# Using Refinement Calculus Techniques to Prove Linearizability [1]

Bengt Jonsson

Dept. IT, Uppsala University, Sweden

**Abstract.** Stepwise refinement is a method for systematically transforming a high-level program into an efficiently executable one. A sequence of successively refined programs can also serve as a correctness proof, which makes different mechanisms in the program explicit. We present rules for refinement of multi-threaded shared-variable concurrent programs. We apply our rules to the problem of verifying linearizability of concurrent objects, that are accessed by an unbounded number of concurrent threads. Linearizability is an established correctness criterion for concurrent objects, which states that the effect of each method execution can be considered to occur atomically at some point in time between its invocation and response. We show how linearizability can be expressed in terms of our refinement relation, and present rules for establishing this refinement relation between programs by a sequence of local transformations of method bodies. Contributions include strengthenings of previous techniques for atomicity refinement, as well as an absorption rule, which is particularly suitable for reasoning about concurrent algorithms that implement atomic operations. We illustrate the application of the refinement rules by proving linearizability of Treiber's concurrent stack algorithm and Michael and Scott's concurrent queue algorithm.

**Keywords:** Refinement Calculus, Multi-Threading, Formal Verification, Linearizability

## 1. Introduction

Stepwise refinement is an important method for systematic construction of sequential and concurrent programs: a high-level program is transformed into an efficiently executable one through a sequence of correctness preserving refinement steps. Such a sequence of successively refined programs can also serve as a clarifying proof of correctness, provided that different mechanisms of the program are introduced in a way that makes their rôles in the final program explicit [LT87, Jon94].

The refinement calculus is a formalization of the stepwise refinement approach. It was pioneered by Back

*Correspondence and offprint requests to*: Bengt Jonsson, Dept. IT, Uppsala University, Box 337, SE-751 05, Sweden, e-mail: bengt@it.uu.se

[1] This is a manuscript which will appear as: B. Jonsson. *Using refinement calculus techniques to prove linearizability.* Formal Asp. Comput., 24(4-6):537-554 (2012) DOI 10.1007/s00165-012-0250-7, in a Festschrift on the occasion of the sixtieth birthday of Pofessor Charles Carroll Morgan.  The work was supported by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center

and by Morgan [Bac88, Mor90] for sequential programs, and has since then been further developed for several contexts, including that of parallel and concurrent programs (e.g., [BS89, CM88]). The refinement calculus provides a collection of generally applicable laws or rules for refining a program by replacing a statement by another one, while preserving correctness. Each rule typically embodies some principle, which clarifies the rôle of the replacing statement in the program.

In this paper, we present rules for refinement of concurrent programs, in which an unbounded number of threads interact via shared variables. We are particularly interested in using these rules for programs in which concurrent threads interact by fine-grained synchronization mechanisms, such as fine-grained locking or atomic test-and-set primitives (such as compare-and-swap). Such programs can be found, e.g., in concurrency libraries, such as the Intel Threading Building Blocks or the `java.util.concurrent` package. The fine-grained synchronization makes such programs notoriously hard to get correct, as witnessed by bugs found in published algorithms (e.g., [DDG$^+$04, MS95]).

We will primarily use our rules to refine methods that access the shared state in a multi-threaded program. We therefore define a correctness criterion which is based on sequences of call and return actions on a program's methods. This criterion deviates from the refinement calculus as presented in, e.g., [Bac88, Mor90, Bac89], where the correctness criterion is based on total correctness for sequential programs. An important concern when refining concurrent programs is that of atomicity. Typically, high-level abstract programs employ large atomic statements, which are often refined by combinations of smaller atomic actions. An important purpose of our rules is therefore to establish that a (possibly compound) atomic statement is correctly refined by a non-atomic sequence of statements. Such rules both help to establish correctness, and to clarify how a program's fine-grained synchronization mechanisms respect correctness.

As an application of our framework, we use it for constructing proofs of linearizability for concurrent objects, which implement data structures that can be concurrently accessed by many threads. Algorithms for concurrent objects typically employ fine-grained synchronization. It is important to show that they correctly implement an easily understandable specification. The standard correctness criterion is that of *linearizability* [HW90], which intuitively states that each operation of the concurrent object can be viewed as being performed atomically at some point between its invocation and return. This criterion allows users of the concurrent object to understand its behavior in terms of large atomic actions, without considering the fine-grained synchronization in its implementation.

One of our contributions is to express the criterion of linearizability in terms of our refinement relation. Intuitively, a program is linearizable if it refines a specification which consist of the same program, in which each operation (except for its invocation and return actions) is performed atomically at some point between invocation and return. This way to state the correctness of a concurrent object has the advantage that it is not necessary to provide a separately constructed specification: the specification is simply the atomic version of the concurrent object.

The problem of refining atomic actions in a concurrent program has been considered in the context of the refinement calculus by Back [Bac89], using techniques based on commutativity developed by Lipton [Lip75]. Our framework includes an adaptation of the rules from [Bac89], but also contributes new rules that are significantly more powerful. One contribution is a strengthening of the atomicity refinement rule [Bac89], which allows new forms of loops in the refining sequence of statements. Another contribution is a so-called absorption rule, which establishes that a sequence, consisting of an assignment and a following test, correctly refines the atomic combination of these statements. The absorption rule can be seen as a simple formulation of the ideas behind removal of so-called pure loops by Wang and Stoller [WS05]. As a result, we have developed new and powerful techniques for proving linearizability of concurrent objects. We illustrate these techniques by application to two well-known algorithms for concurrent objects from the literature: Treiber's stack [Tre86] and the concurrent queue by Michael and Scott [MS96]. In order to show how our refinement rules can be used for correctness proofs, we show in detail how linearizability can be proven through a sequence of small and simple refinement steps, which explain the mechanisms that underlie these algorithms. We conjecture that it would be possible to build an automated framework that generates refinement proofs, such as the ones shown in this paper. A main bottleneck is that application of refinement rules may depend on assertions over the program state; such assertions have to be derived by a sufficiently powerful program analysis. For singly linked lists, as in [Tre86, MS96], there are powerful techniques, e.g., [BR06], which in principle are able to perform the needed reasoning about heap structures.

**Related Work** Back [Bac89] has presented a technique for refining atomic actions in a concurrent program in the context of a refinement calculus, using techniques based on commutativity, originally developed by

Lipton [Lip75]. Similar criteria for refining atomic actions have also been developed in several other contexts (e.g., [Lam90]). Our work includes a more powerful version of this technique, which allows loops in the refining sequence of statements.

Several previous works have presented techniques for establishing that parts of a program can be considered as being executed atomically. Freund and Qadeer [FQ04] uses type inference techniques to find atomic sections in a program. The approach is automatic, but too weak to prove correctness of algorithms as considered in this paper. The techniques in the work by Wang and Stoller [WS05] are more powerful, and include techniques specifically designed for programs that synchronize using Compare-and-Swap statements, which are similar in spirit to our cancellation and absorption rules. To simplify the reasoning they employ slightly different basic synchronization primitives (using Load Linked/Store Conditional (LL/SC) instead of Compare and Swap (CAS)), which makes reasoning substantially simpler.

Elmas et al. [EQS+10] show how techniques for atomicity refinement (based on commutation) can be used to simplify linearizability proofs. Proofs in their framework can be checked by an automatic prover, but rely on substantial manual guidance (e.g., introduction of new auxiliary variables). Groves [Gro08, Gro09] use techniques based on atomicity refinement (using commutation) and establishment of simulation relations to verify linearizability of concurrent objects. The approach relies heavily on non-automated operational reasoning, and appears not to be suitable for automated verification.

**Organization of Paper** The paper is organized as follows. In the next section, we introduce our program model, intended to represent the behavior of concurrent objects, and introduce our refinement relation. In Section 3, we show how linearizability can be defined in terms of our refinement relation. In Section 4, we present rules for establishing refinement between programs by transformation of particular statements. We apply the presented framework to Treiber's stack in Section 5, and to Michael and Scott's concurrent queue in Section 6. Section 7 contains conclusions and directions for further work.

## 2. Model of Concurrent Objects

We consider concurrent systems consisting of a collection of sequential threads that interact by invoking operations on shared concurrent objects. In this section, we introduce a program model, intended for representing the behavior of concurrent objects.

Define a *program* $\mathcal{P}$ as consisting of a set of *global variables*, and a set of *methods*. A method consists of a set of *local variables*, including a set of *input parameters*, and a *method body*. Local and global variables may reference cells in a shared *heap*. A method body is a compound sequential statement built from atomic commands using standard control flow constructs. Method execution is terminated by executing a **return** command, which may return a value.

When representing the behavior of a concurrent object in our program model, global variables and the shared heap are used to maintain the persistent state of the object. Methods can be invoked by concurrently executing threads at arbitrary points in time. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method.

In this paper, we assume that (global and local) variables are either *data variables*, which assume values from an infinite domain, denoted by $\mathbb{D}$, or *pointer variables*, which contain references to heap cells. We specialize the presentation for the examples in Sections 5 and 6, and assume that each heap cell has two fields: a *data* field which contains a data value in $\mathbb{D}$, and a *next* field which contains a reference to a heap cell or *null*. In addition, fields can also contain the value $\perp$ (undefined). The initial value of a variable or field is $\perp$ (undefined).

