

Modular Full Functional Specification and Verification of Lock-Free Data Structures

Bart Jacobs *Frank Piessens*

Report CW 551, June 2009



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Modular Full Functional Specification and Verification of Lock-Free Data Structures

Bart Jacobs *Frank Piessens*

Report CW551, June 2009

Department of Computer Science, K.U.Leuven

Abstract

We propose an approach for specifying and verifying full functional (partial) correctness of modules of multithreaded imperative programs that implement or use lock-free data structures for inter-thread communication. The approach extends separation logic with simple *atomic spaces*, which are regions of memory that may be accessed concurrently using atomic operations. A fixed invariant is associated with each atomic space. The specification of an atomic operation consists of the usual precondition and postcondition, as well as a number of *proof obligations*. The precondition requires permission to access an atomic space. The proof obligations state that, in the specific context of the call, the atomic space's invariant can be rewritten to separate out the lock-free data structure, and that updating the data structure preserves the invariant. To allow threads to retain information about the state of the data structure, a combination of fractional permissions and ghost cells is used in a way similar to the use of auxiliary variables in Owicki-Gries reasoning.

The approach has been implemented in the VeriFast program verifier, and used to verify an implementation and a client program of a multiple-enqueueer, single-dequeueer lock-free queue.

Modular Full Functional Specification and Verification of Lock-Free Data Structures

Bart Jacobs* Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
 {bart.jacobs,frank.piessens}@cs.kuleuven.be

```
createCell(c)
x := get(c)
set(c, x)
x := compareAndSet(c, x0, x1)
disposeCell(c)
```

Figure 1. Syntactic interface of the cell module. The `createCell(c)` operation turns the memory location at address `c` into a cell. `disposeCell(c)` turns it back into an ordinary memory location.

Abstract

We propose an approach for specifying and verifying full functional (partial) correctness of modules of multithreaded imperative programs that implement or use lock-free data structures for inter-thread communication. The approach extends separation logic with simple *atomic spaces*, which are regions of memory that may be accessed concurrently using atomic operations. A fixed invariant is associated with each atomic space. The specification of an atomic operation consists of the usual precondition and postcondition, as well as a number of *proof obligations*. The precondition requires permission to access an atomic space. The proof obligations state that, in the specific context of the call, the atomic space’s invariant can be rewritten to separate out the lock-free data structure, and that updating the data structure preserves the invariant. To allow threads to retain information about the state of the data structure, a combination of fractional permissions and ghost cells is used in a way similar to the use of auxiliary variables in Owicki-Gries reasoning.

The approach has been implemented in the VeriFast program verifier, and used to verify an implementation and a client program of a multiple-enqueueer, single-dequeueer lock-free queue.

1. Overview

We first introduce the approach using two example lock-free data structures: a simple cell in Section 2, and a queue in Section 3. We sketch a soundness proof in Section 4 and we describe how our program verifier prototype supports the approach in Section 5. The paper ends with a preliminary survey of related work in Section 6, and a conclusion in Section 7.

2. Cell Example

Figures 1, 2, and 3 show the syntactic interface, the implementation, and a client program for a simple atomic cell data structure.

The challenge is to provide a specification for the cell module that performs proper information hiding but at the same time allows the client developer to verify that the assert statement does not fail.

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

```
createCell(c)  $\triangleq$  skip
x := get(c)  $\triangleq$  atomic(x := [c])
set(c, x)  $\triangleq$  atomic([c] := x)
x := compareAndSet(c, x0, x1)  $\triangleq$ 
  atomic(x := [c]; if x = x0 then [c] := x1)
disposeCell(c)  $\triangleq$  skip
```

Figure 2. Implementation of the cell module

```
c := cons(0);
createCell(c);
fork (set(c, 1));
x := get(c);
y := get(c);
assert x ≤ y
```

Figure 3. A client program of the cell module

We propose the specification in Figure 4. The atomic operations `get`, `set`, and `compareAndSet` operate on a fraction π (a real number between zero, exclusive, and one, inclusive) of an *atomic space* with invariant I . The specifications of these operations include a number of *proof obligations* that state that it must be possible to rewrite the invariant, when combined with some context information and resources P , into a separate conjunction of the cell data structure itself, and some remainder S , which may depend on the value of the cell. They further require that starting from the combination of the updated cell and the same remainder, the invariant can again be obtained, along with a new remainder Q that remains local to the current thread. We use the \Leftrightarrow operator to indicate that ghost field mutations and other ghost operations may be used during the rewriting.

An atomic space may be created from any piece of state:

$$S \Rightarrow \boxed{S}$$

(If no fraction is specified, 1 is implied.) It may also be “dismantled”:

$$\boxed{S} \Rightarrow S$$

A fraction of an atomic space may be split into multiple fractions, which may then be distributed among multiple threads:

$$(\pi_1 + \pi_2) \boxed{S} \Rightarrow \pi_1 \boxed{S} * \pi_2 \boxed{S}$$

Conversely, multiple fractions of an atomic space may be combined into one:

$$\pi_1 \boxed{S} * \pi_2 \boxed{S} \Rightarrow (\pi_1 + \pi_2) \boxed{S}$$

$$\begin{array}{c}
\{c \mapsto v\} \text{createCell}(c) \{\text{cell}(c, v)\} \quad \frac{I * P \Leftrightarrow \exists v \bullet \text{cell}(c, v) * S(v) \quad \forall v \bullet \text{cell}(c, v) * S(v) \Rightarrow I * Q(v)}{\{\pi \overline{I}\} * P\} x := \text{get}(c) \{\pi \overline{I}\} * Q(x)} \\
\\
\frac{I * P \Leftrightarrow \text{cell}(c, _)* S \quad \text{cell}(c, v') * S \Rightarrow I * Q}{\{\pi \overline{I}\} * P\} \text{set}(c, v') \{\pi \overline{I}\} * Q\} \\
\\
\frac{I * P \Leftrightarrow \exists v \bullet \text{cell}(c, v) * S(v) \quad \text{cell}(c, v_1) * S(v_0) \Rightarrow I * Q(v_0) \quad \forall v \bullet \text{cell}(c, v) * S(v) \wedge v \neq v_0 \Rightarrow I * Q(v)}{\{\pi \overline{I}\} * P\} x := \text{compareAndSet}(c, v_0, v_1) \{\pi \overline{I}\} * Q(x)} \\
\\
\overline{\{\text{cell}(c, v)\} \text{disposeCell}(c) \{c \mapsto v\}}
\end{array}$$

