $tryPush_p^2$.

$$\textbf{type}\ Ptr \qquad\qquad \textbf{const}\ null : Ptr$$

$$\textbf{type}\ NPtr \mathrel{\widehat{=}} Ptr \setminus \{null\} \qquad\qquad \textbf{var}\ h : NPtr \pfun Node := \varnothing$$

$$\textbf{type}\ Node \mathrel{\widehat{=}} (val : T\ ;\ next : Ptr) \qquad\qquad \textbf{var}\ Top : Ptr := null$$

Fig. 6. Concrete data declarations

To be a valid representation of a stack, the linked list must be finite and contain no cycles. We express this as a state invariant, Inv, which postulates the existence of a finite sequence of unique pointers corresponding to Top followed by the values in the $next$ fields of the nodes in the linked list:

$$\begin{aligned} Inv(Top, h) \mathrel{\widehat{=}} \exists f : \mathrm{iseq}\ Ptr \bullet \\ f(1) = Top \land last(f) = null \land \\ (\forall i : 1 \mathinner{..} \#f - 1 \bullet f(i) \in \mathrm{dom}\, h \land h(f(i)).next = f(i+1)) \end{aligned}$$

We now define an abstraction relation, Abs, showing how the abstract and concrete states are related, in terms of a function, abs, which returns the abstract stack represented by the linked list and is well-defined provided that $Inv(Top, h)$ holds:[6]

$$Abs(s, Top, h) \mathrel{\widehat{=}} s = abs(Top, h) \land Inv(Top, h)$$

$$\begin{aligned} abs(Top, h) \mathrel{\widehat{=}} &\textbf{if}\ Top = null\ \textbf{then}\ \langle\rangle \\ &\textbf{else}\ \langle h(Top).val\rangle \frown abs(h(Top).next, \{Top\} \ndres h) \end{aligned}$$

Clearly, $Abs(\langle\rangle, Top, h)$ implies $Top = null$, so initialising Top to $null$, as shown in Fig. 6, correctly represents the empty stack. Initialising h to the empty function is also correct, though not actually required.

Using the data refinement calculation techniques from [MG90] along with simple properties of sequences, we can data refine $tryPush^2$ to:

$$\begin{matrix} Top, \\ h, \\ r \end{matrix} : \left[Inv(Top, h) \;\middle/\; \begin{array}{l} Inv(Top, h) \land Top \neq null \land h(Top).val = x \land \\ (h(Top).next, \{Top\} \ndres h) \approx (Top_0, h_0) \land r = true \ \lor \\ \\ (Top, h) \approx (Top_0, h_0) \land r = false \end{array} \right]$$

where the relation $\approx$ is defined as:

$$(Top_1, h_1) \approx (Top_2, h_2) \ \mathrel{\widehat{=}}\ Inv(Top_1, h_1) \land Inv(Top_2, h_2) \land abs(Top_1, h_1) = abs(Top_2, h_2),$$

i.e. (Top_1, h_1) and (Top_2, h_2) are both valid representations of the same abstract stack. Since $Inv(Top_0, h_0)$ follows from the precondition, and so does $Inv(h(Top).next, \{Top\} \ndres h)$ when $Top \neq null$, the interesting part of $(Top, h) \approx (Top_0, h_0)$ is $abs(Top, h) = abs(Top_0, h_0)$.

This specification allows a $tryPush$ operation to restructure the entire heap—for example, it may construct an entirely new linked list to represent the new stack. This freedom is important for universal constructions which automatically convert a sequential data structure into a lock-free or wait-free one [Her91], but that is very inefficient. To obtain a more efficient implementation, we choose to leave the rest of the heap unchanged when a successful $tryPush^2$ is performed, and establish $(h(Top).next, \{Top\} \ndres h) \approx (Top_0, h_0)$ by requiring $h(Top).next = Top_0$ and $\{Top\} \ndres h = h_0$.

Next, we observe that $Inv(Top, h)$ is equivalent to $Inv(h(Top).next, \{Top\} \ndres h)$ when $Top \neq null$, and we have already seen that $Inv(h(Top).next, \{Top\} \ndres h)$ follows from the precondition in this case. Finally, we rewrite $h(Top).val = x \land h(Top).next = Top_0 \land \{Top\} \ndres h = h_0$ as $Top \notin \mathrm{dom}\, h_0 \land h = h_0 \cup \{Top \mapsto (x, Top_0)\}$, highlighting the important requirement that Top be a new heap location. Leaving the precondition implicit, we obtain specification for $tryPush^3$ shown in Fig. 7. Note that we have left the weak requirement, $(Top, h) \approx (Top_0, h_0)$, in the second disjunct—we will see in Sect. 4.2.1 why this is important.

[6] Recall that $\{Top\} \ndres h$ is h with Top removed from its domain.

$$tryPush^3(\mathbf{in}\ x : T)\ r : bool \mathrel{\widehat=} \left[Top, h, r \;:\; \left[\begin{array}{l} Top \notin \mathrm{dom}\, h_0 \cup \{null\} \wedge \\ h = h_0 \cup \{Top \mapsto (x, Top_0)\} \wedge \\ r = true \ \vee \\ \\ (Top, h) \approx (Top_0, h_0) \wedge r = false \end{array}\right] \right]$$

$$tryPop^3(\mathbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat=} \left[Top, y, r \;:\; \left[\begin{array}{l} Top = Top_0 = null \wedge y = \perp \wedge r = true \ \vee \\ \\ Top_0 \neq null \wedge h(Top_0) = (y, Top) \wedge \\ \quad r = true \ \vee \\ \\ Top = Top_0 \wedge r = false \end{array}\right] \right]$$

Fig. 7. Concrete stack specification

Calculating the data refinement of $tryPop^2$ in a similar way gives:

$$\left[Top, h, y, r \;:\; \left[\begin{array}{l} Top_0 = null \wedge (Top, h) \approx (Top_0, h_0) \wedge y = \perp \wedge r = true \ \vee \\ \\ Top_0 \neq null \wedge y = h_0(Top_0).val \wedge \\ (h_0(Top_0).next), \{Top_0\} \lhd h_0) \approx (Top, h) \wedge r = true \ \vee \\ \\ (Top, h) \approx (Top_0, h_0) \wedge r = false \end{array}\right] \right]$$

Again, this allows the entire heap to be restructured by each operation. To obtain a more efficient implementation, we choose to always leave the heap unchanged, giving the specification for $tryPop^3$ shown in Fig. 7. This is an important design decision which simplifies the implementation, but means that popped heap locations remain as part of the heap and cannot be recycled. We will revisit this decision in Sect. 4.4.

We have shown that any occurrence of $tryPush^3(x)$ or $tryPop^3(y)$ can be reduced to $tryPush^2(x)$ or $tryPop^2(y)$, respectively, so $STACK^3$ is a valid refinement of $STACK^2$. Note that this data refinement replaces a single "abstract" action ($tryPush^2$ or $tryPop^2$) by a single "concrete" action ($tryPush^3$ or $tryPop^3$), so there is no possibility of introducing interference.

4.2. Implementing $tryPush^3$ and $tryPop^3$ ($STACK^4$)

We now wish to refine $tryPush^3$ and $tryPop^3$ so that they are lock-free and can be implemented using atomic instructions on a standard computer architecture. We will assume that a new heap node can be allocated in an atomic action, and that a single field of a node can be read or assigned in an atomic action. These actions are defined as follows:

$$\begin{array}{lcl} n := new\_node() & \mathrel{\widehat=} & n, h : \left[\, n \in \mathrm{dom}\, h \wedge \{n\} \lhd h = h_0 \,\right] \\ n.val := x & \mathrel{\widehat=} & h := h \oplus \{n \mapsto (x, h(n).next)\} \\ n.next := y & \mathrel{\widehat=} & h := h \oplus \{n \mapsto (h(n).val, y)\} \end{array}$$

In writing code, we assume implicit dereferencing of pointers, and thus also abbreviate $h(n).val$ and $h(n).next$ in expressions to $n.val$ and $n.next$.

We assume that there is no other way of accessing or inspecting the heap, so the only nodes that are visible to any stack operation are those it can reach by starting from Top, or a pointer stored in a local variable, and following $next$ pointers. Any dynamic storage used by the rest of the program is disjoint from the heap used to represent the stack.

4.2.1. Implementing $tryPush^3$

We first consider the specification of $tryPush^3$ shown in Fig. 7. The first disjunct says that a successful $tryPush^3$ has to allocate a new node ($Top \notin \mathrm{dom}\, h_0$) and initialise its fields ($h = h_0 \cup \{Top \mapsto (x, Top_0)\}$). Since Top has to point to a node whose $next$ field is the previous value of Top, we need to introduce a local variable to point to the new node and assign its value to Top after the $next$ field has been initialised. In a sequential setting we would implement this part of $tryPush$ in the obvious way, as:

$$\begin{array}{l}\textbf{var } n : Ptr \textbf{ ;}\\ n := new\_node() \textbf{ ;}\\ n.val := x \textbf{ ;}\\ n.next := Top \textbf{ ;}\\ Top := n\end{array}$$

In a concurrent environment, however, this would not be linearisable. Between assigning Top to $n.next$ and assigning n to Top, other processes may change the stack, so the $next$ field of the new node may no longer point to the top of the stack and the rest of the stack may be completely different. Following the pattern illustrated in the shared counter example in Sect. 2.2, we will take a snapshot of the shared state, and use it to check whether interference had occurred before updating Top. However, since the shared state consists of h as well as Top, copying and comparing the entire heap would be prohibitively expensive. Thus, we should attempt to identify a smaller part of the shared state to serve as a snapshot, which can be copied and compared more efficiently, but is also sufficient to determine whether the abstract value of the stack has changed.

We first introduce an existentially quantified variable to denote the new pointer value and use it in place of Top in the first two conjuncts:

$$\begin{array}{l}Top,\\ h,\\ r\end{array} \textbf{:} \left[\begin{array}{l}(\exists n : Ptr \bullet\\ \qquad n \notin \mathrm{dom}\, h_0 \wedge h = h_0 \cup \{n \mapsto (x,\ Top_0)\} \wedge\\ \qquad Top = n \wedge r = true) \ \vee\\ \\ (Top, h) \approx (Top_0, h_0) \wedge r = false\end{array}\right]$$

Next, we observe that $(Top, h) \approx (Top_0, h_0)$ can be established by leaving Top unchanged and augmenting h in the same way as in the first disjunct (i.e. $n \notin \mathrm{dom}\, h_0 \wedge h = h_0 \cup \{n \mapsto (x,\ Top_0)\} \wedge Top = Top_0 \Rightarrow (Top, h) \approx (Top_0, h_0)$). So we can move the quantifier and the first two conjuncts out of the disjunction, further refining $tryPush^3$ to:

$$\begin{array}{l}Top,\\ h,\\ r\end{array} \textbf{:} \left[\begin{array}{l}\exists n : Ptr \bullet\\ \qquad n \notin \mathrm{dom}\, h_0 \wedge h = h_0 \cup \{n \mapsto (x,\ Top_0)\} \wedge\\ \qquad (Top = n \wedge r = true \ \vee \ Top = Top_0 \wedge r = false)\end{array}\right]$$

and then introduce n as a local variable:

$$\begin{array}{l}\textbf{var } n : Ptr\textbf{;}\\ n,\ Top,\\ h,\ r\end{array} \textbf{:} \left[\begin{array}{l}n \notin \mathrm{dom}\, h_0 \wedge h = h_0 \cup \{n \mapsto (x,\ Top_0)\} \wedge\\ (Top = n \wedge r = true \vee Top = Top_0 \wedge r = false)\end{array}\right]$$

Next, we wish to split this specification into a sequential composition where the first component addresses the first two conjuncts and the second component addresses the last conjunct. However, we have to ensure that the disjunct $Top = n \wedge r = true$ is only chosen if the abstract stack is the same as it was when the first component was executed, which is where the snapshot is needed. Since we do not want to copy and compare the heap, our only other option is to use Top as the snapshot, but it is not obvious at this stage that this will work. So, we will pursue the idea of taking snapshots of both Top and h just far enough to be able to justify only using Top as the snapshot.