Atomic commands include assignments between data variables, pointer variables, or fields of cells pointed to by a pointer variable. The command **new**($x$) creates a new cell on the heap, and assigns a reference to that cell to the pointer variable $x$. We assume a memory management mechanism, which automatically collects garbage, and also ensures that a newly created cell is fresh, i.e., has not been used before program; this avoids the so-called ABA problem (e.g., [MS96]). The assume command $\{\{g\}\}$ blocks execution of the thread when the guard $g$ (a boolean expression) evaluates to false, and is executed without any effect when $g$ evaluates to true. The compare-and-swap command CAS($x, y, z$) is an atomic command, which first tests whether $x$ and $y$ have the same value; if so, it assigns the value of $z$ to $x$ and returns *true*; if not, it leaves the value of $x$ unchanged and returns *false*. Note that the values of two pointer variables are equal only if they reference the same cell (i.e., contain the same "address"). The atomic construct $[\![ S ]\!]$ lets the compound statement $S$

```
struct Node {
  Data data;
  Node * next
}

void Push(Data d) {
  Node * t, *x;
  new(x);
  x.data := d;
  do {
    t := S;
    x.next := t
  } while ¬CAS(S, t, x);
  return
}

void Init() {
  Node S;
  S := null;
}

Data Pop() {
  Node * t, *x;
  do {
    t := S;
    if (t = null)
      return empty;
    x := t.next
  } while ¬CAS(S, t, x);
  return t.data
}
```

**Fig. 1.** Treiber's concurrent stack.

be executed atomically if this is possible, otherwise the statement $[\![ S ]\!]$ blocks (e.g., if $S$ contains an assume command that would block execution from the current configuration). The **return** command terminates execution of a method, and may also return a value. We represent the execution of a **return** command by assigning the return value to the local variable *retval*, which is assumed to be an implicit local variable of each method. In the initial local state, the value of *retval* is undefined.

An example of a program occurs in Figure 1, which models Treiber's concurrent stack algorithm [Tre86]. The algorithm represents a stack by a linked list, with the top pointed to by a global variable $S$. Initially, the stack is empty, i.e., the value of $S$ is *null*. The stack can be accessed by an arbitrary number of concurrent invocations of *Push* and *Pop* methods. An invocation of the *Push* method inserts a data value at the top. This is done by first creating a new cell, pointed to by $x$, whose *data* field contains the value to be inserted. Thereafter, the method enters a retry loop, in which it remains as long as the compare-and-swap command is unsuccessful, typically because other concurrent threads modify the value of $S$ between the assignment to $t$ and the compare-and-swap command. When the compare-and-swap command succeeds, the local variable $t$ is assigned the value of the global variable $S$, and the *next* field of the cell pointed to by $x$ is assigned the cell pointed to by $t$. Finally, the cell is inserted into the stack by the atomic compare-and-swap operation $\mathrm{CAS}(S, t, x)$. An invocation of *Pop* returns *empty* if the stack is empty, or else removes and returns the data value at the top of the stack. This is done in a retry loop, analogous to that in the *Push* method.

A precise semantics of a program can be given in terms of a transition system. Define a *heap* as a triple $\langle M, Next, Val\rangle$, where $M$ is a finite set of *cells*, where $Next : M \mapsto (M \cup \{null, \bot\})$ maps each cell to either a cell in $M$, the value *null*, or $\bot$ (undefined), and where $Val : M \mapsto (\mathbb{D} \cup \{\bot\})$ maps each memory cell to a data value in $\mathbb{D}$ or to $\bot$. A *valuation* maps each data variable to a data value in $\mathbb{D}$ or to $\bot$, and each pointer variable to either a cell in $M$, the value *null*, or $\bot$. Define a *local configuration* as a pair $\langle \pi, \sigma\rangle$, where $\pi$ is a control location in some method, and $\sigma$ is a valuation of the local variables of that method (including the input parameters and the variable *retval*). Define a *thread configuration* $\Pi$ as a mapping from a set $Dom(\Pi)$ of thread identifiers (which represent the currently active method invocations) to local configurations. Define a *global configuration* as a triple $\gamma = \langle H, \Sigma, \Pi\rangle$, where $H$ is a heap, $\Sigma$ is a valuation of the global variables, and $\Pi$ is a thread configuration. The *initial global configuration* is the configuration that results after execution of the *Init* method from a configuration consisting of an empty heap with no cells, undefined initial values of global variables, and an empty set of active threads.

The dynamic behavior of a program is represented by a set of *labeled transitions*. We omit a detailed account of each command, and just assume that the meaning of commands and atomic statements is given in terms of *computation steps* of a thread. Each computation step represents the execution of a command or atomic statement, which changes the local configuration of the thread, possibly updating the heap and global variables. An *observable label* is a term of form $\texttt{invoke}[i](m, d)$ or $\texttt{return}[i](m, d)$, where $i$ is a thread identifier, where $m$ is a method name, and $d$ is the value of the input or return parameter associated with the invocation or response. We omit the obvious modifications to allow for constant or missing return values (such as *empty*) and methods without input parameters.

A *transition* is a triple of form $\gamma \xrightarrow{l} \gamma'$, where $\gamma$ and $\gamma'$ are global configurations and $l$ is a label, which is either an observable label or $\tau$. A transition of a program can be of one of the following forms.

- A transition of form $\langle H, \Sigma, \Pi\rangle \xrightarrow{\tau} \langle H', \Sigma', \Pi'\rangle$ represents a computation step of a thread $i$ in $Dom(\Pi)$. The global configuration is updated from $\langle H, \Sigma, \Pi\rangle$ to $\langle H', \Sigma', \Pi'\rangle$ to reflect the change to the global configuration in the natural way.
- A transition of form $\langle H, \Sigma, \Pi\rangle \xrightarrow{\texttt{invoke}[i](m,d)} \langle H, \Sigma, \Pi'\rangle$ represents an invocation of a method $m$ with input parameter value $d$, where $i$ is a thread identifier which is not in $Dom(\Pi)$, and where $\Pi'$ is obtained from $\Pi$ by adding $i$ to its domain, and mapping $i$ to the initial local state of $m$.
- A transition of form $\langle H, \Sigma, \Pi\rangle \xrightarrow{\texttt{return}[i](m,d)} \langle H, \Sigma, \Pi'\rangle$ represents a return of a method $m$ with return value $d$. The transition can be performed when $i$ is in $Dom(\Pi)$ and $\Pi(i)(\mathit{retval}) = d$. Then the new thread configuration $\Pi'$ is obtained from $\Pi$ by removing $i$ from its domain.

A *computation* of a program is a finite sequence of form $\gamma_0 \xrightarrow{l_1} \gamma_1 \xrightarrow{l_2} \cdots \xrightarrow{l_n} \gamma_n$, such that $\gamma_0$ is the initial global configuration, and such that $\gamma_{i-1} \xrightarrow{l_i} \gamma_i$ is a transition for $i = 1, \ldots, n$. A *trace* of a program is the sequence of observable labels in a computation of the program, i.e., the sequence of invoke and return labels. We use $\mathcal{T}(\mathcal{P})$ to denote the set of traces of the program $\mathcal{P}$.

**Refinement** From the point of view of threads that invoke operations on an object, the behavior of an object is characterized by the possible sequences of invocations and responses, i.e., by its traces. We therefore define refinement between programs in terms of traces.

**Definition 2.1.** Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two programs with the same method signatures (i.e., names, input parameters, and possible return values). Then $\mathcal{P}_2$ *is refined by* $\mathcal{P}_1$ if $\mathcal{T}(\mathcal{P}_1) \subseteq \mathcal{T}(\mathcal{P}_2)$. □

In other words, $\mathcal{P}_2$ is refined by $\mathcal{P}_1$ if for each computation of $\mathcal{P}_1$, there is a computation of $\mathcal{P}_2$ with the same sequence of invoke and return labels. This definition of refinement reflects the view that the interface of a concurrent object consists in the sequences of invocations and responses that are possible. Note that the refinement relation does not consider requirements on termination of method calls. For instance, a program whose methods do not terminate may refine a program whose methods always terminate. As will be shown in Section 3, this view turns out to be sufficient to characterize linearizability.

## 3. Linearizability

In this section, we show how the refinement relation between programs can express the concept of linearizability [HW90]. Linearizability is a key correctness property for concurrent objects, which states that it should always be possible to view operations on concurrent objects as though they occur atomically, in some sequential order. To quote from [HW90]:

Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

This point between invocation and response of an operation is commonly referred to as the *linearization point*.

We can define linearizability in terms of sequences of invoke and return labels, i.e., traces, as defined just before Definition 2.1. Let us consider such traces in more detail. The sequence of labels in a trace contains a sequence of invokes. Each invoke label is either followed by a later matching return label, or not. In the former case, we say that the invoke is *matched*, in the latter case we say that it is *pending*. A computation is called *atomic* if it contains no unmatched invoke labels, and if for each operation, the transitions associated with this operation occur as follows: first an invocation of the operation, thereafter an arbitrary sequence of transitions of other threads, thereafter all non-observable computation steps of its body (including the assignment of return value to the variable *retval*) without interference of computations steps by other threads, thereafter transitions of other threads, and finally the return. Thus, in an atomic computation, each method body is executed as a single atomic computation step at some point in time between invocation and return. A trace is *atomic* if it is the sequence of observable labels in an atomic computation. According to [HW90], a trace is *linearizable* if it can be transformed into an atomic trace by first appending a sequence of return labels for some of its unmatched invoke labels, and thereafter removing the unmatched invoke labels.