Figure 4. Specification of the cell module

Atomic spaces may be accessed using primitive atomic operations:

$$\frac{\{S * P\} c \{S * Q\}}{\{\pi \overline{S}\} * P\} \text{atomic}(c) \{\pi \overline{S}\} * Q\}$$

Verification of the cell module's implementation is trivial, using the following definition of the cell predicate:

$$\text{cell}(c, v) \triangleq c \mapsto v$$

To verify the client program, we need Owicki-Gries reasoning. To enable this, we use fractional permissions on points-to assertions (Bornat et al. 2005) as well:

$$\ell \mapsto v \Leftrightarrow \ell \stackrel{1/2}{\mapsto} v * \ell \stackrel{1/2}{\mapsto} v$$

Owicki-Gries reasoning is enabled by the following property:

$$\ell \stackrel{1/2}{\mapsto} v_1 * \ell \stackrel{1/2}{\mapsto} v_2 \Rightarrow v_1 = v_2$$

We introduce ghost memory cell allocations and accesses, that serve the purpose of Owicki-Gries auxiliary variables. It is easy to show using separation logic that these auxiliary operations do not modify the program's behavior.

The proof outline is shown in Figure 5. It uses the following proof rule for the **fork** command:

$$\frac{\text{FORK} \quad \{P\} c \{\text{true}\}}{\{P * R\} \text{fork}(c) \{R\}}$$

For simplicity, we do not worry about resource cleanup at the end of a thread in the examples.

3. Queue example

In this section we modularly specify and verify full functional partial correctness of a lock-free queue that supports concurrent execution of a single dequeue operation and any number of enqueue operations. Figures 6 and 7 show the syntactic interface and an example client program.

Figure 8 shows how this module can be specified using our approach. We use α, β, \dots to denote lists of integers, ϵ to denote the empty list, and \cdot to denote concatenation of lists and elements.

A proof of the example client program is shown in Figure 9. To express that the queue holds addresses of cells whose values are strictly increasing, the proof uses the auxiliary definitions from Figure 10. The proof uses two auxiliary variables l and u , which hold a lower bound and an upper bound on the values of the cells in the queue, respectively. The upper bound is controlled by the enqueueing thread, the lower bound by the dequeueing thread.

```

{emp}
c := cons(0); createCell(c);
{cell(c, 0)}
a := gcons(0); (ghost cell allocation)
{cell(c, 0) * a ↦ 0}
{∃v, b • cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1 * a 1/2 0}
fork (
  {1/2∃v, b • cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1}
  set(c, 1)
  {1/2∃v, b • cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1}
);
{1/2∃v, b • cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1 * a 1/2 0}
cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1 * a 1/2 0 ∧ x = v
⇔ merging of fractions
cell(c, v) * a ↦ 0 * 0 ≤ v * v ≤ 1 ∧ x = v
⇔ ghost cell update
cell(c, v) * a ↦ v * 0 ≤ v * v ≤ 1 ∧ x = v
⇔ splitting of fractions
cell(c, v) * a 1/2 v * v ≤ v * v ≤ 1 * a 1/2 v ∧ x = v
x := get(c);
{1/2∃v, b • cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1 * a 1/2 x}
y := get(c);
{1/2∃v, b • cell(c, v) * a 1/2 b * b ≤ v * v ≤ 1 * a 1/2 x ∧ x ≤ y}
assert x ≤ y

```

Figure 5. Proof of the cell module client program. The proof of the second proof obligation of the first get call is shown; the other proof obligations can be discharged analogously or trivially.

Notice again how the approach allows client code to both extend and constrain the queue data structure in a modular way.

The implementation of the queue module is shown in Figure 11. We use field notation for readability. The internal state of the queue consists of two singly linked lists that both end at the same node, pointed to by $q.middle$. The first list starts at $q.first$ and the second one at $q.last$. The node at $q.first$ is a dummy node; the other nodes, starting at $q.first.next$, through $q.middle$ and up to and including $q.last$ contain the elements of the queue. Procedure enqueue allocates a new node and prepends it at $q.last$. Procedure

$$\begin{array}{c}
\frac{}{\{\mathbf{emp}\} q := \mathbf{create}() \{ \mathbf{queue}(q, \epsilon) * \mathbf{consumer}(q) \}} \quad \frac{I * P \Leftrightarrow \exists \alpha \bullet \mathbf{queue}(q, \alpha) * S(\alpha) \quad \forall \alpha \bullet \mathbf{queue}(q, \alpha \cdot v) * S(\alpha) \Rightarrow I * Q}{\{ \pi \overline{I} * P \} \mathbf{enqueue}(q, v) \{ \pi \overline{I} * Q \}} \\
\frac{I * P \Leftrightarrow \exists \alpha \bullet \mathbf{queue}(q, \alpha) * S(\alpha) \quad \forall v, \alpha \bullet \mathbf{queue}(q, \alpha) * S(v \cdot \alpha) \Rightarrow I * Q(v) \quad \mathbf{queue}(q, \epsilon) * S(\epsilon) \Rightarrow I * Q(0)}{\{ \mathbf{consumer}(q) * \pi \overline{I} * P \} x := \mathbf{tryDequeue}(q) \{ \mathbf{consumer}(q) * \pi \overline{I} * Q(x) \}} \\
\frac{}{\{ \exists \alpha \bullet \mathbf{queue}(q, \alpha) * \mathbf{consumer}(q) \} \mathbf{disposeQueue}(q) \{ \mathbf{emp} \}}
\end{array}$$