We therefore take a snapshot of the stack representation, recording the values of Top and h as ss and ssh, and then set Top to the new node only if the abstract stack is that represented by the snapshot (i.e. if $(Top_0, h) \approx (ss, ssh)$ holds). This gives us:

$$\textbf{var } n, ss : Ptr \textbf{ ; } ssh : Ptr \nrightarrow Node \textbf{ ;}$$

$$n,\ ss,\ ssh,\ h \textbf{:} \left[\begin{array}{l}n \notin \mathrm{dom}\, h_0 \wedge ss = Top \wedge ssh = h_0 \wedge\\ h = h_0 \cup \{n \mapsto (x,\ Top)\}\end{array}\right] \textbf{ ;} \qquad (A)$$

$$Top,\ r \textbf{:} \left[\begin{array}{ll}(Top_0, h) \approx (ss, ssh) \wedge Top = n \wedge r = true & \vee\\ Top = Top_0 \wedge r = false & \end{array}\right] \qquad (B)$$

$$
\begin{array}{ll}
tryPush^4(\mathbf{in}\ x : T)\ r : bool \mathrel{\widehat{=}} & \\
\quad \mathbf{var}\ n, ss : Ptr\ ; & \\
\quad n := new\_node()\ ; & (P1) \\
\quad n.val := x\ ; & (P2) \\
\quad ss := Top\ ; & (P3) \\
\quad n.next := ss\ ; & (P4) \\
\quad r := CAS(Top, ss, n) & (P5)
\end{array}
$$

Fig. 8. Initial implementation of $tryPush$

Now, consider how we can avoid copying and comparing the entire heap. First, we observe that A is the only action that modifies h, and it only changes h by augmenting it. Second, we observe that Top can only be modified by a process executing B or $tryPop$: B will make Top point to the newly added node, whose $next$ field points to the previous head of the list; $tryPop$ will may make Top point to the $next$ field of the node currently pointed to by Top. It follows that all nodes that are reachable from Top contain the val and $next$ values they were given when they were added to the heap. This means that if Top_1 and h_1 are the values of Top and h at one point in the program execution, Top_2 and h_2 are their values at a later point in the program execution, and $Top_1 = Top_2$, then $(\mathrm{dom}\ h_1) \lhd h_2 = h_1$, from which we can infer $(Top_1, h_1) \approx (Top_2, h_2)$. Intuitively, if Top has the same value at two points in an execution, then the abstract stack is the same at both points—if the stack is changed between these two points, it can only be by pushing elements onto the stack and then popping them off again, leaving the rest of the stack unchanged. Notice that this argument relies critically on the assumption that $tryPop^3$ does not remove pointers from dom h, so heap locations are never recycled; as indicated earlier, we will revisit this decision in Sect. 4.4.

The above argument shows that we can replace the test $(Top_0, h) \approx (ss, ssh)$ by the simpler test $Top_0 = ss$. We can now also remove ssh completely, and use ss in place of Top in updating h, giving:

$$
\begin{array}{ll}
\mathbf{var}\ n, ss : Ptr\ ; & \\
n, ss, h : \left[\begin{array}{l} n \notin \mathrm{dom}\ h_0 \land ss = Top \land \\ h = h_0 \cup \{n \mapsto (x, ss)\} \end{array} \right] ; & (A') \\
Top, r : \left[\begin{array}{l} Top_0 = ss \land Top = n \land r = true \lor \\ Top = Top_0 \land r = false \end{array} \right] & (B')
\end{array}
$$

We can now refine A' to a sequence of assignments to allocate and set n to a new location, initialise its fields, and store the snapshot of Top in ss. In doing this, we first perform the assignments that do not depend upon Top, so as to reduce the opportunity for other processes to interfere with this operation, and use ss in place of Top to initialise the new node. Finally, we can strengthen the second disjunct of B' so that it is only taken when $Top_0 \neq ss$, which allows us to implement B' using a CAS (cf. Fig. 1). Strengthening B' in this way also means that $tryPush$ will now only fail when interference is actually detected, which ensures that the implementation is lock-free. The complete implementation of $tryPush$ is shown in Fig. 8, where $P1$–$P4$ implement A' and $P5$ implements B'.

To show that this is a valid refinement, we show that any complete execution of $tryPush^4$ can be reduced to an equivalent execution of $tryPush^3$.

Now, any execution of $STACK^4$ containing a completed $tryPush$ operation by process p has a trace t of the form $\alpha\ P1_p\ \beta\ P2_p\ \gamma\ P3_p\ \delta\ P4_p\ \epsilon\ P5_p\ \zeta$, where $P5_p$ may succeed or fail. Note that, at this stage, we have not provided an implementation for $tryPop^4$, so we use $tryPop^3$ instead; i.e. we consider $tryPop$ operations to be performed atomically. We can assume that α, β, γ, δ, ϵ and ζ preserve $Inv(Top, h)$, since this is part of the postcondition of $tryPush^3$ and $tryPop^3$, and must be preserved by all of the atomic steps in their implementations.

Allocating a new node ($P1$) adds a new location to the heap, which by definition cannot be seen by any other process. This action is non-deterministic, since we do not know what location will be allocated, only that it will not be a location that has previously been allocated. If a and b are both $P1$ actions (by different processes), then changing the order of a and b does not lead to any observable difference in the resulting state, since if a allocates

l_1 and b allocates l_2, there is also a possible execution in which a allocates l_2 and b allocates l_1. If a is a $P1$ action and b is any action other than $P1$ occurring immediately after a, then b cannot depend on whether the location allocated by a is in the heap, so ab and ba are indistinguishable. Thus, we can treat $P1$ as a both mover.

The new heap node allocated by $P1$ only becomes visible to other processes when n is assigned to Top by a successful CAS at $P5$. Thus, when $P2$ and $P4$ are executed, n is a unique pointer to a node which cannot be seen by any other process, so changing the order of $P2$ or $P4$ and an action of another process cannot affect the resulting state. So we can also treat $P2$ and $P4$ as both movers.

Actions $P3$ and $P5$ require more careful consideration as they access Top. If $P5$ succeeds, we can move $P3$ right over δ and ϵ, since it will still read the same value. In this case, since $P1$, $P2$ and $P4$ can move right, t is equivalent to $\alpha\,\beta\,\gamma\,\delta\,\epsilon\,P1_p\,P2_p\,P3_p\,P4_p\,P5S_p\,\zeta$, and our discussion above shows that this is equivalent to $\alpha\,\beta\,\gamma\,\delta\,\epsilon\,tryPushS^3_p(x)\,\zeta$. The observable effect is the same as if execution of the $tryPush$ had been delayed until after ϵ and then completed without interruption. If $P5$ fails, we cannot justify moving either $P3$ or $P5$, since we know that they read different values of Top, which means that Top must be changed by δ or ϵ. Thus, if we moved $P3$ and $P5$ together, the CAS would then succeed and lead to a different execution. Although we cannot reduce the execution of $tryPush^4$ to a sequential execution in this case, we can still show that it is equivalent to an execution of $tryPush^3$. The fact that $P1$, $P2$ and $P4$ are both movers means that t is equivalent to $\alpha\,\beta\,\gamma\,P1_p\,P2_p\,P3_p\,P4_p\,\delta\,\epsilon\,P5F_p\,\zeta$, and it is easy to see that this is equivalent to $\alpha\,\beta\,\gamma\,tryPushF^3_p(x)\,\delta\,\epsilon\,\zeta$. The observable effect is the same as if execution of the $tryPush$ had been delayed until after γ and then completed without interruption—although this is not a behaviour that $tryPush^4$ can produce when executed atomically, it is allowed by $tryPush^3$.

Thus, we have shown that any complete execution of $tryPush^4$ can be reduced to an equivalent execution of $tryPush^3$, as required.[7]

It is also easy to see that $tryPush^4$ preserves $Inv(Top, h)$. We have already shown that $P5$ preserves $Inv(Top, h)$, and $P2$–$P4$ do not affect either Top or any node in h that is reachable from Top, so cannot affect the invariant. $P1$ adds a node to the heap, but it is not reachable from Top, so does not affect the invariant (more formally, $Inv(Top, \{n\} \lhd h) \wedge n \in \mathrm{dom}\, h \Rightarrow Inv(Top, h)$).

4.2.2. Implementing $tryPop^3$

Next, we consider the specification of $tryPop^3$ shown in Fig. 7. The first two disjuncts say that a successful $tryPop^3$ must either set y to $\perp$ and leave Top unchanged, if Top is $null$ ($Top = Top_0 = null \wedge y = \perp$), or else set y to the value in the first node of the list and Top to the next field of that node ($Top_0 = null \wedge h(Top_0) = (y, Top)$). In a sequential setting, we would implement this in the obvious way as:

if $Top = null$ **then**
  $y := \perp$
else
  $y := Top.val$ **;**
  $Top := Top.next$
fi

In a concurrent environment, however, this would not be linearisable, because between testing Top and updating it, other processes may change the stack. As with $tryPush$, we address this problem by taking a snapshot of Top, and using a CAS to update Top only if it is equal to the snapshot, which implies that the abstract stack is equal to its value at the point where the snapshot was taken. Again, we also use the snapshot to test whether the stack is empty and to access the element at the top of the stack.

We thus declare a local pointer variable ss and split the body of $tryPop^3$ into a sequential composition in which the first component initialises ss to Top, with the intention of refining the second component to an **if** statement. In forming the second component of the sequential composition, we need to consider carefully how to

[7] In [GC07], we introduced a weak form of CAS, which could fail when $loc = old$ holds, and used that to justify treating a failed CAS as a left mover. The approach adopted here provides a simpler justification for this case.

adjust the disjuncts of $tryPop$[3], since Top_0 plays two different rôles: holding the value of Top at the point where the snapshot was taken, and stipulating that Top is not changed. When the specification is split into a sequence, these rôles need to be considered separately.

- In the third disjunct, Top_0 is only used to ensure that Top is not changed. This disjunct will remain unchanged for now, so it can always be taken—we will see later when this option should be avoided.
- In the second disjunct, Top_0 is used to refer to the snapshot value, and this disjunct should only be chosen if Top has not changed since the snapshot was taken. We thus strengthen this disjunct with $Top_0 = ss$ and replace the other occurrences of Top_0 in the disjunct with ss; the statement will fail via the third conjunct when $Top_0 = ss$ does not hold.
- The first disjunct is more interesting, since Top_0 is used both to refer to the snapshot value when determining whether this disjunct may be chosen ($Top_0 = null$), and to ensure that Top is not changed ($Top = Top_0$). Thus, the first occurrence is replaced by ss, while the second remains unchanged. This decision is crucial to the correctness of the final algorithm and determines the location of the linearisation point for this case.

This gives us, as the body of $tryPop$:

$$\mathbf{var}\ ss : Ptr\ ;$$
$$ss := Top\ ;$$
$$Top, y, r : \left[\begin{array}{l} Top = Top_0 \land ss = null \land y = \bot \land r = true \ \lor \\ Top_0 = ss \land ss \neq null \land h(ss) = (y, Top) \land r = true \ \lor \\ Top = Top_0 \land r = false \end{array}\right]$$

In preparation for introducing an **if** statement, we strengthen the specification so that the third disjunct can only be selected when ss is not $null$. This disjunct was introduced to allow the operation to fail when interference is detected, which is not relevant when ss is $null$. We can now combine this with the second disjunct, leaving two disjuncts which will become the branches of the **if** statement:

$$Top, y, r : \left[\begin{array}{l} ss = null \land Top = Top_0 \land y = \bot \land r = true \ \lor \\ ss \neq null \land (Top_0 = ss \land h(ss) = (y, Top) \land r = true \lor Top = Top_0 \land r = false) \end{array}\right]$$

We now introduce the anticipated **if** statement, using $ss = null$ as the test. Since ss is local, its value cannot change between the test and the selected statement, so we can now omit $ss = null$ from the **then** part and $ss \neq null$ from the **else** part. We also expand the record equality, giving:

$$\mathbf{if}\ ss = null\ \mathbf{then}$$
$$Top, y, r : [\ Top = Top_0 \land y = \bot \land r = true\ ]$$
$$\mathbf{else}$$
$$Top, y, r : \left[\begin{array}{l} Top_0 = ss \land y = h(ss).val \land Top = h(ss).next \land r = true \ \lor \\ Top = Top_0 \land r = false \end{array}\right]$$
$$\mathbf{fi}$$

The first branch can now be implemented with two assignments, setting y to $\bot$ and r to $true$ ((Q3) and (Q4) in Fig. 9).

In the second branch, we can strengthen the second disjunct to assign y the same value as in the first disjunct, giving:

$$Top, y, r : \left[\begin{array}{l} Top_0 = ss \land y = h(ss).val \land Top = h(ss).next \land r = true \ \lor \\ Top = Top_0 \land y = h(ss).val \land r = false \end{array}\right]$$

Now, since y is local and we know that $h(ss).val$ cannot be changed, this can be refined to:

$$y := ss.val\ ;$$
$$Top, r : \left[\begin{array}{l} Top_0 = ss \land Top = h(ss).next \land r = true \ \lor \\ Top = Top_0 \land r = false \end{array}\right]$$

$$
\begin{array}{ll}
tryPop^4(\mathbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat{=}} & \\
\quad \mathbf{var}\ ss : Ptr\ \mathbf{;} & \\
\quad ss := Top\ \mathbf{;} & \text{(Q1)} \\
\quad \mathbf{if}\ ss = null\ \mathbf{then} & \text{(Q2)} \\
\quad\quad y := \perp\ \mathbf{;} & \text{(Q3)} \\
\quad\quad r := true & \text{(Q4)} \\
\quad \mathbf{else} & \\
\quad\quad y := ss.val\ \mathbf{;} & \text{(Q5)} \\
\quad\quad r := CAS(Top, ss, ss.next) & \text{(Q6)} \\
\quad \mathbf{fi} &
\end{array}
$$

Fig. 9. Initial implementation of $tryPop$

Finally, we can strengthen the second disjunct of the remaining specification statement so that it is only taken when $Top_0 \neq ss$, which allows this specification to be implemented as $r := CAS(Top, ss, ss.next)$ (cf. Fig. 1). As in the implementation of $tryPush$, this means that $tryPop$ will now only fail when interference is actually detected, which ensures that the implementation is lock-free. The resulting implementation of $tryPop$ is shown in Fig. 9.