The above discussion can be summarized into the below characterization of linearizability. For a program $\mathcal{P}$, let **atomic**$\{\mathcal{P}\}$ denote the program in which each method body $S$ is replaced by $[\![ S ]\!]$.

**Definition 3.1.** A program $\mathcal{P}$ is *linearizable* if and only if **atomic**$\{\mathcal{P}\}$ is refined by $\mathcal{P}$ □

This proposition captures the intuition that $\mathcal{P}$ is linearizable if in each computation, each operation appears to be performed atomically at some point between its invocation and its response. Note that the definition does not require progress. For instance, a program in which no operation ever returns is trivially linearizable. This is consistent with the definition in [HW90]. Of course, it is of interest to establish that operations eventually return, according to some notion of progress, but this is not within the scope of the current paper.

## 4. Rules for Establishing Refinement between Programs

In this section, we present rules for refining a program by replacing some of its statements. We use $\mathcal{P}[S]$ to denote a program which contains an occurrence of the statement $S$, and $\mathcal{P}[T]$ to denote the result of replacing that occurrence by $T$. Rules provide constraints that guarantee that the replacement is correct, i.e., that $\mathcal{P}[S]$ is refined by $\mathcal{P}[T]$. Rules should preferably be local in the sense that a replacement is correct regardless of the enclosing program, or depend on some easily checkable properties of the enclosing program.

**Definition 4.1.** Let $S$ and $T$ be two statements, and $\mathcal{P}[\cdot]$ be a program context in which $\cdot$ denotes a "place holder" in some method body, which can be filled by a statement. Define $S \sqsubseteq_{\mathcal{P}[\cdot]} T$ to mean that $\mathcal{P}[S]$ is refined by $\mathcal{P}[T]$. Define $S \sqsubseteq T$ to mean that $S \sqsubseteq_{\mathcal{P}[\cdot]} T$ for any program context $\mathcal{P}[\cdot]$. □

It follows that $\sqsubseteq$ is a congruence with respect to the constructors for forming compound statements (sequential composition, iteration, etc.). We use $S \equiv T$ to denote that $S \sqsubseteq T$ and $T \sqsubseteq S$. In the following, we will use the notation $\sqsubseteq$ also for $\sqsubseteq_{\mathcal{P}[\cdot]}$, since the program context will in most cases be clear.

Definition 4.1 does not provide concrete guidance for reasoning about the validity of refinement rules. We therefore present sufficient conditions for $\sqsubseteq$ in terms of sequences of computation steps. For a statement $S$, let $\pi_0^S$ be the initial control location of $S$. For two heaps $H$ and $H'$ and a thread identifier $i$, let $H \simeq_i H'$ denote that $H$ and $H'$ agree on all cells that can be reached from threads other than $i$.

**Proposition 4.2.** Let $S$ and $T$ be statements that are associated with the same sets of local and global variables. Then $S \sqsubseteq T$ if the following two conditions are satisfied.

1. Whenever $\langle H_0, \Sigma_0, \Pi_0^T \rangle \xrightarrow{l_1} \cdots \xrightarrow{l_n} \langle H, \Sigma, \Pi \rangle$ is a finite sequence of transitions of some program with $\Pi_0^T(i) = \langle \pi_0^T, \sigma_0 \rangle$, in which $i$ performs a sequence of computation steps of $T$,

   there is a finite sequence of transitions $\langle H_0, \Sigma_0, \Pi_0^S \rangle \xrightarrow{l_1} \cdots \xrightarrow{l_n} \langle H', \Sigma, \Pi' \rangle$ with the same sequence of observable labels, where $\Pi_0^S(i) = \Pi_0^T[i \mapsto \langle \pi_0^S, \sigma_0 \rangle]$ (i.e., the only difference between $\Pi_0^S$ and $\Pi_0^T$ is the initial control location of $i$), in which $i$ performs a sequence of computation steps of $S$, such that $H \simeq_i H'$ and $\Pi'(i') = \Pi(i')$ for all $i' \neq i$.
2. Whenever $\Pi(i)$ in the above sequence of transitions is at the end of $T$ then $\langle H', \Sigma, \Pi' \rangle$ can be chosen so that $\Pi'(i)$ is at the end of $S$, so that the valuation of local variables of $i$ is the same in $\Pi'$ and $\Pi$, and so that $H' = H$. □

The proposition can be justified from Definition 2.1, by considering the structure of computations. If the criterion above is satisfied, then a sequence of transitions in which the computation steps of $i$ are from $T$ can be replaced by sequence of transitions in which the computation steps of $i$ are from $S$.

The criterion in Proposition 4.2 can be used to motivate the rules for refinement that will be presented subsequently. We first need some preliminary definitions.

- For a local pointer variable $x$, let the assertion *local*$(x)$ denote that the cell referenced by $x$ cannot be reached (by a chain of pointers or `next` fields) from any other thread than that in which $x$ is local. Such assertions will be used to let accesses of form $x.next$ and $x.data$ be treated in the same way as accesses to local variables.
- A *local occurrence* is either an occurrence of a field access (of form $x.next$ or $x.data$), where the variable $x$ occurs in an immediately preceding assume command of form $\{\{local(x)\}\}$, or an occurrence of a local variable.

- A local variable $x$ is *dead* after a statement $S$ in a method body if the next occurrence of $x$ (if any) in the same method body performs an assignment to $x$.
- A field (of form $x.next$ or $x.data$) is *dead* after a statement $S$ in a method body if $S;\{\{local(x)\}\} \sqsubseteq S$, i.e., $local(x)$ holds after $S$, and the next occurrence (if any) of the field in the same method body is a local occurrence which performs an assignment to it.
- A *non-conflicting access* is either an access to a local variable, or a field access (of form $x.field$) which is either (1) a local occurrence, or (2) a read access such that any write access in the program to a field named *field* (of form $y.field$) is a local occurrence.
- Let $C$ and $C'$ be commands or atomic statements (note that commands are by definition atomic). We say that $C'$ *commutes right* with $C$ if for any sequence of two transitions $\langle H_0, \Sigma_0, \Pi_0\rangle \xrightarrow{\tau} \langle H_1, \Sigma_1, \Pi_1\rangle \xrightarrow{\tau} \langle H_2, \Sigma_2, \Pi_2\rangle$, where a thread $i'$ performs $C'$ in the first one, and a different thread $i$ performs $C$ in the second one, then there is another sequence of two transitions $\langle H_0, \Sigma_0, \Pi_0\rangle \xrightarrow{\tau} \langle H_1', \Sigma_1', \Pi_1'\rangle \xrightarrow{\tau} \langle H_2, \Sigma_2, \Pi_2\rangle$, where thread $i$ performs $C$ in the first one, and thread $i'$ performs $C'$ in the second one, and the two sequence begin and end in the same global configurations. We say that $C'$ *commutes left* with $C$ if the swap between $C'$ and $C$ can be performed in the other direction. A sufficient criterion for a command $C'$ to commute left and right with any other command is that $C'$ only contains non-conflicting accesses.
- For commands or atomic statements $C$ and $C'$ in a program, we say that $C'$ *weakly commutes right* with $C$ if for any sequence of two transitions $\langle H_0, \Sigma_0, \Pi_0\rangle \xrightarrow{\tau} \langle H_1, \Sigma_1, \Pi_1\rangle \xrightarrow{\tau} \langle H_2, \Sigma_2, \Pi_2\rangle$, where a thread $i'$ performs $C'$ in the first one, and a different thread $i$ performs $C$ in the second one, then there is another sequence of two transitions $\langle H_0, \Sigma_0, \Pi_0\rangle \xrightarrow{\tau} \langle H_1', \Sigma_1', \Pi_1'\rangle \xrightarrow{\tau} \langle H_2, \Sigma_2, \Pi_2\rangle$, where thread $i$ performs an atomic statement $S$ in the first one, and thread $i'$ an atomic statement $S'$ in the second one, and the two sequence begin and end in the same global configurations, where the statements $S$ and $S'$ can be chosen in one of the following ways.
  - If $C'$ is contained in a statement of form **do** $C'$, then $S'$ can be chosen as $[\![$ **do** $C'$ $]\!]$, otherwise $S'$ is $C'$.
  - If $C$ is contained in a statement of form **do** $C$, then $S$ can be chosen as $[\![$ **do** $C$ $]\!]$,
  - If $C$ is contained in a statement of form **do** $C''$ ; $C$, then $S$ can be chosen as $[\![$ **do** $C''$ ; $C$ $]\!]$.
  - If none of the two previous cases apply, then $S$ is $C$.