Figure 8. Specification of the queue module

```

{emp}
q := create();
{queue(q, ε) * consumer(q)}
l := gcons(0); (ghost cell allocation)
u := gcons(0); (ghost cell allocation)
{I * l 1/2 0 * u 1/2 0 * consumer(q)}
fork (
  { $\frac{1}{2} \overline{I} * l \xrightarrow{1/2} 0 * \mathbf{consumer}(q)$ }
  Second proof obligation:
  queue(q, α) * ∃β, k, t • cells(a · α, β) * l 1/2 k * u 1/2 t ∧ ok(k, β, t) ∧ k ≤ t + 1 * l 1/2 0
  ⇒ Merging l; destructing β
  queue(q, α) * ∃X, β, t • cells(a · α, X · β) * l ↦ 0 * u 1/2 t ∧ ok(0, X · β, t) ∧ 0 ≤ t + 1
  ⇒ Mutation of l; splitting of l; unfolding of cells, ok
  queue(q, α) * ∃β, X, t • cells(α, β) * l 1/2 X + 1 * u 1/2 t ∧ ok(X + 1, β, t) ∧ X + 1 ≤ t + 1 * a ↦ X * l 1/2 X + 1
  a := tryDequeue(q);
  { $\frac{1}{2} \overline{I} * \mathbf{consumer}(q) * (a \neq 0 \Rightarrow \exists X \bullet a \mapsto X * l \xrightarrow{1/2} X + 1)$ }
  b := tryDequeue(q);
  { $\frac{1}{2} \overline{I} * \mathbf{consumer}(q) * (a \neq 0 \wedge b \neq 0 \Rightarrow \exists X, Y \bullet a \mapsto X * b \mapsto Y * l \xrightarrow{1/2} X + 1 \wedge X < Y)$ }
  if a ≠ 0 ∧ b ≠ 0 then (
    x := [a];
    y := [b];
    assert x < y
  )
);
{ $\frac{1}{2} \overline{I} * u \xrightarrow{1/2} 0$ }
c := cons(1);
enqueue(q, c);
{ $\frac{1}{2} \overline{I} * u \xrightarrow{1/2} 1$ }
d := cons(2);
enqueue(q, d)
where I = ∃α, β, k, t • queue(q, α) * cells(α, β) * l 1/2 k * u 1/2 t ∧ ok(k, β, t) ∧ k ≤ t + 1

```

Figure 9. Proof of the example client program; a representative proof obligation proof is shown

```

q := create()
enqueue(q, v)
v := tryDequeue(q) returns zero if queue is empty
disposeQueue(q)

```

Figure 6. Syntactic interface of the queue module

tryDequeue first checks if there are any elements in the front part of the queue. If $q.\mathbf{first}$ equals $q.\mathbf{middle}$, this is not the case, and the $q.\mathbf{last}$ pointer is read. All nodes between this value of $q.\mathbf{last}$ and $q.\mathbf{middle}$ are then transferred into the front part by flipping the

direction of the next pointers and updating the $q.\mathbf{middle}$ pointer. If no new nodes were added, 0 is returned. Otherwise, $q.\mathbf{first}$ is now distinct from $q.\mathbf{middle}$, and the value of the first non-dummy node is returned, the dummy node is disposed, and the first non-dummy node becomes the new dummy node.

The pointer to the last node is accessed atomically. To illustrate that our approach supports modular verification of lock-free data structures built on top of other lock-free data structures, we use the cell module to store this pointer.

This implementation supports a single dequeue operation happening concurrently with any number of enqueue operations. This is encoded in the specification by the fact that a dequeue operation

$$\begin{aligned}
\text{queue}(q, \gamma) &\triangleq \exists m, l, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{cell}(\&q.\text{last}, l) * \text{lseg}(l, m, \beta) * m.\text{next} \xrightarrow{1/2} _ * q.\text{front} \xrightarrow{1/2} \alpha \wedge \gamma = \alpha \cdot \beta^\dagger \\
\text{consumer}(q) &\triangleq \exists f, m, \alpha \bullet q.\text{first} \mapsto f * q.\text{middle} \mapsto m * q.\text{middle}' \xrightarrow{1/2} m * \text{lseg}'(f, m, _, \alpha) * q.\text{front} \xrightarrow{1/2} \alpha \\
\text{lseg}(f, l, \alpha) &\triangleq (f = l \wedge \alpha = \epsilon) \vee (f \neq l \wedge \exists v, n, \alpha' \bullet f.\text{value} \mapsto v * f.\text{next} \mapsto n * \text{lseg}(n, l, \alpha') \wedge \alpha = v \cdot \alpha') \\
\text{lseg}'(f, l, v, \alpha) &\triangleq \exists n, v', \alpha' \bullet f.\text{value} \mapsto v * f.\text{next} \xrightarrow{1/2} n * ((f = l \wedge \alpha = \epsilon) \vee (f \neq l \wedge f.\text{next} \xrightarrow{1/2} n * \text{lseg}'(n, l, v', \alpha') \wedge \alpha = v' \cdot \alpha'))
\end{aligned}$$

Figure 12. Definitions for the queue implementation proof

```

q := create();
fork (
  a := tryDequeue(q);
  b := tryDequeue(q);
  if a ≠ 0 ∧ b ≠ 0 then (
    x := [a];
    y := [b];
    assert x < y
  )
);
c := cons(1);
enqueue(q, c);
d := cons(2);
enqueue(q, d)

```

Figure 7. Example client program for the queue module

$$\begin{array}{c}
\frac{}{\text{cells}(\epsilon, \epsilon)} \quad \frac{l \neq 0 \wedge l \mapsto v * \text{cells}(\alpha, \beta)}{\text{cells}(l \cdot \alpha, v \cdot \beta)} \quad \frac{}{\text{ok}(k, \epsilon, t)} \\
\frac{k \leq v \quad v \leq t \quad \text{ok}(k + 1, \beta, t)}{\text{ok}(k, v \cdot \beta, t)}
\end{array}$$