To show that this is a valid refinement, we show that any complete execution of $tryPop^4$ can be reduced to an equivalent execution of $tryPop^3$. Let t be a trace containing a completed execution of $tryPop^4$ by process p. We need to consider three possible execution paths for $tryPop^4$, according to the outcomes of $Q2$ and $Q6$.

If $Q2$ succeeds, trace t is of the form $\alpha\ Q1_p\ \beta\ Q2S_p\ \gamma\ Q3_p\ \delta\ Q4_p\ \epsilon$. Since ss, y and r are local (recall that we treat an **out** parameter as a local), $Q2$, $Q3$ and $Q4$ are both-movers. However, we cannot move $Q1$, since it reads a shared variable and we have no way to check whether its value has changed, so we take $Q1$ as the linearisation point in this case. Moving $Q2$, $Q3$ and $Q4$ left over β, γ and δ, we see that t is equivalent to $\alpha\ Q1_p\ Q2S_p\ Q3_p\ Q4_p\ \beta\ \gamma\ \delta\ \epsilon$, which is equivalent to $\alpha\ tryPopS^3(y)\ \beta\ \gamma\ \delta\ \epsilon$. The observable effect is the same as if the operation completed without interruption as soon as it began, and $Q1$ is the linearisation point since it is only at that point that we know the stack was empty. It does not matter whether the stack is still empty when the $tryPop$ returns—$tryPop$ can return $\perp$ as long as the stack was observed to be empty at some point in its execution.

If $Q2$ fails and $Q6$ succeeds, trace t is of the form $\alpha\ Q1_p\ \beta\ Q2F_p\ \gamma\ Q5_p\ \delta\ Q6S_p\ \epsilon$. Since $Q6S$ updates Top, it cannot be moved and will be the linearisation point. Since ss is local, $Q2$ is a both mover, and since y and ss are local and no process can modify the val field of a node accessible from Top, we can treat $Q5$ as a both mover. Also, since $Q6$ succeeds, we can move $Q1$ right over β, γ and δ, because we know it will still read the same value and this implies that the abstract stack is the same as it was when the snapshot was taken. Thus, t is equivalent to $\alpha\ \beta\ \gamma\ \delta\ Q1_p\ Q2S_p\ Q5_p\ Q6S_p\ \epsilon$, which is equivalent to $\alpha\ \beta\ \gamma\ \delta\ tryPopS^3(y)\ \epsilon$. The observable effect is the same as if execution of the $tryPop$ had been delayed until after δ and then completed without interruption.

If $Q2$ and $Q6$ both fail, trace t is of the form $\alpha\ Q1_p\ \beta\ Q2F_p\ \gamma\ Q5_p\ \delta\ Q6F_p\ \epsilon$. $Q2$ and $Q5$ are again both movers, as explained above. However, $Q1$ and $Q6$ both read Top, and we cannot move either of them as we know that Top is changed by the intervening actions. Thus, we can show that t is equivalent to $\alpha\ Q1_p\ Q2S_p\ Q5_p\ \beta\ \gamma\ \delta\ Q6S_p\ \epsilon$, which is equivalent to $\alpha\ tryPopF^3(y)\ \beta\ \gamma\ \delta\ \epsilon$. The observable effect is the same as if execution of the $tryPop^3$ had been completed without interruption as soon as it began, with p choosing to fail. As with the failure case in $tryPush^4$, although this is not a behaviour that $tryPop^4$ can produce when executed atomically, it is allowed by $tryPop^3$.

Thus, we have shown that any complete execution of $tryPop^4$ can be reduced to an equivalent execution of $tryPop^3$.

It is also easy to see that each step of $tryPop^4$ preserves $Inv(Top, h)$, since $tryPop$ does not alter h and the only action that alters Top is the CAS in $P6$, which has already been shown to preserve $Inv(Top, h)$.

We have shown that any complete execution of $tryPush^4$ can be reduced to an equivalent execution of $tryPush^3$, assuming that $tryPop^3$ is atomic, and that any complete execution of $tryPop^4$ can be reduced to an equivalent execution of $tryPop^3$. It follows that for any execution of $STACK^4$ there is equivalent execution of $STACK^3$,

$push^5(\textbf{in}\ x : T) \mathrel{\widehat{=}}$
  var $n : Ptr$;
  $n := new\_node()$; $(P1)$
  $n.val := x$; $(P2)$
  do
    $tryPush^5(n)$
  od

$tryPush^5(\textbf{in}\ n : Ptr)\ r : bool \mathrel{\widehat{=}}$
  var $ss : Ptr$;
  $ss := Top$; $(P3)$
  $n.next := ss$; $(P4)$
  $r := CAS(Top, ss, n)$ $(P5)$

$pop^5(\textbf{out}\ y : T_\perp) \mathrel{\widehat{=}}$
  var $n : Ptr$;
  do
    $tryPop^5(n)$
  od
  $y := \textbf{if}\ n = null\ \textbf{then}\ \perp\ \textbf{else}\ n.val\ \textbf{fi}$ $(Q7)$

$tryPop^5(\textbf{out}\ n : Ptr)\ r : bool \mathrel{\widehat{=}}$
  var $ss : Ptr$;
  $ss := Top$; $(Q1)$
  if $ss = null$ **then** $(Q2)$
    $n := null$; $(Q3')$
    $r := true$ $(Q4)$
  else
    $n := ss$; $(Q5')$
    $r := CAS(Top, ss, ss.next)$ $(Q6)$
  fi

Fig. 10. Improved concrete stack implementation

and so that $STACK^4$ is a valid refinement of $STACK^3$. The assumption that $tryPop^3$ is atomic means that we reduce all execution of $tryPop^4$ before reducing executions of $tryPush^4$, but we would not have been needed if we had delayed proving $tryPush^4$ until we had completed the implementation of $tryPop^4$.

This algorithm is lock-free, subject to our assumption that new_node is atomic, 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.

4.3. A small optimisation ($STACK^5$)

The implementation of $push$ obtained by combining $push^2$ and $tryPush^4$ (Figs. 5, 8) needlessly allocates a new node for each execution of $tryPush$, and the nodes allocated in failed attempts are not recycled. It would be better to allocate a new node and initialise it with x in $push$, and pass a pointer to it to $tryPush$, so that only one node is allocated for each execution of $push$. Similarly, the implementation of pop obtained by combining pop^2 and $tryPop^4$ (Figs. 5, 9) reads the value of $ss.val$ in every execution of $tryPop$, which is wasted effort in the case of failed attempts. It would be better for $tryPop$ to return a pointer to the popped node (or $null$ in the case where the stack is empty), so the return value is extracted in pop, and this is only done once a node has been successfully removed from the stack. This suggests an implementation in which the arguments to $tryPush$ and $tryPop$ are pointers to nodes, rather than values, as shown in Fig. 10.

We could obtain this implementation by redoing the entire derivation so as to introduce the concrete data representation, and split $push$ and pop into sequential compositions, before introducing the try–retry loops. Then, the first component of the sequence in $push$ could allocate and initialise the new node, and the second component of the sequence in pop could extract the return result from the popped node. Taking this approach initially would require considerable foresight into both the structure of the final algorithm and the reasoning required to obtain it—our reasons for taking the approach we did were explained at the beginning of Sect. 3.

Rather than redoing the derivation, we can show that this is a correct refinement by showing how to translate any execution of $STACK^5$ into an equivalent execution of $STACK^4$. First, consider an execution containing a completed execution of $push^5$ by process p. This execution of $push^5$ consists of one execution of $P1_p$ and $P2_p$, followed by one or more executions of $P3_p$, $P4_p$ and $P5_p$, possibly interleaved with steps of other processes. To obtain an equivalent execution of $push^4$, we simply need to insert new occurrences of $P1_p$ and $P2_p$ before each

occurrence of $P3_p$ after the first. The resulting execution is equivalent since it still only adds one node to the linked list—the other nodes added to the heap have no effect on the abstract stack.

Next, consider an execution containing a completed execution of pop^5 by process p. The only difference between pop^5 and pop^4 is that where pop^4 assigns to y ($Q3_p$ and $Q5_p$), pop^5 assigns to n ($Q3'_p$ and $Q5'_p$), and pop^5 has the additional statement $Q7$. We can thus obtain an equivalent execution of pop^4 by replacing all occurrences of $Q3'_p$ and $Q5'_p$ by $Q3_p$ and $Q5_p$, respectively, and deleting $Q7_p$. To see that this preserves equivalence, notice that $Q7$ is equivalent to $Q3$ (i.e. $y := \bot$) when the last assignment to n is $Q3'$, and to $Q5$ (i.e. $y := ss.val$) when the last assignment to n is $Q5'$, and that pop^4 reads the same value at $Q5$ as pop^5 reads at $Q7$ since the val field of a node cannot be changed after it has been assigned in the $push$ operation that added it to the list.

This implementation is essentially what [HSY04] use as the basis for their elimination algorithm (their $tryPerformStackOp$ operation incorporates our $tryPush$ and $tryPop$ operations). Expanding $tryPush$ and $tryPop$ in-line and making a couple of small simplifications gives an algorithm which is the same as Treiber's stack algorithm (as given in [MS98]), except for the handling of dynamic memory, which we discuss below.

4.4. Memory management and the ABA problem ($STACK^6$)

The above derivation relied critically on the fact that heap locations are not recycled. If we allowed pop to free the node it removes from the stack while another process holds a pointer to it, that process could subsequently get a memory violation when it attempts to access the node because the storage it occupies is no longer part of this program's address space. In terms of our model, the pointer would be removed from the domain of h, and a subsequent attempt to evaluate h at that point would lead to an undefined result. While ensuring that storage is never recycled gives us a correct stack implementation, this solution is impractical (except perhaps for short run programs), because it requires storage proportional to the number of $push$ operations performed—which is usually regarded as a "memory leak".

4.4.1. Reusing memory

The ideal solution to this problem is to ensure that a popped node is only freed if no other process holds a pointer to it in a local variable (in this implementation, no node reachable from Top can contain a pointer to the node being popped). This can be achieved by modifying the implementation to keep reference counts for all heap locations, or by using an implementation language that provides automatic garbage collection. In the latter case, the implementation would be unchanged, but to justify this approach, we would need to modify the derivation to weaken the specification of $tryPush^3$ in Fig. 7 to require only that the new value of Top is not reachable from any pointer variable used in the program rather than just $Top \notin \text{dom } h_0$—alternatively, this requirement could be built into a data refinement mapping non-recyclable nodes to recyclable ones.[8] Such an implementation would only be lock-free if the reference counting mechanism or garbage collector is lock-free. Using automatic garbage collection is not always applicable—for example, it is not possible to take this approach if one is using the stack in order to implement such a garbage collector.

A simple alternative is for the implementation to maintain its own free list. Popped nodes are added to the free list, not released to the system; nodes are then allocated from the free list if possible, only allocating new nodes if the free list is empty. Since nodes are never freed to the system, this removes the possibility of memory violations. Memory use is now proportional to the historical maximum stack size, which is a big improvement on the number of $push$ operations performed.[9] Figure 11 shows specifications for a free list with $free\_node$ and get_node operations—the free list is modelled as a set, since the order in which nodes are added to and removed from the free list is immaterial.

[8] While it is straightforward to formalise the notion of a pointer being reachable from a shared variable, such as Top, to formalise the notion of being reachable from a local variable we need some way to identify processes that are currently executing a stack operation (e.g. using explicit program counters or control predicates [Lam88], or using a more explicit storage model mapping process names to their local variables when the process is executing a stack operation), since the value of a local variable in a procedure is only meaningful while that procedure is being executed.

[9] This should be acceptable in most situations where the stack grows and shrinks in a somewhat uniform way. It is still not ideal, however, and may be unacceptable—for example, if the stack grows to a very large size early in the execution and is thereafter always very small.

$$\textbf{var } free : \textbf{set } Ptr := \varnothing$$

$$free\_node(\textbf{in } n : Ptr) \mathrel{\widehat{=}} free : [\, free = free_0 \cup \{n\} \,]$$

$$get\_node(\textbf{out } n : Ptr) \mathrel{\widehat{=}} n, h, free : \left[\begin{array}{l} n \in \operatorname{dom} h \wedge \{n\} \lhd h = \{n\} \lhd h_0 \wedge \\ (free_0 \neq \varnothing \Rightarrow n \in free_0 \wedge free = free_0 \setminus \{n\}) \end{array}\right]$$

Fig. 11. Specification for free list

```
var Free : Ptr := null

free_node(in n : Ptr) ≙
  var ss : Ptr ;
  do
      ss := Free ;
      n.next := ss ;
      CAS(Free, ss, n)
  od
```

```
get_node(out n : Ptr) ≙
  var ss : Ptr ;
  do
      ss := Free ;
      if ss = null then
        n := new_node() ;
        exit
      else
        n := ss ;
        CAS(Free, ss, ss.next)
      fi
  od
```

Fig. 12. Implementation for free list

The free list can be implemented using code very similar to that shown in Fig. 10, where *free_node* is like *push*, except that it takes a node to be added to the free list (so $P1$ and $P2$ are deleted), and *get_node* is like *pop*, except that it returns the node it removes from the free list if there is one (so $Q7$ is deleted), and allocates a new node if there is not (so $Q3'$ is replaced by $P1$). This code is shown in Fig. 12. We can verify the free list implementation in much the same way that we verified the stack implementation above.