  As an example, let $x$ be a global variable, let $C'$ be the command $x$++ , and let $C$ be the command $[\![\ \{\{even(x)\}\}; x\text{++}\ ]\!]$ which increments $x$ only if it is even. Then $C'$ does not commute right with $C$, since if $x$ is 5 (say), then $C'$ can be performed before $C$, but not vice versa (since $C$ blocks when $x$ is 5). On the other hand, there is a situation in which $C'$ weakly commutes right with $C$. Namely, if $C$ occurs in the statement **do** $x$++ ; $C$, and $C'$ occurs in the statement **do** $C'$, then performing $C'$ before $C$ yields the same result as performing $[\![$ **do** $x$++ ; $C$ $]\!]$ (following the third case in the above definition, with $C'$ as $C''$ and performing one iteration in the loop) before $[\![$ **do** $C'$ $]\!]$ (following the first case in the above definition, with zero iterations of the loop).
- We say that $C'$ *weakly commutes left* with $C$ in the analogous manner.

To make rules more widely applicable, the control structure of each method will first be transformed into an equivalent one that uses the control structures **either** $-$ **or**, which nondeterministically selects a branch, and **do** $S$, which performs an arbitrary number (zero or more) of repetitions of the statement S. The transformation will also put relevant control-flow tests into assume commands of form $\{\{g\}\}$. For a non-nested loop, the transformed method will contain a loop of form **do** $S$, where $S$ contains only the non-exiting parts of the loop, followed by a selection between the different ways in which the loop can be exited, using the **either** $-$ **or** control structure. As an example, a control structure of form

$S \triangleq$ **do** $\{S_1;$ **if** $(b_1)$**return** $e; S_2\}$**while** $(b_2)$

will be transformed into the following control structure.

$T \triangleq$ **do** $\{S_1; \{\{\neg b_1\}\}; S_2\{\{b_2\}\}\};$ **either** $\{S_1; \{\{b_1\}\};$ **return** $e\}$**or**$\{S_1; \{\{\neg b_1\}\}; S_2\{\{\neg b_2\}\}\}$

We have $S \equiv T$, since $S$ and $T$ can perform the same sequences of computation steps.

We are now ready to present the rules that will be used. Since we will use the refinement relation $\sqsubseteq$ to start from a concrete program, and prove, by a sequence of gradual refinement steps, that it refines an easily understood specification, we will present the rules with the refining statement to the left of the $\sqsupseteq$ relation.

**Skip** The statement **skip** can be introduced and removed arbitrarily, i.e., $S; \mathbf{skip} \equiv S$ for any statement $S$.

**Thread-Local Refinement** If for any computation step $\langle H_0, \Sigma_0, \Pi_0^T \rangle \xrightarrow{\tau} \langle H, \Sigma, \Pi \rangle$ of $[\![\ T\ ]\!]$ performed by thread $i$, there is a computation step $\langle H_0, \Sigma_0, \Pi_0^S \rangle \xrightarrow{\tau} \langle H, \Sigma, \Pi' \rangle$ of $[\![\ S\ ]\!]$ performed by thread $i$, where the only difference is the initial and final control locations of $i$, then $[\![\ T\ ]\!] \sqsupseteq [\![\ S\ ]\!]$. That is, inside atomic statements, the criteria for refinement between statements need not consider interference from concurrent threads. How to refine sequential statements is well-understood, and techniques can be adapted from, e.g., [Bac88, Mor90]. In the subsequent examples, we will use rather straight-forward rules, mostly propagation of assertions. We will often not distinguish between obviously equivalent atomic statements, and, e.g., identify $[\![\ S; \{\{g\}\}; \{\{g'\}\}\ ]\!]$ with $[\![\ S; \{\{g \wedge g'\}\}\ ]\!]$.

**Annotations** Assume commands of the form $\{\{g\}\}$, where $g$ is a boolean expression over local variables, are introduced in order to introduce information that is necessary for applying other rules. They can be introduced and propagated as follows.

- The rule $\{\{g\}\} \sqsupseteq \mathbf{skip}$ trivially holds for any boolean expression $g$, implying that an assume command $\{\{g\}\}$ can always be introduced when refining a statement.
- If $g$ is guaranteed to be true after any terminating execution of statement $S$ (which can be interleaved with statements of concurrent threads), then $S \sqsupseteq S; \{\{g\}\}$. As a special case, if any terminating execution of $S$ from a state satisfying *pre* will establish *post*, then $\{\{pre\}\}; S \sqsupseteq \{\{pre\}\}; S; \{\{post\}\}$.
- If $S$ assigns only to local occurrences, and any terminating execution of $S$ (which can be interleaved with concurrent threads) that reaches a state satisfying *post* is guaranteed to satisfy *pre* prior to the execution of $S$, then $S; \{\{post\}\} \sqsupseteq \{\{pre\}\}; S; \{\{post\}\}$.
- If $g$ only contains non-conflicting accesses, then $[\![\ S; \{\{g\}\}\ ]\!]; [\![\ T\ ]\!] \sqsupseteq [\![\ S; \{\{g\}\}\ ]\!]; [\![\ \{\{g\}\}; T\ ]\!]$, and $[\![\ S\ ]\!]; [\![\ \{\{g\}\}; T\ ]\!] \sqsupseteq [\![\ S; \{\{g\}\}\ ]\!]; [\![\ \{\{g\}\}; T\ ]\!]$. That is, local assertions $\{\{g\}\}$ can be propagated between atomic statements.

**Locality Annotations** Assertions of form $local(x)$ are introduced by $\mathbf{new}(x) \sqsupseteq \mathbf{new}(x); \{\{local(x)\}\}$. Furthermore, if $S$ does not contain any occurrence of $x$, then $\{\{local(x)\}\}; S \sqsupseteq \{\{local(x)\}\}; S; \{\{local(x)\}\}$. In this way, assertions of form $local(x)$ can be propagated through method bodies.

**Cancellation** The Cancellation rule states that $S; T \sqsupseteq T$ if any variable or field of a heap cell which is updated by $S$ is a local occurrence in $S$ and dead after $S$. The justification for the rule is that the computation steps performed by $S$ have no effect on other threads or the continuation $T$. This rule can be used to remove loops that have no lasting effect on the configuration. This rule can be seen as a simple formulation of the ideas behind removal of so-called pure loops by Wang and Stoller [WS05].

**Absorption** If $v$ is a local variable and $e$ is any (possibly global) expression which does not contain an occurrence of $v$, then

$$v := e; [\![\ \{\{v = e\}\}; S\ ]\!] \sqsupseteq [\![\ v := e; \{\{v = e\}\}; S\ ]\!]$$

The intuitive justification for the Absorption rule is that a computation of $v := e; [\![\ \{\{v = e\}\}; S\ ]\!]$, in which other threads intervene between the assignment $v := e$ and the atomic statement, can be simulated by a computation in which the assignment is redone just before the test $\{\{v = \exp\}\}$: since the test will be successful, the reassignment has no effect. Thereafter the first assignment will be redundant, and can be dropped. The Absorption rule can also be derived using the Cancellation rule, as follows.

$$v := e; [\![\ \{\{v = e\}\}; S\ ]\!] \sqsupseteq v := e; [\![\ v := e; \{\{v = e\}\}; S\ ]\!] \sqsupseteq [\![\ v := e; \{\{v = e\}\}; S\ ]\!]$$

**Atomicity Refinement** Let $S$ and $T$ be statements and $A$ be a command or atomic statement, and let the statement $T$ be guaranteed to terminate. Then $S; A; T \sqsupseteq [\![\ S; A; T\ ]\!]$ if

(1) all atomic statements in $S$ commute right with all atomic statements in the program, and
(2) all atomic statements in $T$ commute left with all atomic statements in the program.

The justification for this rule is that, because of commutativity, a sequence of transitions in which commands in $S$ are interleaved with commands of concurrent threads can be simulated by a sequence

of transitions in which first all the commands of the concurrent threads are performed followed by an execution of $S$. By an analogous argument, the statement $T$ can be executed before potentially interleaving threads (and since it is guaranteed to terminate) without affecting the sequence of observable labels or the resulting configuration. In summary, any computation in which $S; A; T$ is executed non-atomically has the same visible effect as a computation in which $S; A; T$ is executed without interleaving by other threads; hence it can be made atomic.

The above technique for refining atomic statements in a concurrent context was introduced into the refinement calculus by Back [Bac89], and builds on ideas of commutation originally defined by Lipton [Lip75].

**Strengthened Atomicity Refinement** In this paper, we additionally present a strengthening of the atomicity refinement rule, for the case that the program contains statements of form **do** $C$, where $C$ is a command or atomic statement. This strengthening allows requirements (1) and (2) above to be weakened by requiring only weak commutativity instead of commutativity.

This strengthening can be justified as follows. If $C'$ weakly commutes right, then a sequence of transitions where a thread $i'$ performs $m$ repetitions of $C'$ interleaved with commands of other threads, can be simulated by a sequence in which the interleaving commands are moved to the left of the commands $C'$. This simulating sequence may have a different number of contiguous repetitions of $C'$, and may also insert or remove an arbitrary number of commands of interleaving threads. However, the change still results in a sequence of transitions of the program, since $C'$ occurs inside a statement of form **do** $C'$. Analogous arguments justify the addition or removal of commands in the interleaving threads.