Figure 10. Auxiliary definitions for the proof of the queue client program

requires a non-shared consumer permission, of which there is only one, whereas only the shared queue permission is required for enqueue operations. The definitions of these predicates are given in Figure 12, along with definitions of the auxiliary predicates lseg and lseg' . These predicates refer to the *ghost fields* $q.\text{middle}'$ and $q.\text{front}$. It follows from these definitions that not all of the state of the queue is in the queue predicate. That is, only the nodes from $q.\text{last}$ to $q.\text{middle}$ are in the atomic space; the nodes from $q.\text{first}$ to $q.\text{middle}$ are owned exclusively by the consumer. This means that the consumer does not need to perform an expensive atomic operation while elements are left in the front part of the queue. When the front part is empty, a single atomic read is performed of $q.\text{last}$ and all nodes reachable from that pointer are transferred out of the atomic space and into the consumer's ownership.

The (unverified) progress and complexity properties of the implementation are as follows. `enqueue` is lock-free (but not wait-free) and, in the absence of contention, runs in constant time. `tryDequeue` is wait-free and runs in amortized constant time.

The ghost field $q.\text{middle}'$ is used to enable the consumer to know that the linked list in the atomic space starting at $q.\text{last}$ runs to $q.\text{middle}$. Notice that this approach is superior to sharing $q.\text{middle}$ itself between consumer and queue because then a separate atomic operation would have to be used to update the $q.\text{middle}$ field, which is a real field, whereas the ghost field $q.\text{middle}'$ can be updated during the atomic operation on $q.\text{last}$.

The second argument of the queue predicate reflects the entire contents of the queue, even though only the back part is stored in the atomic space; this is enabled by the use of the $q.\text{front}$ ghost field, which reflects the values of the front part.

The queue predicate contains half of the permission on the next field of the middle node; this allows an enqueue operation to prove that the middle node is an allocated node and therefore the newly allocated node is distinct from the middle node. This, together with the fact that the $q.\text{first}$ node is a dummy node, is the reason why lseg' is defined the way it is.

The proof of the queue module is shown in Figures 13, 14, and 15. The proof of dequeue uses the following rule to update $q.\text{front}$:

$$\begin{array}{c}
\text{ATOMIC-NOOP} \\
I * P \Rightarrow I * Q \\
\hline
\{\pi \boxed{I} * P\} \text{ ATOMIC-NOOP } \{\pi \boxed{I} * Q\}
\end{array}$$

This rule allows the proof to insert an operation on an atomic space provided that it modifies only the ghost state. Note: This ghost operation may not be inserted in the body of an **atomic** command.

4. Soundness

In this section, we state and prove the soundness of the program logic used in our approach, which is an extension of separation logic with ghost fields and atomic spaces.

4.1 Programming language

For simplicity, we consider a programming language without mutable variables; mutable variables can be simulated using heap cells. Commands have a result. We ignore subroutines in the proof.

$$c ::= \text{cons}(\bar{v}) \mid [\ell] \mid [\ell] := v \mid \text{let } x := c \text{ in } c(x) \mid \text{return } v \mid \text{atomic}(c) \mid \text{assert}(b) \mid \text{fork}(c)$$

We sometimes write `let $x := c$ in $c'(x)$` as `$x := c; c'$` . If c' does not depend on x , we also write `$c; c'$` .

4.2 Configurations

A full heap is a finite map from addresses (positive integers) to integers:

$$\text{FullHeaps} = \mathbb{Z}_0^+ \rightarrow_{\text{fin}} \mathbb{Z}$$

A continuation is either the **done** continuation or a **let** continuation:

$$\kappa ::= \text{done} \mid \text{let } x := c \text{ in } \kappa(x)$$

A thread id is a positive integer:

$$\text{ThreadIds} = \mathbb{Z}_0^+$$

A configuration consists of a full heap, and a finite map from thread ids to continuations:

$$\text{Configs} = \text{FullHeaps} \times (\text{ThreadIds} \rightarrow \kappa)$$