We can introduce a free list into the stack implementation as a data refinement which partitions the domain of h into those nodes that are reachable from *Top* and those that are not, with the latter being represented by *free*. The representation invariant says that *free* contains all pointers in that are not reachable from *Top*, and the abstraction relation maps elements of dom $h \setminus free$ in the new model to elements of dom h in the previous model, so the data refined version of *new_node* is able to return a pointer from *free*. In Fig. 10, we then replace the call on *new_node* in $P1$ by a call on *get_node*, which satisfies this weakened specification. To maintain the invariant on *free*, we also insert a call on *free_node* in *pop*, in the **else** part of $Q7$. These changes are incorporated in Fig. 13.

4.4.2. The ABA problem

If we attempt to verify the code obtained by adding memory reuse to $STACK^5$, as described above, we get stuck. We can no longer guarantee that n in *tryPush* is the only pointer to the new node, since another process may hold a pointer to it in a local variable, which invalidates our assumption that the fields of a node are not modified once they become reachable from *Top*.

Moreover, we can no longer determine whether the stack has the same value when a CAS is attempted that it had at an earlier point, just by comparing the value of *Top* with the saved snapshot ss. This was an essential part of our reasoning in the preceding derivation, and relied critically on the fact that heap locations are not recycled to infer that the stack has the same value at two points in the program execution from the fact that *Top* has the same value at these points.

Once we allow popped nodes to be recycled, it becomes possible, between a *tryPush* taking a snapshot and performing its CAS, for the node at the top of the stack to be popped by another process and then pushed again by another process, after the rest of the stack has changed. In this case, we cannot infer that the stack value is the same just by comparing *Top* and ss—we really want to know whether *Top* has changed, not just whether it has the same value that it had before.

```
type CountedPtr ≙ (ptr : Ptr ; count : nat)

type Node ≙ (val : T ; next : Ptr)

var Top : CountedPtr := (null, 0)

push^6(in x : T) ≙
  var n : Ptr ;
  n := get_node() ;
  n.val := x ;
  do
    tryPush^6(n)
  od

tryPush^6(in n : Ptr) r : bool ≙
  var ss : CountedPtr ;
  ss := Top ;
  n.next := ss.ptr ;
  r := CAS(Top, ss, (n, ss.count + 1))

pop^6(out y : T⊥) ≙
  var n : Ptr ;
  do
    tryPop^6(n)
  od ;
  if n = null then
    y := ⊥
  else
    y := n.val ;
    free_node(n.ptr)
  fi

tryPop^6(out y : Ptr) r : bool ≙
  var ss : CountedPtr ;
  ss := Top ;
  if ss.ptr = null then
    y := null ;
    r := true
  else
    y := ss.ptr ;
    r := CAS(Top, ss, (ss.ptr.next, ss.count + 1))
  fi
```

Fig. 13. Stack implementation with modification counts

This kind of situation is called an *ABA problem*, because it arises when it is possible for a variable being updated using a CAS to change from one value (A) to another value (B) and then back to its former value (A), and we really want to know whether the variable has been changed. CAS can only tell whether the value of a variable is different from the previous value, not whether it has changed in between. Note, however, that the possibility of a variable changing back to an earlier value does not always constitute an ABA problem: for example, it is not a problem for a stack when storage is not recycled, or for the shared counter discussed in Sect. 2.2.

One way to avoid the ABA problem when reusing heap locations is to avoid using CAS and instead use the Linked-Load/Store-Conditional (LL/SC) pair of instructions, in which an SC succeeds only if the variable it operates on has not changed since it was last loaded by that process using an LL. Unfortunately, suitable LL/SC instructions are not widely available on current architectures, though they can be simulated using CAS (see [Moi97]).

A popular response to the ABA problem, adopted in many lock-free algorithms, including Treiber's stack and Michael and Scott's queue [MS98], is to associate a *modification count* with each pointer variable for which changes need to be detected. If the modification count is incremented every time the variable is modified, an ABA situation can detected by comparing the modification count along with the pointer value—if the pointer has changed back to its prior value, this change will be detected because the modification count will have changed.

In the stack implementation in Fig. 10, the only pointer variable for which we need to detect changes is Top. So we change the type of Top to a new "counted pointer" type $CountedPtr$, which packages a pointer and a modification count into a single value. We then ensure that the pointer component is extracted in places where it is required, and that the counter component is updated every time Top is modified. Thus, 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. The resulting code is shown in Fig. 13, which is essentially the same as the version of Treiber's algorithm given in [MS98]—the latter does not separate out $tryPush$ and $tryPop$,

which leads to a little simplification in the logic for handling *pop* on an empty stack, and it has unnecessary modification counts on list nodes and a redundant assignment in *push*.[10]

To show that $STACK^6$ is a valid refinement of $STACK^5$, we must show that any complete execution of $tryPush^6$ or $tryPop^6$ can be reduced to an equivalent execution of $tryPush^5$ or $tryPop^5$, respectively. To do this, we must show that a completed execution of *get_node* can be treated as a right mover, and that a completed execution of *free_node* can be treated as a left mover. This can be done using reasoning similar to that used earlier. In this case, however, we may need to swap the order of CAS operations updating *Free*, but the results will still satisfy the specifications for *get_node* and *free_node*. We also need to show that if $Top.ptr$ changes from one value to another and back to its previous value, then the value of *Top* in the two states is different, because the modification count has changed. With this reasoning, we see that in fact we only need to update the modification count when a push occurs, since a *pop* alone cannot return a former node to the top of the stack.

This "verification" relies on the assumption that repeatedly increasing a modification count will never lead to an earlier value being repeated, which is only valid if modification counts are unbounded. To be practical, however, we must assume that a pointer and its modification count can be tested and assigned atomically using a CAS—for example, 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, and is generally assumed to be small enough to make this solution acceptable for practical purposes, and can be eliminated with a small cost overhead [Moi97]. At this stage, we should also introduce a "concrete" pointer type, with a finite range, to reflect the fact that heap space is also finite, and allow *new_node* to fail (e.g. returning *null*) if the heap is exhausted.

Efficient management of dynamic storage in lock-free algorithms is the subject of on-going research and further discussion is beyond the scope of this paper (see, for example [DMMS01, DHLM04, HLMM05]).

5. The elimination stack

The stack implementation presented in Sect. 4 works well at medium loads, but does not scale well [HSY04]. 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 et al. [HSY04] propose an algorithm, building on a central stack implementation like the one in Sect. 4, which incorporates two key ideas.

Firstly, if a process fails in its attempt to apply an operation, it waits for a while 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 which is waiting to retry an operation looks for a complementary operation to eliminate with. If each process has its own processor, this will be done while the processor would otherwise be idle.

The description presented in [HSY04] focuses on the two arrays used to implement their elimination mechanism and on its performance characteristics. Here, we attempt to give a more illuminating account of the elimination algorithm by starting with an abstract model of elimination in which the different rôles of the processes involved in an elimination are described separately. We then show how these rôles can be combined, and how the treatment of *push* and *pop* can be combined, and refine the resulting specification to pseudocode that can be

[10] The code Treiber gives in [Tre86] is written in System/370 assembler and provides a stack-like implementation of a free list, similar to that given in Fig. 12 but with a modification count for *Free*. It does not include the storage management operations shown in Michael and Scott's version. It only keeps a modification count on the head of the free list, and this is only incremented when an element is added to the list. Treiber also uses a form of CAS instruction which, when it fails, replaces *old* (which must be a local variable) with the current value of the shared location—this means that *Top* does not need to be read into *ss* each time a *tryPush* is attempted.

$push^7(\mathbf{in}\ x : T) \mathrel{\widehat{=}}$
do
$tryPush^7(x)\ []\ tryPushElim^7(x)$
od

$pop^7(\mathbf{out}\ y : T_\perp) \mathrel{\widehat{=}}$
do
$tryPop^7(x)\ []\ tryPopElim^7(x)$
od

$tryPushElim^7(\mathbf{in}\ x : T)\ r : bool \mathrel{\widehat{=}}$
$s, r : \left[\begin{array}{l} s = \langle x \rangle \frown s_0 \wedge r = true \vee \\ s = s_0 \wedge r = false \end{array} \right]$

$tryPopElim^7(\mathbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat{=}}$
$s, y, r : \left[\begin{array}{l} s_0 = \langle y \rangle \frown s \wedge r = true \vee \\ s = s_0 \wedge r = false \end{array} \right]$

Fig. 14. Introducing elimination ($tryPush^7 \equiv tryPush^2$ and $tryPop^7 \equiv tryPop^2$)

executed on an architecture supporting atomic load, store and CAS operations. Our derivation initially assumes only that the central stack implementation provides *tryPush* and *tryPop* operations, as in Fig. 5; in Sect. 5.7, we introduce an optimisation which relies on the *Node* type used in Sect. 4. The resulting code differs from [HSY04]'s in a number of ways, and we conclude the section by discussing these differences.

5.1. An abstract model of elimination ($STACK^7$)

The basic idea of elimination is that, instead of immediately retrying a failed operation on the central stack, as we did in Fig. 5, 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.[11] 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 directly approaching other waiting customers, or by posting or inspecting notices on a notice board. We will use the latter analogy in developing an abstract model of elimination.

In [HSY04], a process p performing a stack operation first attempts its operation on the stack, as described in Sect. 4. If this attempt fails, however, instead of immediately retrying, p attempts to match up with a process attempting 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 [MNSS05] to implement a scalable lock-free queue).

In proving linearisability, we can simply treat elimination as an alternative way in which an operation may satisfy its specification, ignoring for now that elimination operations have to occur in pairs, and non-deterministically select which alternative to try each time an operation is attempted. Thus, we obtain $STACK^7$ (see Fig. 14) from $STACK^2$ (Fig. 5), by refining the loop bodies in *push* and *pop* to a non-deterministic choice between trying on the stack and trying to eliminate. The new operations, $tryPushElim^7$ and $tryPopElim^7$, have the same specifications as $tryPush^7$ and $tryPop^7$ (except that $tryPopElim^7$ will never return $\perp$), but will be implemented differently.[12]

This modification is clearly a valid refinement, since at this level *tryPushElim* is equivalent to *tryPush* and *tryPopElim* refines *tryPop*, so occurrences of *tryPushElim* and *tryPopElim* can be transformed in the same way that occurrences of *tryPush* and *tryPop* were transformed in Sect. 3.2.

Throughout this discussion, we will assume that $tryPush^7$ and $tryPop^7$ are atomic actions operating on the abstract stack, as in $tryPush^2$ and $tryPop^2$. No interference is possible between these operations and the elimination mechanism, since the latter does not access s. Once the derivation is complete, we can insert suitable implementations for *tryPush* and *tryPop* (e.g. those in either Figs. 10 or 13). In the initial implementation, which

[11] 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.

[12] Although $tryPopElim^7$ cannot return $\perp$, because the value it returns will always the result of a successful elimination, it will be convenient later to keep the type of the returned values as $T_\perp$.

is completed in Sect. 5.6, there is still no possibility of interference, because the elimination mechanism uses separate data structures from the central stack (in particular, it makes no use of heap storage). The final version of the elimination mechanism, discussed in Sect. 5.7, is specialised to use the same node type as in the central stack representation, at which point we will revisit this issue.

5.2. Distinguishing active and passive elimination ($STACK^8$)

To describe elimination more precisely, we need to consider the rôles of the two processes involved. We will assume, as in [HSY04], that an elimination is initiated by one of these processes.[13] Thus, continuing with our notice board analogy, a process attempting to find a matching process to eliminate with may proceed in either of two ways:

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

Thus, the notice board can be viewed as a set of *requests*, each describing an operation which is either waiting to be performed or has already been performed. 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 using a Z-like discriminated union type (using the notation of [Mor94, Chap. 15]), where $Push(T)$ and Pop describe waiting requests, and $PushDone$ and $PopDone(T_\perp)$ describe completed requests:

$$Request \mathrel{\widehat=} None \mid Push(T) \mid Pop \mid PushDone \mid PopDone(T_\perp)$$

Since there can be at most one request associated with each process, we will model the notice board (N) as a function from processes to requests, with $None$ indicating that a process currently has no request on the notice board. So, initially, $N(p) = None$ for every process p.

We will assume, again following [HSY04], that each process performing an operation may take either an active or a passive approach.[14] At this level of abstraction, we can regard this as a non-deterministic choice, giving the description in Fig. 15—we will examine this choice more closely in Sect. 5.4.