As an example, let $x$, $C'$, and $C$ be as in the example following the definition of weak commutativity. Then we can use the strengthened atomicity refinement rule to establish that **do** $C'$ ; $C \sqsupseteq [\![$ **do** $C'$ ; $C$ $]\!]$, i.e., that

$$\textbf{do}\ x{+}{+}\ ;\ [\![\ \{\{even(x)\}\}; x{+}{+}\ ]\!] \quad \sqsupseteq \quad [\![\ \textbf{do}\ x{+}{+}\ ;\ [\![\ \{\{even(x)\}\}; x{+}{+}\ ]\!]\ ]\!]\ .$$

Following the definition of weak commutativity, we established that $C'$ commutes weakly right with $C$ and with itself.

At the end of Section 6, we apply the strengthened atomicity refinement rule in the proof of linearizability of Michael and Scott's concurrent queue.

**Global Invariants** For the application of some rules, we may need to first establish an invariant over the global variables and the heap. Such an invariant is established in the standard way, by checking that (1) it holds upon initialization of the program, and (2) that any command or atomic statement which modifies a global variable or a heap cell maintains the invariant.

## 5. Application to Treiber's Concurrent Stack

In this section, we illustrate how our framework and the refinement rules in Section 4 can be used to establish linearizability of Treiber's concurrent stack [Tre86]. A description of this algorithm appeared in Figure 1. For the proof of refinement, we will first transforms its control structure, as explained in Section 4. We represent a successful CAS$(x, y, z)$ statement as $[\![\ \{\{x = y\}\}; x := z\ ]\!]$, and an unsuccessful CAS$(x, y, z)$ statement as $\{\{x \neq y\}\}$. The resulting methods are shown in Figure 2. As a first step, we insert annotations of form $\{\{local(x)\}\}$ into the *Push* method. Such annotations are inserted between all pairs of commands that follow the **new**$(x)$ command (this conforms to the rule for introducing annotations of form $\{\{local(x)\}\}$) and precede the next-to-last line $[\![\ \{\{S = t\}\}; S := x\ ]\!]$ (it is not possible to propagate the annotation past this statement, since it exposes the cell pointed to by $x$). For space reasons, we do not display the result.

The annotation using $\{\{local(x)\}\}$ allows to conclude that

- accesses of form $x.next$ in the *Push* method are non-conflicting, since they occur after the assertion $\{\{local(x)\}\}$, and that
- accesses of form $t.next$ in the *Pop* method are non-conflicting, since they are read accesses, and the accesses of form $x.next$ in the *Push* method occur after the assertion $\{\{local(x)\}\}$.

We are now ready to present a proof of linearizability of the program as a sequence of refinement steps. We consider each method separately.

**The *Push* method** The relation $\sqsupseteq$ between the *Push* method and its atomic version can be established through a sequence of applications of the refinement rules. We assume that the assertions of form $\{\{local(x)\}\}$

```
void Push(Data d) {
  Node * t, *x;
  new(x);
  x.data := d;
  do {
    t := S;
    x.next := t;
    {{S ≠ t}}
  };
  t := S;
  x.next := t;
  [[ {{S = t}}; S := x ]];
  return
}
```

```
Data Pop() {
  Node * t, *x;
  do {
    t := S;
    {{t ≠ null}};
    x := t.next;
    {{S ≠ t}}
  };
  either {
    t := S;
    {{t = null}};
    return empty
  } or {
    t := S;
    {{t ≠ null}};
    x := t.next;
    [[ {{S = t}}; S := x ]];
    return t.data
  }
```

**Fig. 2.** Treiber's concurrent stack with transformed control structure.

have been inserted, as just explained. We can now perform the following sequence of steps.

```
new(x);                 new(x);              new(x);              new(x);              [[ new(x);           [[ new(x);
x.data := d;            x.data := d;         x.data := d;         x.data := d;            x.data := d;         x.data := d;
do{                     t := S;              t := S;              [[ t := S;           t := S;              do{
  t := S;               x.next := t;         [[ {{S = t}};           {{S = t}};           {{S = t}};           t := S;
  x.next := t;          [[ {{S = t}};           x.next := t;         x.next := t;         x.next := t;         x.next := t;
  {{S ≠ t}}               S := x ]];            {{S = t}};           {{S = t}};           {{S = t}};           {{S ≠ t}}
};                      return;              S := x;              S := x;              S := x;           };
t := S;                                      return ]];           return ]];           return ];            t := S;
x.next := t;                                                                                                x.next := t;
[[ {{S = t}};                                                                                               {{S = t}};
  S := x ]];                                                                                                S := x;
return;                                                                                                     return ];
```

We motivate each of the steps in turn.

1. The **do** statement is removed using the Cancellation rule. This is justified by observing that both $t$ and $x.next$ are dead after the **do** statement.
2. First the scope of the atomic command is enlarged using the rule for atomicity refinement: the **return** command is local, and the command $x.next := t$ is also local (remember that $local(x)$ is implicitly asserted). Thereafter, the assume command $\{\{S = t\}\}$ is propagated to the beginning of the atomic statement, exploiting the fact that it is inside an atomic statement.
3. Using the Absorption rule, the assignment $t := S$ can be performed together with the atomic section. Note how the just inserted assertion $\{\{S = t\}\}$ is exploited to make the rule applicable.
4. The scope of the atomic section is enlarged with the two first commands, which are purely local.
5. The **do** statement is reinserted. This step is trivially correct, since the inserted statement can only add executions

By composing these refinement relations, it is established that the *Push* method refines its atomic version.

**The *Pop* method** Let us now consider the *Pop* method, as shown in Figure 2. Similarly as for the *Push* method, we first observe that the first **do** statement, i.e.,

$$\textbf{do}\{t := S; \{\{t \neq null\}\}; x := t.next; \{\{S \neq t\}\}\} \quad ,$$

can be removed, since $t$ and $x$ are dead after the **do** loop (note that the local variable $x$ does not appear in the first branch of the **either** – **or** statement). For clarity, consider each of the two alternatives of the **either** – **or** statement separately. For the first one, we have

$$\begin{array}{l} t := S; \\ \{\{t = null\}\} \\ \textbf{return } empty \end{array} \quad \sqsupseteq \quad \begin{array}{l} [\![\ t := S; \\ \quad \{\{t = null\}\} \\ \quad \textbf{return } empty\ ]\!] \end{array}$$

using the atomicity refinement rule, with the assignment $t := S$ as the non-local command. For the second one, we can establish the following sequence of refinement steps.

$$\begin{array}{l} t := S; \\ \{\{t \neq null\}\}; \\ x := t.next; \\ [\![\ \{\{S = t\}\}; \\ \quad S := x\ ]\!]; \\ \textbf{return } t.data \end{array} \sqsupseteq \begin{array}{l} t := S; \\ \{\{t \neq null\}\}; \\ [\![\ x := t.next; \\ \quad \{\{S = t\}\}; \\ \quad S := x\ ]\!]; \\ \textbf{return } t.data \end{array} \sqsupseteq \begin{array}{l} t := S; \\ [\![\ \{\{t \neq null\}\}; \\ \quad x := t.next; \\ \quad \{\{S = t\}\}; \\ \quad S := x; \\ \textbf{return } t.data\ ]\!] \end{array} \sqsupseteq \begin{array}{l} t := S; \\ [\![\ \{\{S = t\}\}; \\ \quad \{\{t \neq null\}\}; \\ \quad x := t.next; \\ \quad \{\{S = t\}\}; \\ \quad S := x; \\ \textbf{return } t.data\ ]\!] \end{array} \sqsupseteq \begin{array}{l} [\![\ t := S; \\ \quad \{\{S = t\}\}; \\ \quad \{\{t \neq null\}\}; \\ \quad x := t.next; \\ \quad \{\{S = t\}\}; \\ \quad S := x; \\ \textbf{return } t.data\ ]\!] \end{array}$$

Following is a motivation for each step in turn.

1. The command $x := t.next$ can be moved inside the atomic section, since $x$ is a local variable, and $t.next$ is a non-conflicting access.
2. The atomic section is expanded to include the command **return** $t.data$: note that $t.data$ is a non-conflicting access. The command $\{\{t \neq null\}\}$ is also included into the atomic statement, since $t$ is local.
3. The assertion $\{\{S = t\}\}$ is added to the beginning of the atomic section, similarly as for the *Push* method.
4. By the *Absorption Rule*, the assignment $t := S$ can be performed together with the atomic section.

Following this sequence of refinement steps, we use the obvious rule

$$\textbf{either } [\![\ S\ ]\!] \textbf{ or } [\![\ T\ ]\!] \quad \equiv \quad [\![\ \textbf{either } S \textbf{ or } T\ ]\!]$$

to merge the two alternatives into one atomic statement. Finally, we reinsert the do-loop as was done for the *Push* method.

$$\textbf{do } \{t := S; \{\{t \neq null\}\}; x := t.next; \{\{S \neq t\}\}\}$$

at the beginning of the atomic statement. By composing these refinement relations, it is established that the *Pop* method refines its atomic version.