4.3 Executions

An execution of a program c is a finite or infinite sequence of configurations, starting in the initial configuration $\langle \emptyset, \{(1, c; \text{done})\} \rangle$,

```

enqueue( $q, v$ )  $\triangleq$ 
{ $A * P$ }
 $n := \text{cons}(\text{value} := v, \text{next} := 0);$ 
{ $A * P * n.\text{value} \mapsto v * n.\text{next} \mapsto 0$ }
let iter()  $\triangleq$ 
{ $A * P * n.\text{value} \mapsto v * n.\text{next} \mapsto \_$ }
Framing  $n.\text{value} \mapsto v * n.\text{next} \mapsto \_$ 
{ $A * P$ }
  First proof obligation:
   $I * P$ 
   $\Leftrightarrow$  (1)
   $\exists \gamma \bullet \text{queue}(q, \gamma) * S(\gamma)$ 
   $\Leftrightarrow$ 
   $\exists l \bullet \text{cell}(\&q.\text{last}, l) * \exists \gamma, m, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{lseg}(l, m, \beta) * m.\text{next} \xrightarrow{1/2} \_ * q.\text{front} \xrightarrow{1/2} \alpha * \gamma = \alpha \cdot \beta^\dagger * S(\gamma)$ 
  Second proof obligation:
   $\text{cell}(\&q.\text{last}, l) * \exists \gamma, m, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{lseg}(l, m, \beta) * m.\text{next} \xrightarrow{1/2} \_ * q.\text{front} \xrightarrow{1/2} \alpha * \gamma = \alpha \cdot \beta^\dagger * S(\gamma)$ 
   $\Rightarrow$ 
   $\exists \gamma \bullet \text{queue}(q, \gamma) * S(\gamma)$ 
   $\Leftrightarrow$  (1)
   $I * P$ 
   $l := \text{get}(\&q.\text{last});$ 
  { $A * P$ }
{ $A * P * n.\text{value} \mapsto v * n.\text{next} \mapsto \_$ }
 $n.\text{next} := l;$ 
{ $A * P * n.\text{value} \mapsto v * n.\text{next} \mapsto l$ }
  First proof obligation:
   $I * P * n.\text{value} \mapsto v * n.\text{next} \mapsto l$ 
   $\Leftrightarrow$  (1)
   $\exists \gamma \bullet \text{queue}(\gamma) * S(\gamma) * n.\text{value} \mapsto v * n.\text{next} \mapsto l$ 
   $\Leftrightarrow$ 
   $\exists l' \bullet \text{cell}(\&q.\text{last}, l') * \exists \gamma, m, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{lseg}(l', m, \beta) * m.\text{next} \xrightarrow{1/2} \_ * q.\text{front} \xrightarrow{1/2} \alpha * \gamma = \alpha \cdot \beta^\dagger * S(\gamma) * n.\text{value} \mapsto v * n.\text{next} \mapsto l$ 
  Second proof obligation:
   $\text{cell}(\&q.\text{last}, n) * \exists \gamma, m, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{lseg}(l, m, \beta) * m.\text{next} \xrightarrow{1/2} \_ * q.\text{front} \xrightarrow{1/2} \alpha * \gamma = \alpha \cdot \beta^\dagger * S(\gamma) * n.\text{value} \mapsto v * n.\text{next} \mapsto l \wedge l' = l$ 
   $\Leftrightarrow$  rearranging
   $\exists m, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * (n.\text{value} \mapsto v * n.\text{next} \mapsto l * \text{lseg}(l, m, \beta)) * m.\text{next} \xrightarrow{1/2} \_ * q.\text{front} \xrightarrow{1/2} \alpha * S(\alpha \cdot \beta^\dagger) \wedge l' = l$ 
   $\Rightarrow$  folding lseg, applying (2)
   $I * Q \wedge l' = l$ 
  Third proof obligation:
   $\text{cell}(\&q.\text{last}, l') * \exists \gamma, m, \alpha, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{lseg}(l', m, \beta) * m.\text{next} \xrightarrow{1/2} \_ * q.\text{front} \xrightarrow{1/2} \alpha * \gamma = \alpha \cdot \beta^\dagger * S(\gamma) * n.\text{value} \mapsto v * n.\text{next} \mapsto l \wedge l' \neq l$ 
   $\Leftrightarrow$  (1)
   $I * P * n.\text{value} \mapsto v * n.\text{next} \mapsto l \wedge l' \neq l$ 
 $l' := \text{compareAndSet}(\&q.\text{last}, l, n);$ 
{ $A * ((l' = l \wedge Q) \vee (l' \neq l \wedge P * n.\text{value} \mapsto v * n.\text{next} \mapsto l))$ }
if  $l' \neq l$  then iter()
{ $A * Q$ }
in iter()
{ $A * Q$ }
where  $A = \pi \boxed{I}$ 
and  $I * P \Leftrightarrow \exists \gamma \bullet \text{queue}(q, \gamma) * S(\gamma)$  (1)
and  $\forall \gamma \bullet \text{queue}(q, \gamma \cdot v) * S(\gamma) \Rightarrow I * Q$  (2)

```

Figure 13. Proof of procedure enqueue

$$\begin{aligned}
& \text{dequeue}(q) \triangleq \\
& \left\{ \begin{array}{l} \exists f, n, m, v, \alpha \bullet q.\text{first} \mapsto f * q.\text{middle} \mapsto m * q.\text{middle}' \xrightarrow{1/2} m' * \\ f.\text{value} \mapsto _ * f.\text{next} \mapsto n * \text{lseg}'(n, m, v, \alpha) * q.\text{front} \xrightarrow{1/2} v \cdot \alpha * A * P \end{array} \right\} \\
& f := q.\text{first}; \\
& n := f.\text{next}; \\
& x := n.\text{value}; \\
& q.\text{first} := n; \\
& \text{dispose}(f.\text{first}, f.\text{next}) \\
& \left\{ \begin{array}{l} \exists m, \alpha \bullet q.\text{first} \mapsto n * q.\text{middle} \mapsto m * q.\text{middle}' \xrightarrow{1/2} m' * \\ \text{lseg}'(n, m, x, \alpha) * q.\text{front} \xrightarrow{1/2} x \cdot \alpha * A * P \end{array} \right\} \\
& I * P * q.\text{front} \xrightarrow{1/2} x \cdot \alpha \\
& \Leftrightarrow (1) \\
& \exists \gamma \bullet \text{queue}(\gamma) * S(\gamma) * q.\text{front} \xrightarrow{1/2} x \cdot \alpha \\
& \Leftrightarrow \\
& \exists m, l, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{cell}(\&q.\text{last}, l) * \text{lseg}(l, m, \beta) * m.\text{next} \xrightarrow{1/2} _ * q.\text{front} \mapsto x \cdot \alpha * S(x \cdot \alpha \cdot \beta^\dagger) \\
& \Leftrightarrow \text{Mutation of } q.\text{front} \\
& \exists m, l, \beta \bullet q.\text{middle}' \xrightarrow{1/2} m * \text{cell}(\&q.\text{last}, l) * \text{lseg}(l, m, \beta) * m.\text{next} \xrightarrow{1/2} _ * q.\text{front} \mapsto \alpha * S(x \cdot \alpha \cdot \beta^\dagger) \\
& \Rightarrow (2) \\
& I * Q(x) * q.\text{front} \xrightarrow{1/2} \alpha \\
& \text{ATOMIC-NOOP} \\
& \left\{ \begin{array}{l} \exists m, \alpha \bullet q.\text{first} \mapsto n * q.\text{middle} \mapsto m * q.\text{middle}' \xrightarrow{1/2} m' * \\ \text{lseg}'(n, m, x, \alpha) * q.\text{front} \xrightarrow{1/2} \alpha * A * Q(x) \end{array} \right\} \\
& \{\text{consumer}(q) * A * Q(x)\} \\
& \text{where } A = \pi \boxed{I} \\
& \text{and } I * P \Leftrightarrow \exists \gamma \bullet \text{queue}(q, \gamma) * S(\gamma) \quad (1) \\
& \text{and } \forall v, \gamma \bullet \text{queue}(q, \gamma) * S(v \cdot \gamma) \Rightarrow I * Q(v) \quad (2) \\
& \text{and } \text{queue}(q, \epsilon) * S(\epsilon) \Rightarrow I * Q(0) \quad (3)
\end{aligned}$$