In $tryActivePush^8$ and $tryActivePop^8$, the first disjunct describes successful active elimination as outlined above: q is some process attempting a complementary operation (and so must be distinct from p), and its request is modified to show that it has been fulfilled. The second disjunct allows an active elimination attempt to fail if a suitable partner is not found—again, we treat this as a non-deterministic choice and do not specify at this stage the conditions under which an elimination attempt may fail, in particular, we do not insist that elimination only fail if there is no suitable partner available.

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 between them—there is no need to model the waiting explicitly. In $tryPassivePush^8$ and $tryPassivePop^8$, the first specification statement adds p's request to the notice board, while the second describes p's behaviour after its delay. The elimination attempt will succeed if p's request has changed to the corresponding completed request (first disjunct), and will fail if the request is unchanged (second disjunct). In both cases, p removes its request from the notice board.[15]

To show that this is a valid refinement, we show that any trace of $STACK^8$ can be transformed into an equivalent trace of $STACK^7$.

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

[14] We could consider other alternatives, for example, having either $push$ or pop always active and the other always passive. This would simplify the code, but it is not clear how it would affect performance.

[15] Note that the $tryPassive$ operations depend on the identity of the current process (p), or some equivalent mechanism, to identify this process's request.

$$\textbf{type}\ \mathit{Request} \mathrel{\widehat{=}} \mathit{None} \mid \mathit{Push}(T) \mid \mathit{Pop} \mid \mathit{PushDone} \mid \mathit{PopDone}(T_\perp)$$

$$\textbf{var}\ N : \mathcal{P} \to \mathit{Request} := (\lambda\, p : \mathcal{P} \bullet \mathit{None})$$

$$\mathit{tryPushElim}^8(\textbf{in}\ x : T)\ r : \mathit{bool} \mathrel{\widehat{=}} r := \mathit{tryActivePush}^8(x) \ [] \ r := \mathit{tryPassivePush}^8(x)$$

$$\mathit{tryPopElim}^8(\textbf{out}\ y : T_\perp)\ r : \mathit{bool} \mathrel{\widehat{=}} r := \mathit{tryActivePop}^8(y) \ [] \ r := \mathit{tryPassivePop}^8(y)$$

$$\mathit{tryActivePush}^8(\textbf{in}\ x : T)\ r : \mathit{bool} \mathrel{\widehat{=}} N, r : \left[\begin{array}{l}(\ \exists\, q : \mathrm{dom}\, N_0 \bullet \\ \quad N_0(q) = \mathit{Pop} \wedge \\ \quad N = N_0 \oplus \{q \mapsto \mathit{PopDone}(x)\} \wedge \\ \quad r = \mathit{true}\ ) \\ \qquad \vee \\ N = N_0 \wedge r = \mathit{false}\end{array}\right]$$

$$\mathit{tryActivePop}^8(\textbf{out}\ y : T_\perp)\ r : \mathit{bool} \mathrel{\widehat{=}} N, y, r : \left[\begin{array}{l}(\ \exists\, q : \mathrm{dom}\, N_0 \bullet \\ \quad N_0(q) = \mathit{Push}(y) \wedge \\ \quad N = N_0 \oplus \{q \mapsto \mathit{PushDone}\} \wedge \\ \quad r = \mathit{true}\ ) \\ \qquad \vee \\ N = N_0 \wedge r = \mathit{false}\end{array}\right]$$

$$\mathit{tryPassivePush}_p^8(\textbf{in}\ x : T)\ r : \mathit{bool} \mathrel{\widehat{=}} N : \left[\ N = N_0 \oplus \{p \mapsto \mathit{Push}(x)\}\ \right]; \quad (C)$$

$$N, r : \left[\begin{array}{l}N_0(p) = \mathit{PushDone} \wedge \\ N = N_0 \oplus \{p \mapsto \mathit{None}\} \wedge r = \mathit{true} \\ \qquad \vee \\ N_0(p) = \mathit{Push}(x) \wedge \\ N = N_0 \oplus \{p \mapsto \mathit{None}\} \wedge \\ r = \mathit{false}\end{array}\right] \quad (D)$$

$$\mathit{tryPassivePop}_p^8(\textbf{out}\ y : T_\perp)\ r : \mathit{bool} \mathrel{\widehat{=}} N : \left[\ N = N_0 \oplus \{p \mapsto \mathit{Pop}\}\ \right]; \quad (C')$$

$$N, y, r : \left[\begin{array}{l}N_0(p) = \mathit{PopDone}(y) \wedge \\ N = N_0 \oplus \{p \mapsto \mathit{None}\} \wedge r = \mathit{true} \\ \qquad \vee \\ N_0(p) = \mathit{Pop} \wedge \\ N = N_0 \oplus \{p \mapsto \mathit{None}\} \wedge \\ r = \mathit{false}\end{array}\right] \quad (D')$$

Fig. 15. Abstract description of elimination

We first consider successful elimination—this is the most interesting case because we have to reduce a successful active elimination attempt and a complementary successful passive elimination attempt at the same time, emphasising the fact that elimination should really be thought of as a single action involving two processes.

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 changed to its "completed" counterpart. Since a process can only add or remove its own request and can only modify another process's request (this is easy to verify—recall that in *tryActivePush* and *tryActivePop*, q must be distinct from p), 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 $\mathit{tryPassivePush}_p$ has a trace t of the form $\alpha\ C_p\ \beta\ \mathit{tryActivePopS}_q\ \gamma\ D_p\ \delta$, where this occurrence of $\mathit{tryActivePopS}_q$ modifies $N(p)$. The fact that $\mathit{tryActivePop}_q$ and D_p both succeed implies that β and γ do not affect $N(p)$, so t is equivalent to $\alpha\ \beta\ C_p\ \mathit{tryActivePopS}_q(x)\ D_p\ \gamma\ \delta$. We cannot move the C_p and D_p together, since $\mathit{tryActivePopS}_q$ must occur between them; instead we will reduce these actions together. The combined effect of C_p, $\mathit{tryActivePopS}_q$ and D_p, executed consecutively, is equivalent $\mathit{tryPushElimS}_p^7(x)\ \mathit{tryPopElimS}_q^7(x)$. In both cases, the combined effect is to set q's output variable (y) to p's input variable (x) and leave the shared variables (N and s, respectively) unchanged. 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, since we only consider completed operations, 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 reduce a successful $\mathit{tryActivePush}^8$ and a successful $\mathit{tryPassivePop}^8$ to a $\mathit{tryPushElim}^7$ followed by a $\mathit{tryPopElim}^7$.

We next consider unsuccessful elimination. An unsuccessful $\mathit{tryActivePush}^8$ or $\mathit{tryActivePop}^8$ has no observable effect, and can be reduced to a $\mathit{tryPushElimF}^7$ or $\mathit{tryPopElimF}^7$, respectively. An execution containing an unsuccessful passive elimination attempt is either of the form $\alpha\ C_p\ \beta\ D_p\ \gamma$ or $\alpha\ C'_p\ \beta\ D'_p\ \gamma$. In either case,

$$tryPushElim^9(\mathbf{in}\ x : T)\ r : bool \mathrel{\widehat=}$$
$$\mathbf{var}\ y : T_\perp\ ;$$
$$\left(\begin{array}{ll} & r := tryActive^9(Push(x), y) \\ [] & r := tryPassive^9(Push(x), y) \end{array}\right)$$

$$tryActive^9(\mathbf{in}\ req : Request\ ;\ \mathbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat=}$$
$$N, y, r\ :\ \left[\begin{array}{l} (\ \exists\, q : \operatorname{dom} N_0 \bullet opp(req, N_0(q)) \land \\ \quad N = N_0 \oplus \{q \mapsto done(N_0(q), val(req))\} \land \\ \quad y = val(N_0(q)) \land r = true\ ) \\ \qquad \lor \\ N = N_0 \land r = false \end{array}\right]$$

$$tryPopElim^9(\mathbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat=}$$
$$y := \perp\ ;$$
$$\left(\begin{array}{ll} & r := tryActive^9(Pop, y) \\ [] & r := tryPassive^9(Pop, y) \end{array}\right)$$

$$tryPassive_p^9(\mathbf{in}\ req : Request\ ;\ \mathbf{out}\ y : T_\perp)\ r : bool \mathrel{\widehat=}$$
$$N : [\ N = N_0 \oplus \{p \mapsto req\}\ ]\ ; \qquad (TP1)$$
$$N, y, r\ :\ \left[\begin{array}{l} N_0(p) \neq req \land \\ N = N_0 \oplus \{p \mapsto None\} \land \\ y = val(N_0(p)) \land r = true \\ \qquad \lor \\ N_0(p) = req \land \\ N = N_0 \oplus \{p \mapsto None\} \land r = false \end{array}\right] \qquad (TP2)$$

Fig. 16. Combining *push* and *pop* elimination

the first specification (C or C') adds a request to N, while the second specification (D or D') ensures that this request is no longer present. Thus, the execution is equivalent to $\alpha\ tryPushElimF_p^7\ \beta\ \gamma$, or $\alpha\ tryPopElimF_p^7\ \beta\ \gamma$, respectively.

We have shown that completed executions of successful elimination attempts must occur in matching pairs which can be jointly translated to a $tryPushElim^7$ immediately followed by a $tryPopElim^7$, and that unsuccessful elimination attempts can be translated into the corresponding unsuccessful elimination attempts in $STACK^7$. It therefore follows that any execution of $STACK^8$ can be transformed into an equivalent execution of $STACK^7$, so $STACK^8$ is a valid refinement of $STACK^7$.

5.3. Combining push and pop elimination ($STACK^9$)

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 describing the operation to be attempted, and redefine $tryPushElim$ and $tryPopElim$ to call these versions, as shown in Fig. 16. The second parameter of $tryActive$ and $tryPassive$ is used to return the result when the request is Pop, and otherwise returns a meaningless value (actually $\perp$) which is ignored by the caller.

To specify these procedures, we introduce three auxiliary functions, as shown in Fig. 17:

- opp determines whether two requests are complementary (i.e. one is a $push$ and the other is a pop);
- $done$ maps a waiting request into the corresponding completed request (i.e. $Push(x)$ to $PushDone$ and Pop to $PopDone(y)$), its second argument provides the value to be returned when the first argument is Pop; and
- val returns the value being pushed for a $push$ request and $\perp$ for a pop request.

It is easy to verify that $done$ and val are only applied to waiting requests, i.e. $Push(x)$ or Pop, so these definitions are sufficient.

To show that this step is a valid refinement, we show that we can reduce any occurrence of $tryActive^9$ or $tryPassive^9$ to an equivalent operation of $STACK^8$. Instantiating the first argument and expanding the definitions of opp, $done$ and val, we see that:

- $tryActive^9(Push(x), y)$ is equivalent to $tryActivePush^8(x)\ ;\ y := \perp$, and $y := \perp$ can be discarded as y is local to $tryPushElim^9$.
- $tryActive^9(Pop, y)$ is equivalent to $tryActivePop^8(y)$.

- $tryPassive^9(Push(x), y)$ is equivalent to $tryPassivePush^8(x)$; $y := \bot$, since if $N_0(p)$ is not $Push(x)$, it must be $PushDone$, and $y := \bot$ can be discarded as y is local to $tryPushElim^9$.
- $tryPassive^9(Pop, y)$ is equivalent to $tryPassivePop^8(y)$, since if $N_0(p)$ is not Pop, it must be $PopDone(y)$, where y is the value pushed by the active elimination partner.

Thus, we can transform any execution of $STACK^9$ into an an equivalent execution of $STACK^8$, which shows that $STACK^9$ is a valid refinement of $STACK^8$.

$$
\begin{array}{l}
opp : Request \times Request \to bool \\
\forall\, r, r' : Request \bullet \\
\quad opp(r, r') \Leftrightarrow (\exists\, x : T \bullet \{r, r'\} = \{Push(x), Pop\}) \\
\\
done : Request \times T_\bot \nrightarrow Request \\
\forall\, x : T \bullet done(Push(x), \bot) = PushDone \land done(Pop, x) = PopDone(x) \\
\\
val : Request \nrightarrow T_\bot \\
\forall\, x : T \bullet val(Push(x)) = x \land val(Pop) = \bot
\end{array}
$$

Fig. 17. Functions used in defining $tryActive$

$$
\begin{array}{l}
tryPushElim^{10}(\mathbf{in}\ x : T)\ r : bool \mathrel{\widehat=} \\
\quad \mathbf{var}\ y : T_\bot\ ; \\
\quad r := tryEliminate^{10}(Push(x), y)
\end{array}
\qquad
\begin{array}{l}
tryPopElim^{10}(\mathbf{out}\ y : T_\bot)\ r : bool \mathrel{\widehat=} \\
\quad y := \bot\ ; \\
\quad r := tryEliminate^{10}(Pop, y)
\end{array}
$$

Fig. 18. Combining active and passive elimination (i)

5.4. Combining active and passive elimination ($STACK^{10}$)

We have so far described active and passive elimination as separate actions, because this allowed us to analyse these rôles independently. 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 notice board, 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 should then check 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, while ensuring that at most one of them can succeed. To capture this idea, we will combine $tryActive$ and $tryPassive$, to give a single $tryEliminate$ procedure, and we redefine $tryPushElim$ and $tryPopElim$ again to make a single call on $tryEliminate$, as shown in Fig. 18.