## 6. Application to Michael and Scott's Concurrent Queue

In this section, we illustrate how our framework and the refinement rules in Section 4 can be used to establish linearizability of Michael and Scott's concurrent queue [MS96]. A description of this queue algorithm appears in Figure 3. The queue is represented by linked list in which the first cell is a dummy cell. The global variable $Head$ points to the dummy cell, and $Tail$ points to the last cell in the list. Initially, the list contains only a dummy cell. An invocation of the *Enqueue* method inserts a new cell at the end of the list and makes $Tail$ point to this new cell. This is done using a retry loop, in analogy with the *Push* method of Treiber's stack. An invocation of *Dequeue* returns *empty* if the queue is empty, or else removes and returns the first value in the queue.

For the proof of refinement, we will first transforms its control structure, as explained at the end of Section 4. The resulting methods are shown in Figure 4. As in the treatment of Treiber's stack, we remove pure loops using the Cancellation rule. The result is shown in Figure 5. Our proof that the program in Figure 5 refines the atomic version of the methods in Figure 4 follows a similar pattern as that for Treiber's stack, but contains additional complications. One is that we must first establish a global invariant, that relates the variables $Head$ and $Tail$ and the heap. Let us introduce some notation.

For two variables $x_1$ and $x_2$, let $x_1 \xrightarrow{*} x_2$ denote that the cell pointed to by $x_2$ can be reached by following a chain of zero or more *next* fields from the cell pointed to by $x_1$. Let $x_1 \xrightarrow{+} x_2$ denote that the

```
struct Node {
  Data data;
  Node *next
}

void Enqueue(Data d) {
  Node *t, *n, *x;
  new(x);
  x.data := d;
  do {
    t := Tail;
    n := t.next;
    if (t = Tail) {
      if (n = null) {
        if (CAS(t.next, n, x)) break;
      } else {
        CAS(Tail, t, n)
      }
    }
  };
  CAS(Tail, t, x));
  return
}

void Init() {
  Node *Head, *Tail, *n;
  new(n);
  Head := n;
  Tail := n;
}

Data Dequeue() {
  Node *h, *t, *n;
  do {
    h := Head;
    t := Tail;
    n := h.next;
    if (h = Head) {
      if (h = t) {
        if (n = null) return empty;
        CAS(Tail, t, n));
      } else {
        retval := n.data;
        if (CAS(Head, h, n)) return retval
      }
    }
  }
}
```

**Fig. 3.** Michael and Scott's concurrent queue

cell pointed to by $x_2$ can be reached by following a chain of one or more *next* fields from the cell pointed to by $x_1$. Reasoning about these relations can be mechanized, e.g., using techniques presented by Bingham and Rakamaric [BR06]. We will only need to reason about these relations using simple principles, e.g., that $x_1 \xrightarrow{*} x_2 \wedge x_1 \neq x_2$ implies $x_1 \xrightarrow{+} x_2$.

The global invariant we will need is that $Head \xrightarrow{*} Tail$, i.e., that the cell pointed to by $Tail$ can be reached by following a chain of zero or more *next* fields from the cell pointed to by $Head$. Let us consider how it is preserved by changes to global variables and to heap cells (we need only consider *next* fields).

1. Initially, $Head = Tail$, which conforms to the invariant.
2. The program contains only one update of a *next* field, in the next-to-last atomic statement of the *Enqueue* body. By propagating properties of local variables and local occurrences, this atomic statement is equivalent to

   $$[\![\ \{\{t.next = n = null \wedge local(x) \wedge x.next = null\}\}; t.next := x\ ]\!]\ ,$$

   showing that the heap is modified only by adding new cells, which are pointed to by pointers from cells that previously had *null* as their *next* field. This implies that the heap maintains the shape of a singly linked list. We also observe that a non-*null next* field is never modified.
3. The program contains three updates of $Tail$. The first one in the *Enqueue* method is equivalent, using the observation that the command $n := t.next$ commutes right with all commands in the program (since $n \neq null$ and the program contains no write-access to a non-*null next*-field) and hence can be moved into the atomic statement, to

   $$[\![\ \{\{t = Tail \wedge n = t.next \wedge n \neq null\}\}; Tail := n\ ]\!]\ ,$$

   which shows that $Tail$ is assigned to $Tail.next$ which is non-*null*. We can draw the same conclusion for the other updates of $Tail$: The last line of the *Enqueue* method is equivalent to

   $$[\![\ \{\{t = Tail \wedge x = t.next \wedge x \neq null\}\}; Tail := x\ ]\!]\ ,$$

```
void Enqueue(Data d) {
  Node *t, *n, *x;
  new(x);
  x.data := d;
  do {
    either {
      t := Tail;
      n := t.next;
      {{t ≠ Tail}}
    } or {
      t := Tail;
      n := t.next;
      {{t = Tail}};
      {{n = null}};
      {{t.next ≠ n}}
    } or {
      t := Tail;
      n := t.next;
      {{t = Tail}};
      {{n ≠ null}};
      [[ if (t = Tail) Tail := n ]]
    }
  };
  t := Tail;
  n := t.next;
  {{t = Tail}};
  {{n = null}};
  [[ {{t.next = n}}; t.next := x ]];
  [[ if (t = Tail) Tail := x ]]
  return
}
```

```
Data Dequeue() {
  Node *h, *t, *n;
  do {
    either {
      h := Head;
      t := Tail;
      n := h.next;
      {{h ≠ Head}}
    } or {
      h := Head;
      t := Tail;
      n := h.next;
      {{h = Head}};
      {{h = t}};
      {{n ≠ null}};
      CAS(Tail, t, n)
    } or {
      h := Head;
      t := Tail;
      n := h.next;
      {{h = Head}};
      {{h ≠ t}};
      retval := n.data;
      {{h ≠ Head}}
    }
  };
  either {
    h := Head;
    t := Tail;
    n := h.next;
    {{h = Head}};
    {{h = t}};
    {{n = null}};
    return empty
  } or {
    h := Head;
    t := Tail;
    n := h.next;
    {{h = Head}};
    {{h ≠ t}};
    retval := n.data;
    [[ {{h = Head}}; Head := n ]];
    return retval
  }
}
```

**Fig. 4.** Michael and Scott's concurrent queue with transformed control structure.

and the update in the loop of *Dequeue* is equivalent to

$$[\![\ \{\{t = Tail \wedge n = h.next \wedge h = t \wedge n \neq null\}\}; Tail := n\ ]\!]\ .$$

4. Let us finally consider the update to $Head$ in the next-to-last line of *Dequeue*. We can assume that $Head \xrightarrow{*} Tail$ at the beginning of the method body. This implies that $h \xrightarrow{*} Tail$ immediately after the preceding assignment $h := Head$ (occurring 6 lines earlier). The property $h \xrightarrow{*} Tail$ is then preserved

```
void Enqueue(Data d) {
  Node *t, *n, *x;
  new(x);
  x.data := d;
  do {
    t := Tail;
    n := t.next;
    {{t = Tail}};
    {{n ≠ null}};
    [[ {{t = Tail}}; Tail := n ]]
  };
  t := Tail;
  n := t.next;
  {{t = Tail}};
  {{n = null}};
  [[ {{t.next = n}}; t.next := x ]];
  [[ if (t = Tail) Tail := x ]];
  return
}
```

```
Data Dequeue() {
  Node *h, *t, *n;
  do {
    h := Head;
    t := Tail;
    n := h.next;
    {{h = Head}};
    {{h = t}};
    {{n ≠ null}};
    [[ {{t = Tail}}; Tail := n ]]
  };
  either {
    h := Head;
    t := Tail;
    n := h.next;
    {{h = Head}};
    {{h = t}};
    {{n = null}};
    return empty
  } or {
    h := Head;
    t := Tail;
    n := h.next;
    {{h = Head}};
    {{h ≠ t}};
    retval := n.data;
    [[ {{h = Head}}; Head := n ]];
    return retval
  }
}
```

**Fig. 5.** Michael and Scott's concurrent queue with pure loops removed

throughout the method body, which means that the update to $Head$ is equivalent to

$$[\![\ \{\{h = Head \wedge h \xrightarrow{*} Tail \wedge h \xrightarrow{*} t \wedge h \neq t\}\}; Head := n\ ]\!] \quad .$$

From this, we deduce that $Head$ is assigned to $Head.next$ which is non-*null*, and that $Head \xrightarrow{+} Tail$ just before the assignment.

Taken together, the above steps establish that we can use the global invariant $Head \xrightarrow{*} Tail$ when reasoning about the algorithm. In a similar way, we can also establish that after an assignment of form $t := Tail$ or an assertion $\{\{t = Tail\}\}$, the invariant $t \xrightarrow{*} Tail$ holds until $t$ is modified, and that after an assignment of form $h := Head$, the invariant $h \xrightarrow{*} Head$ holds until $h$ is modified.

**The *Enqueue* Method** We will now consider the five parts of the program in Figure 5, and see how they can be considered atomic. We begin with the body of the **do** loop in *Enqueue*.

```
t := Tail;
n := t.next;
{{t = Tail}};
{{n ≠ null}};
[[ {{t = Tail}};
   Tail := n ]]
```

$\sqsupseteq$

```
t := Tail;
n := t.next;
[[ {{n ≠ null}};
   {{t = Tail}};
   Tail := n ]]
```

$\sqsupseteq$

```
t := Tail;
[[ n := t.next;
   {{n ≠ null}};
   {{t = Tail}};
   Tail := n ]]
```

$\sqsupseteq$

```
t := Tail;
[[ {{t = Tail}};
   n := t.next;
   {{n ≠ null}};
   {{t = Tail}};
   Tail := n ]]
```

$\sqsupseteq$

```
[[ t := Tail;
   {{t = Tail}};
   n := t.next;
   {{n ≠ null}};
   {{t = Tail}};
   Tail := n ]]
```

Following is a motivation for each step in turn.