Figure 14. Proof of auxiliary procedure dequeue

and where each configuration is related to the next by the step relation $\xrightarrow{\alpha}$ defined in Figure 16. The label of the steps is of no consequence for the definition of executions; it is used to define data-race-freedom.

Notice that we do not provide a step rule for general **atomic** commands. This simplifies the definitions and the proof. Also, our proof rules are not sound in the presence of nested atomic commands.

A configuration C is racy if

$$C \xrightarrow{t_1: \ell, k_1} \xrightarrow{t_2: \ell, k_2}$$

where $t_1 \neq t_2$ and $k_1 \neq \mathbf{a} \vee k_2 \neq \mathbf{a}$. A program has a data race if a racy configuration is reachable from its initial configuration. In other words, a program has a data race if it has an execution with two adjacent accesses of the same location ℓ by different threads, and they are not both atomic accesses.

The semantics given here assumes a sequentially consistent memory model. However, this semantics is also sound for a weaker memory model provided that the memory model satisfies the DRF0 property (Adve and Hill 1990) and the program is data-race-free. The DRF0 property states that if all sequentially consistent executions of a program are data-race-free, then all executions of the program are sequentially consistent. The proposed C memory model (Boehm and Adve 2008) satisfies this property, and our definition of data-race-freedom matches the one from (Boehm and Adve 2008) provided the sequentially consistent versions of the atomic operations are used to implement our **atomic** commands.

4.4 Assertions

An assertion is a predicate over a real fractional heap, a ghost fractional heap, and a set of atomic space permissions.

A real fractional heap is a finite map from addresses to pairs of fractions and values. A fraction is a real number between zero, exclusive and one, inclusive.

$$\text{FracHeaps} = \mathbb{Z}_0^+ \rightarrow_{\text{fin}} ((0, 1] \times \mathbb{Z})$$

The sum of two fractional heaps $h_1 + h_2$ is the following set:

$$\begin{aligned}
h_1 + h_2 = & \\
& \{(\ell, (\pi_1, v)) \in h_1 \mid \neg \exists \pi_2 \bullet (\ell, (\pi_2, v)) \in h_2\} \cup \\
& \{(\ell, (\pi_2, v)) \in h_2 \mid \neg \exists \pi_1 \bullet (\ell, (\pi_1, v)) \in h_1\} \cup \\
& \{(\ell, (\pi_1 + \pi_2, v)) \mid (\ell, (\pi_1, v)) \in h_1 \wedge (\ell, (\pi_2, v)) \in h_2\}
\end{aligned}$$

Notice that the sum of two fractional heaps is not a function if the heaps do not agree on the value of a location.

A fractional ghost heap is a finite map from ghost addresses to pairs of fractions and values. A ghost address is a pair of a real address (or zero) and a ghost offset. Ghost addresses for real address zero are used for ghost cells that are not associated with a real memory object, and are allocated separately; ghost addresses for a nonzero real address are ghost fields of real objects.

$$\text{FracGhostHeaps} = (\mathbb{Z}^+ \times \mathbb{Z}^+) \rightarrow_{\text{fin}} ((0, 1] \times \mathbb{Z})$$

The sum of two fractional ghost heaps is defined analogously to the sum of fractional heaps.