Our initial specification for $tryEliminate$ is just a demonic choice between $tryActive$ and $tryPassive$, i.e. $r := tryActive^{10}(req, y)$ [] $r := tryPassive^{10}(req, y)$, which we now refine to follow the strategy outlined above.

We can refine the specification of $tryActive$ to a sequential composition with $TP1$ as its first component, provided that the second component only succeeds if $N_0(p)$ is still equal to req, and resets $N(p)$ to $None$. We can then move $TP1$ out of the choice in the definition of $tryEliminate$ (since S ; T [] S ; $U \equiv S$; (T [] U)) and expand the remaining demonic choice into a single specification statement.

Following [HSY04], we will assume that an active eliminator only tries one potential elimination partner (q) before giving up. Clearly, we could consider other options, but this choice means that we can move the selection of q out of the specification statement, and introduce an **if** statement testing $opp(req, N_0(q))$ to determine whether to continue with an active elimination attempt or revert to the passive approach. The resulting code, following some simplifications to remove redundant disjuncts in the specifications, is shown in Fig. 19.

The first two disjuncts of $TE4$ are identical to $tryActive^9$, with the selection of q removed and modified to ensure that they are only taken when $N_0(p) = req$ and that $N(p)$ is reset to $None$. The third disjunct of $TE4$ completes a passive elimination if p's request has been completed by another process (as indicated by $N_0(p) \neq req$). Specification $TE5$ is identical to $TP2$, the second part of $tryPassive^9$, and so continues with a passive elimination attempt when the test for $opp(req, N_0(q))$ fails.

$$
\begin{array}{l}
tryEliminate_p^{10}(\textbf{in } req : Request \textbf{ ; out } y : T_\perp)\ r : bool \mathrel{\widehat{=}} \\
\quad \textbf{var } q : \mathcal{P} \textbf{ ;} \\
\quad N : \left[\, N = N_0 \oplus \{p \mapsto req\} \,\right] \textbf{ ;} \qquad (TE1) \\
\quad q : \left[\, q \in \mathcal{P} \,\right] \textbf{ ;} \qquad (TE2) \\
\quad \textbf{if } opp(req, N(q)) \textbf{ then} \qquad (TE3) \\
\qquad N, y, r : \left[\begin{array}{l}
N_0(p) = req \wedge opp(req, N_0(q)) \wedge \\
N = N_0 \oplus \{p \mapsto None, q \mapsto done(N_0(q), val(req))\} \wedge \\
y = val(N_0(q)) \wedge r = true \\
\qquad \vee \\
N_0(p) = req \wedge N = N_0 \oplus \{p \mapsto None\} \wedge r = false \\
\qquad \vee \\
N_0(p) \neq req \wedge N = N_0 \oplus \{p \mapsto None\} \wedge \\
y = val(N_0(p)) \wedge r = true
\end{array}\right] \qquad (TE4) \\
\quad \textbf{else} \\
\qquad N, y, r : \left[\begin{array}{l}
N_0(p) \neq req \wedge N = N_0 \oplus \{p \mapsto None\} \wedge \\
y = val(N_0(p)) \wedge r = true \\
\qquad \vee \\
N_0(p) = req \wedge N = N_0 \oplus \{p \mapsto None\} \wedge r = false
\end{array}\right] \qquad (TE5) \\
\quad \textbf{fi}
\end{array}
$$

Fig. 19. Combining active and passive elimination (ii)

Statements $TE1$ and $TE2$ can be performed in either order—doing $TE1$ first gives greater opportunity for this request to be selected for elimination by another process.

To confirm that this step is a valid refinement, we show that any trace of $STACK^{10}$ can be transformed into an equivalent trace of $STACK^9$. Suppose t is a trace of $STACK^{10}$ containing a completed $tryEliminate$ by process p. We need to consider five cases, corresponding to the disjuncts in $TE4$ and $TE5$.

If $opp(req, N(q))$ succeeds, trace t is of the form: $\alpha\ TE1_p\ \beta\ TE2_p\ \gamma\ TE3S_p\ \delta\ TE4_p\ \epsilon$.

In the first 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—they may modify N, but do not change $N(p)$. Therefore, since $TE2$ only has local effect, trace t is equivalent to $\alpha\ \beta\ \gamma\ \delta\ TE1_p\ TE2_p\ TE3S_p\ TE4_p\ \epsilon$, which is equivalent to $\alpha\ \beta\ \gamma\ \delta\ tryActiveS_p^9\ \epsilon$.

The second disjunct of $TE4_p$ corresponds to the second disjunct of $tryActive$. By similar reasoning, in this case, trace t is equivalent to $\alpha\ \beta\ \gamma\ \delta\ TE1_p\ TE2_p\ TES3_p\ TEF4_p\ \epsilon$, which is equivalent to $\alpha\ \beta\ \gamma\ \delta\ tryActiveF_p^9\ \epsilon$.

In the third 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. Since $TE1$ is the same as $TP1$ and the third disjunct of $TE4$ is the same as the first disjunction of $TP2$, trace t is equivalent to $\alpha\ TP1_p\ \beta\ \gamma\ \delta\ TP2S_p\ \epsilon$, which contains a successful $tryPassive^9$.

If $opp(req, N(q))$ fails, trace t is of the form: $\alpha\ TE1_p\ \beta\ TE2_p\ \gamma\ TE3F_p\ \delta\ TE5_p\ \epsilon$. Since $TE5$ does not depend on the outcomes of $TE2$ and $TE3$, they can be ignored, and $TE1$ and $TE5$ are equivalent to $TP1$ and $TP2$. So trace t is equivalent to $\alpha\ TP1_p\ \beta\ \gamma\ \delta\ TP2_p\ \epsilon$, and so contains a completed $tryPassive_p^9$, which succeeds iff $TE5$ succeeds.

Thus, we have established that $STACK^{10}$ is a valid refinement of $STACK^9$.

5.5. A concrete elimination mechanism

To implement the elimination mechanism described above, we must introduce a concrete data structure to represent the notice board, which allows the required operations to be implemented efficiently in a lock-free fashion. As in the lock-free stack implementation in Sect. 4, we will use CAS to update shared variables, using local snapshots to detect interference, and allow operations to fail when interference is detected.

$\textbf{type}\ ProcId \mathrel{\widehat{=}} 1\,..\,NumProc$ $\qquad$ $\textbf{var}\ ops : \textbf{array}\ [ProcId]\ \textbf{of}\ REQUEST$

$\textbf{type}\ OP \mathrel{\widehat{=}} PUSH \mid POP \mid NONE$ $\qquad$ $\textbf{initially}\ \forall\, p : ProcId \bullet ops[p].op = NONE$

$\textbf{type}\ REQUEST \mathrel{\widehat{=}} (op : OP\ ;\ val : T_\perp)$

Fig. 20. Elimination stack data structures

5.5.1. *Data refinement* ($STACK^{11}$)

Since N is a finite function, we can implement it as an array, ops, of requests, indexed by process identifiers of type $ProcId$, which is an integer subrange whose values denote processes in $\mathcal{P}$, so $ProcId \mathrel{\widehat{=}} 1\,..\,NumProc$, where $NumProc$ is the number of processes. Requests are represented by labelled pairs of type $(op : OP,\ val : T_\perp)$, where op is either $PUSH$, POP or $NONE$, indicating what operation (if any) the process is attempting to perform, and val is the value to be pushed for a waiting $push$ request, the value to be returned for a completed pop request, and is disregarded otherwise. Initially, $ops[p].op = NONE$ for all p; the initial values of $ops[p].val$ are immaterial. The relevant declarations are shown in Fig. 20.

At this point, we observe that the difference between a completed request and the absence of a request is only significant to the process that posts the request, and once a process posts a request the only way it will be changed by another process is to mark it as completed. Therefore, it is not necessary to distinguish between a completed request and the absence of a request—we just need to be able to access the value returned by a completed pop request.

The relationship between values r of type $Request$ and r' of type $REQUEST$ is given by the following relation:

$$\begin{aligned} Rep(r, r') \mathrel{\widehat{=}} \forall v : T_\perp \bullet\ & (\ r = Push(v)\ \Rightarrow\ r'.op = PUSH\ ) \wedge \\ & (\ r = Pop\ \Rightarrow\ r'.op = POP\ ) \wedge \\ & (\ r \in \{None, PushDone, PopDone\}\ \Rightarrow\ r'.op = NONE\ ) \wedge \\ & (\ r \in \{Push(v), PopDone(v)\}\ \Rightarrow\ r'.val = v\ ) \end{aligned}$$

and the relationship between the abstract notice board N and its array representation ops (blurring the distinction between a process and its id, to avoid the notational overhead of adding a mapping between them) is given by:

$$Rep(N, ops) \mathrel{\widehat{=}} \forall p : ProcId \bullet Rep(N(p), ops[p])$$

We can now calculate the data refinement of $STACK^{11}$ using this representation. To construct the data refinement of $STACK^{11}$, we change the interface to $tryEliminate$, so its first argument is of type $REQUEST$, rather than $Request$, and modify the calls in $tryPushElim$ and $tryPopElim$ to pass $(PUSH, x)$ instead of $Push(x)$ and $(POP, \perp)$ instead of Pop, respectively. In the body of $tryEliminate$, we replace N by ops and $N(p)$ by $ops[p]$. We write $A[k := z]$ to denote the result of replacing element k of array A by z, and write $(NONE, ?)$ to denote a value of type $REQUEST$ with $NONE$ as its op field when we do not care what value is in the val field. Thus, $N = N_0 \oplus \{p \mapsto None\}$ becomes $ops = ops_0[p := (NONE, ?)]$. We also rewrite $done(req, v)$ as $(NONE, v)$ and $val(req)$ as $req.val$, and define $opp(req, req')$ as $req.op \neq req'.op \wedge req'.op \neq NONE$ (this will only ever be used when $req.op$ is not $NONE$, so we do not need to check that case). The resulting code is shown in Fig. 21.

As with our other data refinements, each "abstract" action ($TE1$–5 in Fig. 19) is replaced by a single "concrete" action ($TE1'$–$5'$ in Fig. 21). We can thus translate any execution of $tryElimination^{11}$ into an equivalent execution of $tryElimination^{10}$ by making a one-to-one substitution, and there is no possibility of introducing interference.

It follows that $STACK^{11}$ is a valid refinement of $STACK^{10}$.

5.5.2. *Algorithmic refinement* ($STACK^{12}$)

We will now consider how to refine $TE1'$ to $TE5'$ to code. In doing this, we will assume that a $REQUEST$ value is stored in a single (or perhaps double) word, and so can be tested and updated atomically using a CAS. When we do not care what the val field is, we will leave it unchanged.

$$
\begin{array}{l}
tryEliminate_p^{11}(\textbf{in } pinfo : REQUEST \textbf{ ; out } y : T_\perp)\ r : bool \mathrel{\widehat=} \\
\quad \textbf{var } q : ProcId \textbf{ ;} \\
\quad ops : [\, ops = ops_0[p := pinfo] \,] \textbf{ ;} \qquad (TE1') \\
\quad q : [\, q \in ProcId \,] \textbf{ ;} \qquad (TE2') \\
\quad \textbf{if } opp(pinfo, ops[q]) \textbf{ then} \qquad (TE3') \\
\qquad \left.\begin{array}{l} ops, \\ y, \\ r \end{array}\right\} : \left[\begin{array}{l}
ops_0[p] = pinfo \wedge opp(pinfo, ops_0[q]) \wedge \\
ops = ops_0[p := (NONE, ?), q := (NONE, pinfo.val)] \wedge \\
y = ops_0[q].val \wedge r = true \\
\qquad \vee \\
ops_0[p] = pinfo \wedge ops = ops_0[p := (NONE, ?)] \wedge \\
r = false \\
\qquad \vee \\
ops_0[p] \neq pinfo \wedge ops = ops_0[p := (NONE, ?)] \wedge \\
y = ops_0[p].val \wedge r = true
\end{array}\right] \qquad (TE4') \\
\quad \textbf{else} \\
\qquad \left.\begin{array}{l} ops, \\ y, \\ r \end{array}\right\} : \left[\begin{array}{l}
ops_0[p] \neq pinfo \wedge ops = ops_0[p := (NONE, ?)] \wedge \\
y = val(ops_0[p]) \wedge r = true \\
\qquad \vee \\
ops_0[p] = pinfo \wedge \\
ops = ops_0[p := (NONE, ?)] \wedge r = false
\end{array}\right] \qquad (TE5') \\
\quad \textbf{fi}
\end{array}
$$

Fig. 21. Data refinement of *tryEliminate*

Specification $TE1'$ is easily encoded as an assignment, and $TE2'$ can be written as $q :=?$, which assigns an arbitrary value to q. We simplify $TE3'$ by defining a function opp from OP to OP such that $opp(PUSH) = POP$, $opp(POP) = PUSH$ and $opp(NONE) = NONE$.