1. The assertion $n \neq null$ only concerns the local variable $n$. Thereafter, we remove the assume command $\{\{t = Tail\}\}$ (assume commands can always be removed).
2. The assignment $n := t.next$ can be moved inside the atomic section, since it commutes right with all commands (follows from the observations that $n \neq null$ and that the program contains no write-access to a non-*null* *next*-field), and hence can be moved into the atomic statement.
3. The assertion $\{\{t = Tail\}\}$ can be propagated backwards inside the atomic section.
4. By the Absorption rule, the assignment $t := Tail$ can be performed together with the atomic section.

Let us consider the next segment

$$\begin{array}{l} t := Tail; \\ n := t.next; \\ \{\{t = Tail\}\}; \\ \{\{n = null\}\}; \\ [\![\ \{\{t.next = n\}\}; \\ \quad t.next := x\ ]\!]; \end{array} \sqsupseteq \begin{array}{l} t := Tail; \\ n := t.next; \\ \{\{t = Tail\}\}; \\ [\![\ \{\{n = null\}\}; \\ \quad \{\{t.next = n\}\}; \\ \quad t.next := x\ ]\!]; \end{array} \sqsupseteq \begin{array}{l} t := Tail; \\ n := t.next; \\ [\![\ \{\{t = Tail\}\}; \\ \quad \{\{n = null\}\}; \\ \quad \{\{t.next = n\}\}; \\ \quad t.next := x\ ]\!]; \end{array} \sqsupseteq \begin{array}{l} t := Tail; \\ [\![\ n := t.next; \\ \quad \{\{t = Tail\}\}; \\ \quad \{\{n = null\}\}; \\ \quad \{\{t.next = n\}\}; \\ \quad t.next := x\ ]\!]; \end{array} \sqsupseteq \begin{array}{l} [\![\ t := Tail; \\ \quad n := t.next; \\ \quad \{\{t = Tail\}\}; \\ \quad \{\{n = null\}\}; \\ \quad \{\{t.next = n\}\}; \\ \quad t.next := x\ ]\!]; \end{array}$$

Following is a motivation for each step in turn.

1. The assertion $n = null$ only concerns the local variable $n$.
2. By the observation that $t \xrightarrow{*} Tail$ holds after $\{\{t = Tail\}\}$, the assertion $\{\{n = null\}\}$ in the atomic statement can be extended to $\{\{n = null \wedge t.next = n \wedge t \xrightarrow{*} Tail\}\}$, which implies $t = Tail$. We therefore include $\{\{t = Tail\}\}$ inside the atomic statement, and thereafter omit the occurrence of $\{\{t = Tail\}\}$ that is outside.
3. Since $t.next = n$ at the entry to the atomic statement, we can include the assignment $n := t.next$ by the Absorption rule.
4. The assertion $\{\{t = Tail\}\}$ now holds at the beginning of the atomic statement, so by the Absorption rule, the assignment $t := Tail$ can be performed together with the atomic section.

**The *Dequeue* Method** Let us next consider the *Dequeue* method. It consists of three parts, each of which begins with the two assignments $h := Head; t := Tail$. After the first assignment, we infer $h \xrightarrow{*} Head$, which is maintained throughout that part. Using the global invariant $Head \xrightarrow{*} Tail$, we infer that the assertion $h \xrightarrow{*} t \xrightarrow{*} Tail$ holds immediately after the second assignment. Furthermore, since $h$ and $t$ are local and not reassigned within the part, the assertion remains valid for a number of commands. We will see how this property is used in the refinement steps below.

Let us consider the body of the **do** loop in the *Dequeue* method.

$$\begin{array}{l} h := Head; \\ t := Tail; \\ n := h.next; \\ \{\{h = Head\}\}; \\ \{\{h = t\}\}; \\ \{\{n \neq null\}\}; \\ [\![\ \{\{t = Tail\}\}; \\ \quad Tail := n\ ]\!] \end{array} \sqsupseteq \begin{array}{l} h := Head; \\ t := Tail; \\ n := h.next; \\ \{\{h = Head\}\}; \\ [\![\ \{\{h = t\}\}; \\ \quad \{\{n \neq null\}\}; \\ \quad \{\{t = Tail\}\}; \\ \quad Tail := n\ ]\!] \end{array} \sqsupseteq \begin{array}{l} h := Head; \\ t := Tail; \\ n := h.next; \\ [\![\ \{\{h = Head\}\}; \\ \quad \{\{h = t\}\}; \\ \quad \{\{n \neq null\}\}; \\ \quad \{\{t = Tail\}\}; \\ \quad Tail := n\ ]\!] \end{array} \sqsupseteq \begin{array}{l} h := Head; \\ t := Tail; \\ [\![\ n := h.next; \\ \quad \{\{h = Head\}\}; \\ \quad \{\{h = t\}\}; \\ \quad \{\{n \neq null\}\}; \\ \quad \{\{t = Tail\}\}; \\ \quad Tail := n\ ]\!] \end{array} \sqsupseteq \begin{array}{l} [\![\ h := Head; \\ \quad t := Tail; \\ \quad n := h.next; \\ \quad \{\{h = Head\}\}; \\ \quad \{\{h = t\}\}; \\ \quad \{\{n \neq null\}\}; \\ \quad \{\{t = Tail\}\}; \\ \quad Tail := n\ ]\!] \end{array}$$

Following is a motivation for each step in turn.

1. The assertions before the atomic command only concern local variables.
2. By combining the inferred properties $h \xrightarrow{*} t \xrightarrow{*} Tail$ and $h \xrightarrow{*} Head$ and the invariant $Head \xrightarrow{*} Tail$ with the assertions in the atomic section (implying $h = t = Tail$), we infer that $h = t = Head = Tail$ at the beginning of the atomic section. We can therefore add the assertion $\{\{h = Head\}\}$ inside the atomic section, and thereafter omit its occurrence before the atomic section.
3. We next infer that $h.next$ is untouched between $n := h.next$ and the atomic statement (since any assignment to a *next*-field only occurs when it is *null*). Since $h.next$ is untouched, we extend the atomic section.

4. By the Absorption rule, the assignment $t := Tail$ can be performed together with the atomic statement. Another application of the Absorption rule moves $h := Head$ inside the atomic statement.

Let us consider the second segment in the *Dequeue* method.

$$\begin{array}{l} h := Head; \\ t := Tail; \\ n := h.next; \\ \{\{h = Head\}\}; \\ \{\{h = t\}\}; \\ \{\{n = null\}\}; \\ \textbf{return } empty \end{array} \sqsupseteq \begin{array}{l} h := Head; \\ t := Tail; \\ [\![\ n := h.next; \\ \quad \{\{h = t\}\}; \\ \quad \{\{n = null\}\}\ ]\!]; \\ \{\{h = Head\}\}; \\ \{\{h = t\}\}; \\ \{\{n = null\}\}; \\ \textbf{return } empty \end{array} \sqsupseteq \begin{array}{l} [\![\ h := Head; \\ \quad t := Tail; \\ \quad n := h.next; \\ \quad \{\{h = t\}\}; \\ \quad \{\{n = null\}\}\ ]\!]; \\ \{\{h = Head\}\}; \\ \{\{h = t\}\}; \\ \{\{n = null\}\}; \\ \textbf{return } empty \end{array} \sqsupseteq \begin{array}{l} [\![\ h := Head; \\ \quad t := Tail; \\ \quad n := h.next; \\ \quad \{\{h = Head\}\}; \\ \quad \{\{h = t\}\}; \\ \quad \{\{n = null\}\}; \\ \quad \textbf{return } empty\ ]\!] \end{array}$$

The linearization point in this part is the command $n := h.next$, so the atomic statement will be built from there. We consider each step in turn.

1. We first move the following assertions with only local variables into the atomic statement.
2. By combining the inferred properties $h \xrightarrow{*} t \xrightarrow{*} Tail$ and $h \xrightarrow{*} Head$ and the invariant $Head \xrightarrow{*} Tail$ with the assertions in the atomic section (implying $h = t \wedge h.next = null$), we infer that $h = t = Head = Tail$ at the linearization point. We can therefore add the assertion $\{\{h = Head \wedge t = Tail\}\}$ at the beginning of the atomic section, and thereafter absorb the two first assignments.
3. The three assumptions that follow the atomic statement are all valid at the end of the atomic statement, so we can move them inside the atomic statement. Finally, the return statement is purely local, and can also be included.