A set of atomic space permissions is a total map from atomic invariants to nonnegative reals. An atomic invariant is a predicate

```

x := tryDequeue(q)  $\triangleq$ 
{consumer(q) * A * P}
if q.first  $\neq$  q.middle then
  dequeue()
else (
  {A * ( $\exists m \bullet q.first \mapsto m * q.middle \mapsto m * q.middle' \xrightarrow{1/2} m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_ * q.front \xrightarrow{1/2} \epsilon$ ) * P}
  First proof obligation:
  I * ( $\exists m \bullet q.first \mapsto m * q.middle \mapsto m * q.middle' \xrightarrow{1/2} m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_ * q.front \xrightarrow{1/2} \epsilon$ ) * P
   $\Leftrightarrow$  (1)
   $\exists l \bullet \text{cell}(\&q.last, l) * (\exists m, \alpha, \beta \bullet q.middle' \xrightarrow{1/2} m * \text{lseg}(l, m, \beta) * m.next \xrightarrow{1/2} \_ * q.front \xrightarrow{1/2} \alpha * S(\alpha \cdot \beta^\dagger)) *$ 
  ( $\exists m \bullet q.first \mapsto m * q.middle \mapsto m * q.middle' \xrightarrow{1/2} m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_ * q.front \xrightarrow{1/2} \epsilon$ )
  Second proof obligation:
   $\text{cell}(\&q.last, l) * (\exists m, \alpha, \beta \bullet q.middle' \xrightarrow{1/2} m * \text{lseg}(l, m, \beta) * m.next \xrightarrow{1/2} \_ * q.front \xrightarrow{1/2} \alpha * S(\alpha \cdot \beta^\dagger)) *$ 
  ( $\exists m \bullet q.first \mapsto m * q.middle \mapsto m * q.middle' \xrightarrow{1/2} m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_ * q.front \xrightarrow{1/2} \epsilon$ )
   $\Leftrightarrow$  Merging q.middle', q.front
   $\exists m, \beta \bullet \text{cell}(\&q.last, l) * q.middle' \mapsto m * \text{lseg}(l, m, \beta) * m.next \xrightarrow{1/2} \_ * q.front \mapsto \epsilon * S(\beta^\dagger) *$ 
   $q.first \mapsto m * q.middle \mapsto m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_$ 
   $\Leftrightarrow$  Assignment to q.middle', q.front; rearrangement
   $\exists m, \beta \bullet \text{cell}(\&q.last, l) * q.middle' \mapsto l * \text{lseg}(l, l, \epsilon) * l.next \xrightarrow{1/2} \_ * q.front \mapsto \beta^\dagger * S(\beta^\dagger) * q.first \mapsto m * q.middle \mapsto m *$ 
   $m.value \mapsto \_ * m.next \xrightarrow{1/2} \_$ 
  ( $l \neq m \Rightarrow \exists v, \beta' \bullet \beta = v \cdot \beta' \wedge m.next \xrightarrow{1/2} \_ * l.next \xrightarrow{1/2} n * l.value \mapsto v * \text{lseg}(n, m, \beta')$ ) *
  ( $l = m \Rightarrow \beta = \epsilon \wedge \text{emp}$ )
   $\Rightarrow$  splitting of q.middle', q.front; if  $l \neq m$ , application of (1 $\Leftrightarrow$ ); if  $l = m$ , application of (3)
  I * q.first  $\mapsto m * q.middle \mapsto m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_ * q.middle' \xrightarrow{1/2} l * q.front \xrightarrow{1/2} \beta^\dagger *$ 
  ( $l \neq m \Rightarrow \exists v, \beta' \bullet \beta = v \cdot \beta' \wedge m.next \xrightarrow{1/2} \_ * l.next \xrightarrow{1/2} n * l.value \mapsto v * \text{lseg}(n, m, \beta') * P$ ) *
  ( $l = m \Rightarrow \beta = \epsilon \wedge Q(0)$ )
  l := get(&q.last);
  {
  A * q.first  $\mapsto m * q.middle \mapsto m * m.value \mapsto \_ * m.next \xrightarrow{1/2} \_ * q.middle' \xrightarrow{1/2} l * q.front \xrightarrow{1/2} \beta^\dagger *$ 
  ( $l \neq m \Rightarrow \exists v, \beta' \bullet \beta = v \cdot \beta' \wedge m.next \xrightarrow{1/2} \_ * l.next \xrightarrow{1/2} n * l.value \mapsto v * \text{lseg}(n, m, \beta') * P$ ) *
  ( $l = m \Rightarrow \beta = \epsilon \wedge Q(0)$ )
  }
  if l = q.middle then (
    x := 0
  ) else (
    let reverse(n, p)  $\triangleq$   $\forall \alpha, v, \beta, m \bullet$ 
    {q.middle  $\mapsto m * m.next \mapsto \_ * \text{lseg}(p, m, \beta) * \text{lseg}'(n, l, v, \alpha)$ }
    if p = q.middle then
      p.next := n
    else (
      p' := p.next;
      p.next := n;
      reverse(p, p')
    )
    { $\exists n, v', \gamma \bullet v' \cdot \gamma = \beta^\dagger \cdot v \cdot \alpha \wedge q.middle \mapsto m * m.next \mapsto n * \text{lseg}'(n, l, v', \gamma)$ }
    in reverse(l, l.next);
    q.middle := l;
    dequeue()
  )
  )
  {consumer(q) * A * Q(x)}
  where A =  $\pi \boxed{I}$ 
  and I * P  $\Leftrightarrow$   $\exists \gamma \bullet \text{queue}(q, \gamma) * S(\gamma)$  (1)
  and  $\forall v, \gamma \bullet \text{queue}(q, \gamma) * S(v \cdot \gamma) \Rightarrow I * Q(v)$  (2)
  and  $\text{queue}(q, \epsilon) * S(\epsilon) \Rightarrow I * Q(0)$  (3)

```

Figure 15. Proof of procedure tryDequeue

```

q := createQueue()  $\triangleq$ 
  n := cons(value := 0, next := 0);
  q := cons(first := n, middle := n, last := n);
  createCell(&q.last)

enqueue(q, v)  $\triangleq$ 
  n := cons(value := v, next := 0);
  let iter()  $\triangleq$ 
    l := get(&q.last);
    n.next := l;
    l' := compareAndSet(&q.last, l, n);
    if l'  $\neq$  l then iter()
  in iter()

x := tryDequeue(q)  $\triangleq$ 
  let dequeue()  $\triangleq$ 
    f := q.first;
    n := f.next;
    x := n.value;
    q.first := n;
    dispose(f.value, f.next)
  in
  if q.first  $\neq$  q.middle then
    dequeue()
  else (
    l := get(&q.last);
    if l = q.middle then x := 0 else (
      let reverse(n, p)  $\triangleq$ 
        if p = q.middle then
          p.next := n
        else (
          p' := p.next;
          p.next := n;
          reverse(p, p')
        )
      in reverse(l, l.next);
      q.middle := l;
      dequeue()
    )
  )

disposeQueue(q)  $\triangleq$ 
  let disposeList(n, l)  $\triangleq$ 
    if n  $\neq$  l then (
      n' := n.next;
      dispose(n.value, n.next);
      disposeList(n', l)
    )
  in
  disposeCell(&q.last);
  disposeList(q.first, q.middle);
  disposeList(q.last, q.middle);
  dispose(q.middle.value, q.middle.next);
  dispose(q.first, q.middle, q.last)