The first two disjuncts of $TE4'$ must change the op field of p's request to $NONE$, provided that it has not already been changed. We therefore use $CAS(ops[p], pinfo, (NONE, pinfo.val))$ to attempt to update $ops[p]$.

If this CAS succeeds, we wish update $ops[q]$, provided that $opp(pinfo, ops[q])$ still holds. To check this, we declare a new variable $qinfo$ in $tryEliminate$ and assign $ops[q]$ to it immediately after selecting a value for q. We then use $qinfo$ in place of $ops[q]$ in the test at $TE3$, and attempt the update using $CAS(ops[q], qinfo, (NONE, pinfo.val))$. If this CAS succeeds, we set y to $ops[p].val$ and r to $true$, to complete the first disjunct of $TE4'$; otherwise, we set r to $false$, to complete the second disjunct of $TE4'$

If the CAS on $ops[p]$ fails, we know that $ops_0[p].op$ is already equal to $NONE$, since that is the only way it can be changed by another process. In this case, we just need to set y to $ops[p].val$ and r to $true$, to complete the third disjunct of $TE4'$. Setting y can be done safely after the CAS, since no other process can alter $ops[p].val$ once $ops[p].op$ has been set to $NONE$.

The second conjunct of $TE5'$ needs to set the op field of p's request to $NONE$, provided that it has not already been changed. Again, we attempt to do this using $CAS(ops[p], pinfo, (NONE, pinfo.val))$. If this CAS succeeds, we just need to set r to $false$ to complete the second conjunct of $TE5$. If the CAS fails, we have the same situation as in the third conjunct of $TE4'$. We know that p's request has been completed and $ops_0[p].op$ has been set to $NONE$, so it remains to set y to $ops[p].val$ and r to $true$ to complete the first disjunct of $TE5'$.

The resulting code is show in Fig. 22, where we have added a $delay$ statement in the second branch to indicate where this would appear in a practical implementation. We have also negated the final CAS so that the order of cases is preserved.

To show that this is a valid refinement, we show that any trace of $STACK^{12}$ can be transformed into an equivalent trace of $STACK^{11}$. To do this, we have to show that any completed execution of $tryEliminate^{12}$ by process p can be translated into an equivalent execution of $tryEliminate_p^{11}$.

$tryEliminate_p^{12}$(**in** $pinfo : REQUEST$; **out** $y : T_\perp$) $r : bool \mathrel{\widehat{=}}$
var $q : ProcId$; $qinfo : REQUEST$;
$ops[p] := pinfo$;
$q :=?$;
$qinfo := ops[q]$;
if $pinfo.op = opp(qinfo.op)$ **then**
  if $CAS(ops[p], pinfo, (NONE, pinfo.val))$ **then** $(CAS1)$
    if $CAS(ops[q], qinfo, (NONE, pinfo.val))$ **then** $(CAS2)$
      $y := qinfo.val$;
      $r := true$ (1)
    else
      $r := false$ (2)
    fi
  else
    $y := ops[p].val$;
    $r := true$ (3)
  fi
else
  $delay$;
  if $\neg\ CAS(ops[p], pinfo, (NONE, pinfo.val))$ **then** $(CAS3)$
    $y := ops[p].val$;
    $r := true$ (4)
  else
    $r := false$ (5)
  fi
fi

Fig. 22. Algorithmic refinement of *tryEliminate*

The assignments $ops[p] := pinfo$ and $q :=?$ are replaced by $TE1'_p$, $TE2'_p$, respectively. The assignment $qinfo := ops[q]$ and the test $pinfo.op = opp(qinfo.op)$ are replaced by either $TE3S'_p$ or $TE3F'_p$, according to the outcome of the test, at the position of $qinfo := ops[q]$, since the test only refers to local variables and can be moved over steps of other processes.

It remains to consider the sequences of actions that can occur after the test $pinfo.op = opp(qinfo.op)$. These correspond to the five execution paths leading to assignments to r, which are numbered in Fig. 22. In describing these cases, we will write $TE4'.i$ to denote the ith disjunct of $TE4'$, and similarly for $TE5'$.

- On path (1), p performs a successful active elimination. Since no other process can change $ops[p]$ once p has set its op field to $NONE$, we can move $CAS1$ right, and the assignments to y and r are local, so can also be moved. Thus, path (1) is equivalent to an uninterrupted execution of $TE1'_p$, $TE2'_p$, $TE3S'_p$, $TE4'.1_p$ at the position of $CAS2$.
- On path (2), p's active elimination attempt fails because q's request has been fulfilled since it was checked at $TE3$. Since $CAS2$ fails, it can be moved left to the position of $CAS1$, as can the local assignment to r. Thus, path (2) is equivalent to an uninterrupted execution of $TE1'_p$, $TE2'_p$, $TE3S'_p$, $TE4'.2_p$ at the position of $CAS1$.
- On path (3), p performs a successful passive elimination, detecting the successful elimination just as it is about to attempt an active elimination. Since another process has set $ops[p].op$ to $NONE$, no other process can change $ops[p]$ again, so the actions on this path can all be moved to the position of $CAS1$. Thus, path (3) is equivalent to an uninterrupted execution of $TE1'_p$, $TE2'_p$, $TE3S'_p$, $TE4'.2_p$ at the position of $CAS1$.

- On path (4), p performs a successful passive elimination, detecting the successful elimination after abandoning its active elimination attempt and waiting. By similar reasoning to the previous case, the actions on this path can all be moved to the position of $CAS3$. Thus, path (4) is equivalent to an uninterrupted execution of $TE1'_p$, $TE2'_p$, $TE3S'_p$, $TE5'.1_p$ at the position of $CAS3$.
- On path (5), p's passive elimination attempt fails because its request is unfulfilled. The local assignment to r can be moved to the position of $CAS3$. Thus, path (5) is equivalent to an uninterrupted execution of $TE1'_p$, $TE2'_p$, $TE3S'_p$, $TE5'.2_p$ at the position of $CAS3$.

To show that the implementation is lock-free, we need to show that an operation cannot continually try to eliminate, and always fail, 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 non-deterministic 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 [MNSS05]) which must then be shown to have the desired property.

5.6. Selecting a potential elimination partner

We still have to determine how a process p selects another process q as a potential elimination partner. An interesting aspect of this algorithm is that, from the point of view of linearisability, it does not matter how q is chosen! Thus, we could just choose q to be an arbitrary value in $ProcId$ and inspect $ops[q]$ to see if q is attempting to perform a complementary operation, as we have done in Fig. 22. In order to get good performance, however, we would like to choose q in a way that makes it likely that q is attempting to perform a complementary operation. We could search through ops looking for a suitable candidate, but this imposes a linear cost which is too expensive.

The approach taken in [HSY04] uses a second array, called $collision$, which is indexed by integers and whose values are process ids. The size of the $collision$ array does not affect the correctness of the algorithm, but will affect its performance [HSY04] suggest dynamically adjusting the size of the array to optimise performance under varying workloads; we will assume an arbitrary fixed size, M, for $collision$. The initial contents of $collision$ also do not matter, for example, all locations may be initialised to the same value.

A process attempting an elimination chooses an arbitrary location in $collision$, takes the process whose id is in that location as its potential partner, q, and writes its own id into that location, making this process available for other processes to select for elimination. A process never removes its id from $collision$, so if it makes several elimination attempts, its id may occur in several locations in $collision$. This means that overwriting q does not stop another process selecting q as a potential partner, but does reduce the likelihood of this happening. When q is read from the selected element of $collision$, we know that q has at some stage attempted an elimination—unless this is the first time that this element of $collision$ has been selected, but that just means that the elimination will probably fail and will only happen once for each location in $collision$.

Formally, we introduce the $collision$ array as a degenerate data refinement in which no constraints are placed on $collision$ apart from its type, which ensures that its elements are valid process ids, and data refine $q :=?$ to:

$$q, collision : \left[\begin{array}{l} \exists\, pos : 1 \,..\, M \bullet \\ \qquad collision_0[pos] = q \land collision = collision_0[pos := p] \end{array} \right]$$

This can now be refined operationally to introduce pos as a local variable, select its value, and then save the value of $collision[pos]$ into q and replace it by p. We will implement the selection of pos with a non-deterministic assignment, $pos :\in 1 \,..\, M$, which assigns pos an arbitrary value in the specified range, which is all that is required to establish correctness; [HSY04] call an unspecified function, $GetPosition$, which may be tailored to implement a strategy intended to improve performance. Saving and replacing the value of $collision[pos]$ could be implemented very simply using two assignments ($q := collision[pos]$ and $collision[pos] := p$), however, [HSY04] use a retry loop with a CAS to do this—presumably because, if $collision[pos]$ has changed, the more recent value is more likely to be the id of a process currently attempting to eliminate. This code is incorporated in Fig. 24.

To verify this step, we show that any completed execution of these statements is equivalent to an execution of $q :=?$. As usual, we discard failed executions of the loop, since they have no observable effect. We can then move $pos :\in 1 \,..\, M$, because it only involves local variables, and we can move $q := collision[pos]$ to the position of the

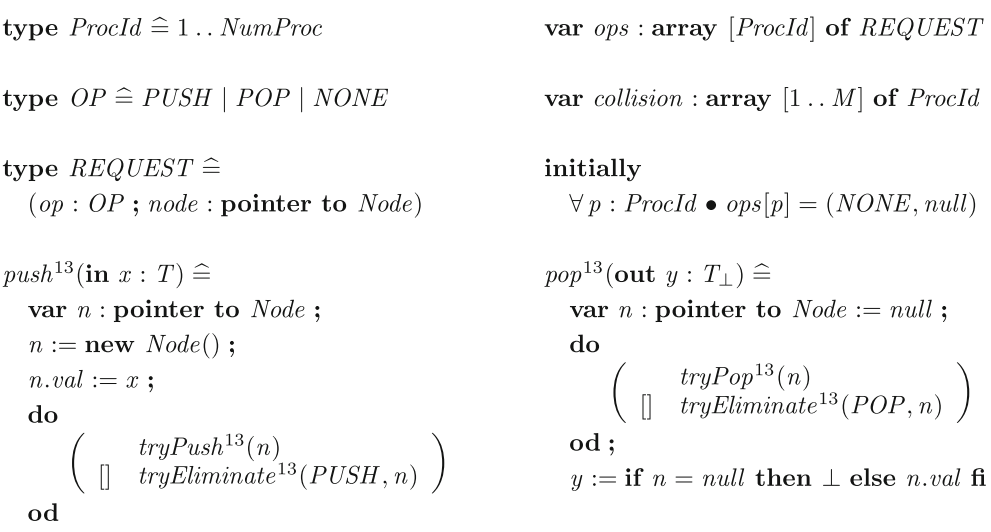

Fig. 23. Final implementation (i)

remaining successful CAS, because we know it will read the same value from $collision[pos]$ (because the CAS succeeds), and it does not matter whether $collision[pos]$ has changed in the meantime, so there is no ABA problem. Thus, we have $pos :\in 1 \,..\, M$, $q := collision[pos]$ and $CAS(collision[pos], q, p)$ executed without interruption, which correctly implements the above specification.

In order to show that the elimination algorithm is lock-free, we have to show that a process cannot continue forever in the retry loop used to save and replace the value of $collision[pos]$ with no other process completing an operation. To show that this cannot happen, we observe that if the CAS in this loop fails an infinite number of times, it must be because other processes perform this CAS successfully an infinite number of times and therefore (because we have a finite number of processes) complete their elimination attempt. Thus, provided that a process cannot continually try (unsuccessfully) to eliminate without ever trying to perform its operation on the central stack (as discussed at the end of Sect. 5.5), the lock-freedom of the elimination algorithm follows from the lock-freedom of the central stack implementation.

5.7. Specialising the implementation for a linked-list central stack ($STACK^{13}$)

Our implementation so far assumes that a CAS can operate on a record containing a value of type $T_\perp$ along with an OP value, which will be the case, say, if T is a 32-bit integer and CAS operates on 64-bit words. If T is a large value, however, this will not be possible, so we would need to replace the val field in a $REQUEST$ by a pointer. This could just be a pointer to the required T value—however, since the final versions of $push$ and pop in Sect. 4 (Figs. 10, 13) already handle nodes containing T values, it is convenient instead to use pointers to such nodes.

Thus, we modify the definition of $REQUEST$ so that its second element is a pointer to a $Node$ (Fig. 6). We also change the interface to $tryEliminate$ so that its first parameter is an OP, and its second parameter is a pointer to a $Node$ as we did to $tryPush$ and $tryPop$ in Fig. 10. The latter is now an **in out** parameter, and is used both to provide the value to be pushed when the first argument is $PUSH$ and to return the value popped when the first argument is POP. The resulting version of $tryEliminate$ is shown in Fig. 24, and the revised versions of $push$ and pop are shown in Fig. 23, where $tryPush^{13}$ and $tryPop^{13}$ are the same as $tryPush^5$ and $tryPop^5$ (Fig. 10) or $tryPush^6$ and $tryPop^6$ (Fig. 13) according to the assumptions we wish to make about memory management.