Let us finally consider the last part of the *Dequeue* method.

$$\begin{array}{l} h := Head; \\ t := Tail; \\ n := h.next; \\ \{\{h = Head\}\}; \\ \{\{h \neq t\}\}; \\ retval := n.data; \\ [\![\ \{\{h = Head\}\}; \\ \quad Head := n\ ]\!]; \\ \textbf{return } retval \end{array} \sqsupseteq \begin{array}{l} h := Head; \\ t := Tail; \\ n := h.next; \\ [\![\ \{\{h = Head\}\}; \\ \quad \{\{h \neq t\}\}; \\ \quad retval := n.data; \\ \quad \{\{h = Head\}\}; \\ \quad Head := n; \\ \quad \textbf{return } retval\ ]\!] \end{array} \sqsupseteq \begin{array}{l} h := Head; \\ t := Tail; \\ [\![\ n := h.next; \\ \quad \{\{h = Head\}\}; \\ \quad \{\{h \neq t\}\}; \\ \quad retval := n.data; \\ \quad \{\{h = Head\}\}; \\ \quad Head := n; \\ \quad \textbf{return } retval\ ]\!] \end{array} \sqsupseteq \begin{array}{l} [\![\ h := Head; \\ \quad n := h.next; \\ \quad \{\{h = Head\}\}; \\ \quad \{\{n \neq null\}\}; \\ \quad retval := n.data; \\ \quad \{\{h = Head\}\}; \\ \quad Head := n; \\ \quad \textbf{return } retval\ ]\!] \end{array} \sqsupseteq \begin{array}{l} [\![\ h := Head; \\ \quad t := Tail; \\ \quad n := h.next; \\ \quad \{\{h = Head\}\}; \\ \quad \{\{h \neq t\}\}; \\ \quad retval := n.data; \\ \quad \{\{h = Head\}\}; \\ \quad Head := n; \\ \quad \textbf{return } retval\ ]\!] \end{array}$$

We consider each step in turn.

1. The expansion of the atomic statement to include $\{\{h \neq t\}\}; retval := n.data$ is straight-forward. Thereafter, the assertion $\{\{h = Head\}\}$ holds at the beginning of the atomic statement, so we can add it there, and finally remove the occurrence of $\{\{h = Head\}\}$ that is just before the atomic statement.
2. By combining the inferred properties $h \xrightarrow{*} t \xrightarrow{*} Tail$ and $h \xrightarrow{*} Head$ with the assertion $h \neq t$, we infer that $h.next \neq null$ after the assignment $t := Tail$. This assignment therefore commutes right with all other commands in the program, and can be moved inside the atomic statement.
3. We replace the assertion $\{\{h \neq t\}\}$ by the implied assertion $\{\{n \neq null\}\}$. Thereafter the assignment $t := Tail$ becomes dead, and can be removed. Finally, we can then absorb the first assignment $h := Head$.
4. We now infer that $h \xrightarrow{+} Tail$ inside the atomic section, and can therefore replace the assertion $\{\{n \neq null\}\}$ by the two commands $t := Tail; \{\{h \neq t\}\}$.

**Making Method Bodies Atomic** We have now transformed the program into two methods consisting of totally six atomic statements. By removing assignments to local variables that are dead when exiting an atomic statement, and making trivial reorganizations of commands inside atomic statements, we can express

the resulting program in the following form.

```
void Enqueue(Data d) {
  Node *t, *n, *x;
  do {
    [[ {{Tail.next ≠ null}};
       Tail := Tail.next ]]
  };
  [[ new(x);
    x.data := d;
    t := Tail;
    {{Tail.next = null}};
    Tail.next := x ]];
  [[ if (t = Tail) Tail := Tail.next ]];
  return
}
```

```
Data Dequeue() {
  Node *h, *t, *n;
  do {
    [[ {{Head = Tail}};
       {{Head.next ≠ null}};
       Tail := Head.next ]];
  };
  [[ either {
       {{Head = Tail}};
       {{Head.next = null}};
       return empty
     } or {
       {{Head ≠ Tail}};
       Head := Head.next;
       return Head.next.data ]]
  }
}
```

Let us introduce names for the atomic statements, so that the two methods can be represented as follows.

void *Enqueue*(Data *d*) : **do** *MoveTail* ; *EnqueueCell* ; *TryMoveTail*
Data *Dequeue*() : **do** *MoveTail'* ; *EmptyOrDequeue*

It would now be natural to use a rule for atomicity refinement to prove that each method body refines a corresponding atomic statement. However, the normal Atomicity Refinement rule cannot easily be used for this. For instance, one condition would then be that *MoveTail* commutes right with *EnqueueCell*. However, this is not true, since if the sequence *MoveTail* ; *EnqueueCell* can be performed from some configuration, then the statement *EnqueueCell* is blocked (since first $Tail$ must be moved).

We therefore use the Strengthened Atomicity Refinement rule, where the concerned statements may occur in **do** loops. We will now establish the commutativity properties between atomic statements that are required by the strengthened atomicity refinement rule. By sequential reasoning (using the global invariant $Head \xrightarrow{*} Tail$), we first establish that *MoveTail* commutes right with all other statements. More precisely, we can establish that

- *MoveTail* commutes right and left with itself (of course),
- *MoveTail* by thread $i$ followed by *EnqueueCell* by $i'$ can be simulated by [[ *MoveTail* ; *EnqueueCell* ]] by $i'$ followed by **skip** by $i$,
- *MoveTail* by thread $i$ followed by *EmptyOrDequeue* by $i'$ can be simulated by [[ **do** *MoveTail'* ; *EmptyOrDequeue* ]] by thread $i'$ followed by [[ **do** *MoveTail* ]] by $i$,
- *MoveTail* by thread $i$ followed by *MoveTail'* by $i'$ is not possible. The same holds for *MoveTail* by thread $i$ followed by *TryMoveTail* by $i'$.

Analogous investigations can be performed to establish that *MoveTail'* commutes right with all other statements. It remains to check that *TryMoveTail* commutes left with *EnqueueCell* and with *EmptyOrDequeue*. For the first case, we note that *EnqueueCell* by a thread $i$ cannot occur before *MoveTail'* by another thread. The second case is straight-forward.

In conclusion, by the Strengthened Atomicity Refinement rule, the two method bodies refine the following ones.

void *Enqueue*(Data *d*) : [[ **do** *MoveTail* ; *EnqueueCell* ; *TryMoveTail* ]]
Data *Dequeue*() : [[ **do** *MoveTail'* ; *EmptyOrDequeue* ]]

As a final step in the proof of linearizability, we can finally re-insert the pure loops, so that the method bodies become exactly the atomic versions of those in Figure 4. This step is analogous to that performed for Treiber's stack.

## 7. Conclusions and Future Work

We have shown how extensions of techniques that originate in the refinement calculus [Bac88, Mor90] can be used for structuring proofs of correctness of programs that implement concurrent objects. In particular, we showed how a natural refinement relation can be used to express that a particular program for concurrent objects is linearizable. We thereafter presented rules for establishing our refinement relation by a sequence of applications of local rules. A main contribution has been stronger techniques for establishing refinement of atomic statements. We applied the rules for proving linearizability of two well-known algorithms for concurrent objects by establishing a refinement relation between two programs through a sequence of refinement steps. Such a sequence also serves as an explanation of why a particular algorithm is linearizable. Since the employed rules mostly rely on locality of accesses, and to a large extent avoid complicated reasoning, e.g., about data types, we think that this work would also be suitable as a basis for automation.

**Acknowledgments** This work has received inspiration and criticism in discussion with colleagues, including Parosh Abdulla, Frédéric Haziza, Lukáš Holík, and Ahmed Rezine, and from the comments by anonymous reviewers. Thanks to Lukáš Holík for reading the manuscript.

## References

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac89] R.J.R. Back. A method for refining atomicity in parallel algorithms. In *PARLE '89, Proc. Parallel Architectures and Languages Europe*, volume 366 of *LNCS*, pages 199–216. Springer, 1989.

[BR06] J.D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Proc. VMCAI*, volume 3855 of *LNCS*, pages 207–221. Springer, 2006.

[BS89] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comput. Program.*, 13(1):133–180, 1989.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[DDG+04] S. Doherty, D. Detlefs, L. Groves, C.H. Flood, V. Luchangco, P.A. Martin, M. Moir, N. Shavit, and G.L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA 2004: Proc.* $16^{th}$ *Annual ACM Symposium on Parallel Algorithms, Barcelona, Spain*, pages 216–224. ACM, 2004.

[EQS+10] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In J. Esparza and R. Majumdar, editors, *Proc. TACAS '10,* $16^{th}$ *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 296–311. Springer, 2010.

[FQ04] Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[Gro08] L. Groves. Verifying Michael and Scott's lock-free queue algorithm using trace reduction. In J. Harland and P. Manyem, editors, *Proc. CATS 2008:* $14^{th}$ *Computing: The Australasian Theory Symposium, Wollongong, NSW, Australia*, volume 77 of *CRPIT*, pages 133–142. Australian Computer Society, 2008.

[Gro09] L. Groves. Reasoning about nonblocking concurrency. *J. UCS*, 15(1):72–111, 2009.

[HW90] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Jon94] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. on Programming Languages and Systems*, 16(2):259–303, 1994.

[Lam90] L. Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[Lip75] R.J. Lipton. Reduction, a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, Dec. 1975.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc.* $6^{th}$ *ACM Symp. on Principles of Distributed Computing, Vancouver, Canada*, pages 137–151. ACM, 1987.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[MS95] M.M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Rochester, NY, USA, 1995.

[MS96] M.M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[Tre86] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.

[WS05] L. Wang and S.D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP, Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 61–71. ACM, 2005.