Apart from the expansion of $q :=?$, which we have already discussed, $STACK^{13}$ (as shown in Figs. 23 and 24, along with the appropriate versions of $tryPush$ and $tryPop$) is simply a data refinement, in which there is a one to one correspondence between atomic actions in the two versions. To show that this refinement is correct, it is easiest

$tryEliminate_p^{13}$(**in** $op : OP$; **in out** n : **pointer to** $Node$) $r : bool \mathrel{\widehat{=}}$
var $pos : int$; $q : ProcId$; $pinfo, qinfo : REQUEST$;
$pinfo := (op, n)$;
$ops[p] := pinfo$;
$pos :\in 1 \,..\, M$;
do
  $q := collision[pos]$;
  $CAS(collision[pos], q, p)$
od ;
$qinfo := ops[q]$;
if $qinfo.op = opp(op)$ **then**
  if $CAS(ops[p], pinfo, (NONE, n))$ **then** (CAS1)
    if $CAS(ops[q], qinfo, (NONE, n))$ **then** (CAS2)
      $n := qinfo.node$;
      $r := true$ (1)
    else
      $r := false$ (2)
    fi
  else
    $n := ops[p].node$;
    $r := true$ (3)
  fi
else
  $delay$;
  if $\neg\ CAS(ops[p], pinfo, (NONE, n))$ **then** (CAS3)
    $n := ops[p].node$;
    $r := true$ (4)
  else
    $r := false$ (5)
  fi
fi

Fig. 24. Final implementation (ii)

to introduce an intermediate version in which the **in out** parameter n is replaced by an **in** parameter, n_{in} and an **out** parameter n_{out}. All occurrences of n in $tryEliminate^{13}$ become n_{in}, except the three occurrences on the left of assignment statements, which become n_{out}. We then observe that the value of $n_{in}.val$ in $tryEliminate^{13}$ is the same as $pinfo.val$ in $tryEliminate^{12}$, and so (op, $n_{in}.val$) in $tryEliminate^{13}$ is the same as $pinfo$ in $tryEliminate^{12}$, and since no process can alter $n_{in}.next$ while $tryEliminate^{13}$ is running, operations involving n_{in} in $tryEliminate^{13}$ are equivalent to operations involving $pinfo.val$ in $tryEliminate^{12}$. Similarly, we show that the value of y in $tryEliminate^{12}$ is the same as $n_{out}.val$ in $tryEliminate^{13}$. A simple data flow analysis will then demonstrate that n_{in} is never read after n_{out} has been assigned, so we are able to combine the two parameters. The alternative to this two stage proof is to use an abstract relation requiring that $n.val$ in $tryEliminate^{13}$ be equal to $pinfo.val$ until it is assigned, after which it is equal to y in $tryEliminate^{12}$.

This step introduces a dependency between the data representation used for the central stack, as discussed in Sect. 4, and that used for the elimination mechanism, so we need to revisit the interaction between these components. To show that no interference is introduced, we observe that the node values passed to or returned

from $tryEliminate$, and stored in the $node$ fields of $REQUEST$ values, are all $node$ values that have been allocated in $push$ but are not part of the linked list used to represent the central stack. Thus, the steps of the elimination mechanism still commute with those of the central stack in the same way as before we made this modification.

The resulting code is very similar to that presented in [HSY04]. The most obvious differences are that we use some different notation and have structured the algorithm differently. More importantly, in [HSY04], ops is an array of pointers to $REQUEST$ nodes (which they call $ThreadInfo$), and they remove p's request from the notice board by setting the $ops[p]$ to $null$. Since these nodes must then be allocated dynamically, their algorithm is susceptible to another form of the ABA problem. We avoid this problem by making ops an array of $Request$ nodes and allowing the op field to take on a third value ($NONE$)—this approach also lessens contention, since it does not matter if q completes its operation and begins another one between when $ops[q]$ is read and when it is updated at $CAS2$. A more detailed comparison is presented in [CG07].

6. Conclusions

We have shown how a sophisticated concurrent stack implementation can be derived from an abstract specification in a series of verifiable steps. We have not given fully formal proofs for these steps, but have stated the correctness conditions that need to be established and have attempted to provide sufficient formality for our proofs to be convincing without overwhelming the reader with an excess of technical minutiae.

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 [HSY04]. We have restructured their algorithm in a way that makes a clearer separation between the elimination mechanism and the underlying stack and, 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. We have also identified a number of points at which alternative implementations might be considered.

We have previously verified this algorithm (using three different memory management regimes for the central stack) using simulation between IOAs [CG07]. 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 operational refinement proofs here translate the entire execution of an operation of A at a time (i.e. it uses induction on the number of completed operations, rather than on the number of atomic actions performed).

The derivation approach used here allowed us to present the various stages of our algorithm in a familiar procedural form, rather than as sets of actions or transitions, which we believe would become very hard to follow through the number of versions involved in this derivation. It also allowed us to present a rigorous proof, highlighting the reasons why the algorithm works, with far fewer technical details than would be required for simulation proofs at a similar level of rigour. Although we have previously verified the algorithm using simulation between IOAs, we now understand many subtleties of the algorithm better as a result of this exercise.

In future work, we intend to mechanise a fully formal proof of the derivation presented here using PVS, so that we can make a more direct comparison with our earlier proof. We will also 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 [MNSS05]. We also intend to apply this derivation approach to other concurrent algorithms, such as the queue implementations described in [CG05, MS98]. The latter will allow valuable comparisons with the simulation proof reported in [DGLM04] and the derivation described in [AC05], which is the only other comparable published derivation of a lock-free algorithm that we know of. Finally, we intend to examine more closely the relationships between the trace reduction approach used in our derivation and other formalisms, such as Back's atomicity refinement [Bac89] and Dingel's refinement calculus [Din02], to see if they can be adapted to support this kind of derivation, and to investigate the extent to which the correctness conditions arising from these proofs can be verified using static analysis or model checking techniques.

Acknowledgments

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

References

[AC05] Abrial J-R, Cansell D (2005) Formal construction of a non-blocking concurrent queue algorithm. J Univ Comput Sci 11(5):744–770

[Bac89] Back R-J (1989) A method for refining atomicity in parallel algorithms. In: PARLE'89 conference on parallel architectures and languages Europe, vol 366 of lecture notes in computer science, Eindhoven, the Netherlands. Springer, New York, pp 199–216

[BKS88] Back RJR, Kurki-Suonio F (1988) Distributed cooperation with action systems. ACM Trans Program Lang Syst 10(4):513–554

[BvW94] Back R-J, von Wright J (1994) Trace refinement of action systems. In: International conference on concurrency theory, Uppsala, Sweden, August 22–25, pp 367–384

[BvW98] Back R-J, von Wright J (1998) Refinement calculus: a systematic introduction. Graduate texts in computer science. Springer, New York

[BvW00] Back R-J, von Wright J (2000) Encoding, decoding and data refinement. Formal Asp Comput 12(5):313–349

[CG05] Colvin R, Groves L (2005) Formal verification of an array-based nonblocking queue. In: ICECCS '05: Proceedings of the internation conference on engineering of complex computer systems, New York, NY, USA. ACM Press, New York, pp 92–101

[CG07] Colvin R, Groves L (2007) A scalable lock-free stack algorithm and its verification. In: 5th IEEE international conference on software engineering and formal methods (SEFM 2007), London, UK, 10–14 September 2007. IEEE Computer Society, Los Alamitos, CA, USA, pp 339–348

[COR+95] Crow J, Owre S, Rushby J, Shankar N, Srivas M (1995) A tutorial introduction to PVS. In: Workshop on industrial-strength formal specification techniques, Boca Raton, Florida

[DB03] Derrick J, Boiten E (2003) Relational concurrent refinement. Formal Asp Comput 15(2):182–214

[DGLM04] Doherty S, Groves L, Luchangco V, Moir M (2004) Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig D, Núñez M (eds) FORTE2004: formal techniques for networked and distributed systems, vol 3235 of lecture notes in computer science. Springer, New York, pp 97–114

[DHLM04] Doherty S, Herlihy M, Luchangco V, Moir M (2004) Bringing practical lock-free synchronization to 64-bit applications. In: PODC '04: proceedings of the twenty-third annual ACM symposium on principles of distributed computing, New York, NY, USA. ACM, New York, pp 31–39

[Din02] Dingel J (2002) A refinement calculus for shared-variable parallel and distributed programming. Formal Asp Comput 14(2):123–197

[DMMS01] Detlefs DL, Martin PA, Moir M, Steele GL Jr (2001) Lock-free reference counting. In: Proceedings of the 20th annual ACM symposium on principles of distributed computing, Newport, Rhode Island, USA, August 26–29

[dRE98] de Roever W-P, Engelhardt K (1998) Data refinement model-oriented proof methods and their comparison. Cambridge University Press, London (with the assistance of J. Coenen, K.-H. Buth, P. Gardiner, Y. Lakhnech, F. Stomp)

[FF04] Flanagan C, Freund S (2004) Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proc. 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), Venice, Italy, January 14–16, pp 256–267

[FQ03] Flanagan C, Qadeer S (2003) A type and effect system for atomicity. In: Proc. ACM SIGPLAN conference on programming language design and implementation, San Diego, California, USA, June 9–11, pp 338–349

[FQ05] Freund SN, Qadeer S (2005) Exploiting purity for atomicity. IEEE Trans Softw Eng 31(4):275–291

[GC07] Groves L, Colvin R (2007) Derivation of a scalable lock-free stack algorithm. Electron Notes Theor Comput Sci 187:55–74

[Gro07] Groves L (2007) Reasoning about nonblocking concurrency using reduction. In: ICECCS '07: proceedings of the 12th IEEE international conference on engineering complex computer systems (ICECCS 2007), Washington, DC, USA. IEEE Computer Society, pp 107–116

[Her91] Herlihy M (1991) Wait-free synchronization. ACM Trans Program Lang Syst 13(1):124–149

[HLM03] Herlihy M, Luchangco V, Moir M (2003) Obstruction-free synchronization: double-ended queues as an example. In: ICDCS '03: proceedings of the 23rd international conference on distributed computing systems. IEEE Computer Society, Los Alamitos, CA, USA, p 522

[HLMM05] Herlihy M, Luchangco V, Martin P, Moir M (2005) Nonblocking memory management support for dynamic-sized data structures. ACM Trans Comput Syst 23(2):146–196

[HSY04] Hendler D, Shavit N, Yerushalmi L (2004) A scalable lock-free stack algorithm. In: SPAA 2004: proceedings of the sixteenth annual ACM symposium on parallel algorithms, 27–30 June 2004, Barcelona, Spain, pp 206–215

[HW90] Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492

[Lam88] Lamport L (1988) Control predicates are better than dummy variables for reasoning about program control. ACM Trans Program Lang Syst 10(2):267–281

[Lip75] Lipton RJ (1975) Reduction: a method of proving properties of parallel programs. Commun ACM 18(12):717–721

[LS89] Lamport L, Schneider FB (1989) Pretending atomicity. Technical Report TR89-1005, DEC, SRC

[LV93] Lynch NA, Vaandrager FW (1993) Forward and backward simulations—part I: untimed systems. In: 135, Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, p 35

[LV95] Lynch NA, Vaandrager FW (1995) Forward and backward simulations: I. Untimed systems. Inf Comput 121(2):214–233

[Lyn96] Lynch NA (1996) Distributed algorithms. Morgan Kaufmann, Menlo Park

[MG90] Morgan C, Gardiner PHB (1990) Data refinement by calculation. Acta Informatica 27(6):481–503 (reprinted in [MV93])
[MNSS05] Moir M, Nussbaum D, Shalev O, Shavit N (2005) Using elimination to implement scalable and lock-free fifo queues. In: Proc. 17th annual ACM symposium on parallelism in algorithms and architectures (SPAA 2005), Las Vegas, Nevada, USA. ACM Press, New York, pp 253–262
[Moi97] Moir M (1997) Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 15th annual ACM symposium on the principles of distributed computing, Santa Barbara, CA, pp 219–228
[Mor94] Morgan C (1994) Programming from specifications, 2nd edn. Prentice Hall, Englewood Cliffs
[MS98] Michael MM, Scott ML (1998) Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. J Parallel Distrib Comput 51(1):1–26
[MV93] Morgan C, Vickers T (eds) (1993) On the refinement calculus. Springer, New York
[OG76] Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Informatica 6:319–340
[Spi92] Spivey M (1992) The Z notation: a reference manual, 2nd edn. Prentice-Hall, Englewood Cliffs
[Tre86] Treiber RK (1986) Systems programming: coping with parallelism. RJ5118. Technical report, IBM Almaden Research Center. http://domino.watson.ibm.com/library/cyberdig.nsf/
[WS06] Wang L, Stoller SD (2006) Runtime analysis of atomicity for multithreaded programs. IEEE Trans Softw Eng 32(2): 93–110
[XdRH97] Xu Q, de Roever WP, He J (1997) The rely-guarantee method for verifying shared variable concurrent programs. Formal Asp Comput 9(2):149–174

Received 18 January 2007
Accepted in revised form 12 August 2008 by B.K. Aichernig, E.A. Boiten, M.J. Butler and J. Derrick
Published online 31 October 2008