```

Figure 11. Queue module implementation

| | |
|--|--|
| <p>ALLOC-NIL allocate h, ℓ, ϵ, h</p> | $\frac{\text{ALLOC-CONS} \quad \ell \notin \text{dom}(h) \quad \text{allocate } h, \ell + 1, \bar{v}, h'}{\text{allocate } h, \ell, v \cdot \bar{v}, h'[\ell := v]}$ |
| <p>S-CONS $(t, \text{let } x := \text{cons}(\bar{v}) \text{ in } \kappa(x)) \in T$</p> | $\frac{\text{allocate } h, \ell, \bar{v}, h'}{\langle h, T \rangle \rightsquigarrow \langle h', T[t := \kappa(\ell)] \rangle}$ |
| <p>S-LOOKUP $(t, \text{let } x := [\ell] \text{ in } \kappa(x)) \in T$</p> | $\frac{(\ell, v) \in h}{\langle h, T \rangle \xrightarrow{t, \ell, r} \langle h, T[t := \kappa(v)] \rangle}$ |
| <p>S-MUTATE $(t, [\ell] := v; \kappa) \in T$</p> | $\frac{(\ell, v_0) \in h}{\langle h, T \rangle \xrightarrow{t, \ell, r} \langle h[\ell := v], T[t := \kappa] \rangle}$ |
| <p>S-LET $(t, \text{let } x := \text{let } x := c \text{ in } c'(x) \text{ in } \kappa(x)) \in T$</p> | $\frac{}{\langle h, T \rangle \rightsquigarrow \langle h, T[t := \text{let } x := c \text{ in } \text{let } x := c'(x) \text{ in } \kappa(x)] \rangle}$ |
| <p>S-RETURN $(t, \text{let } x := \text{return } v \text{ in } \kappa(x)) \in T$</p> | $\frac{}{\langle h, T \rangle \rightsquigarrow \langle h, T[t := \kappa(v)] \rangle}$ |
| <p>S-ATOMIC-LOOKUP $(t, \text{let } x := \text{atomic } [\ell] \text{ in } \kappa(x)) \in T$</p> | $\frac{(\ell, v) \in h}{\langle h, T \rangle \xrightarrow{t, \ell, a} \langle h, T[t := \kappa(v)] \rangle}$ |
| <p>S-ATOMIC-MUTATE $(t, \text{atomic}([\ell] := v); \kappa(x)) \in T$</p> | $\frac{(\ell, v_0) \in h}{\langle h, T \rangle \xrightarrow{t, \ell, a} \langle h[\ell := v], T[t := \kappa] \rangle}$ |
| <p>S-ATOMIC-COMPAREANDMUTATE-EQ $(t, \text{let } x := \text{atomic}(\text{if } [\ell] = v_1 \text{ then } [\ell] := v_2; [\ell]) \text{ in } \kappa(x)) \in T$</p> | $\frac{(\ell, v_1) \in h}{\langle h, T \rangle \xrightarrow{t, \ell, a} \langle h[\ell := v_2], T[t := \kappa(v_1)] \rangle}$ |
| <p>S-ATOMIC-COMPAREANDMUTATE-NEQ $(t, \text{let } x := \text{atomic}(\text{if } [\ell] = v_1 \text{ then } [\ell] := v_2; [\ell]) \text{ in } \kappa(x)) \in T$</p> | $\frac{(\ell, v_0) \in h \quad v_0 \neq v_1}{\langle h, T \rangle \xrightarrow{t, \ell, a} \langle h, T[t := \kappa(v_0)] \rangle}$ |
| <p>S-ASSERT $(t, \text{assert true}; \kappa) \in T$</p> | $\frac{}{\langle h, T \rangle \rightsquigarrow \langle h, T[t := \kappa] \rangle}$ |
| <p>S-FORK $(t, \text{fork}(c); \kappa) \in T$</p> | $\frac{t' \notin \text{dom}(T)}{\langle h, T \rangle \rightsquigarrow \langle h, T[t := \kappa][t' := c; \text{done}] \rangle}$ |

Figure 16. Step rules

over real fractional heaps and ghost fractional heaps.

$$\begin{aligned} \text{AtomicInvs} &= \text{FracHeaps} \times \text{FracGhostHeaps} \rightarrow \text{bool} \\ \text{AtomicPermSets} &= \text{AtomicInvs} \rightarrow \mathbb{R}^+ \end{aligned}$$

The sum of two sets of atomic space permissions is the point-wise sum.

We abbreviate a triple (h, g, A) as a *resource bundle* (or *bundle* for short) s . We define $s_1 + s_2$ as $(h_1 + h_2, g_1 + g_2, A_1 + A_2)$.

The meaning of the assertion syntaxes used in this paper is as follows:

$$\begin{aligned}
h, g, A \models \mathbf{emp} &\Leftrightarrow h = \emptyset \wedge g = \emptyset \wedge A = (\lambda I \bullet 0) \\
h, g, A \models \ell \xrightarrow{\pi} v &\Leftrightarrow h = \{(\ell, (\pi, v))\} \wedge g = \emptyset \wedge A = (\lambda I \bullet 0) \\
h, g, A \models \ell_g \xrightarrow{\pi} v &\Leftrightarrow h = \emptyset \wedge g = \{(\ell_g, (\pi, v))\} \wedge A = (\lambda I \bullet 0) \\
h, g, A \models \pi \boxed{I} &\Leftrightarrow h = \emptyset \wedge g = \emptyset \wedge A = (\lambda I \bullet 0)[I := \pi] \\
s \models P * Q &\Leftrightarrow \exists s_1, s_2 \bullet s = s_1 + s_2 \wedge (s_1 \models P) \wedge (s_2 \models Q) \\
s \models \phi &\Leftrightarrow \phi \quad \text{for a pure formula } \phi \\
s \models P \odot Q &\Leftrightarrow (s \models P) \odot (s \models Q) \quad \text{for } \odot \in \{\wedge, \vee, \Rightarrow\}
\end{aligned}$$

4.5 Command validity

In Figure 17, we define a weakest precondition-like predicate transformer that is equivalent to the proof rules, but that is more suitable for formulating the data-race-freedom proof.

Lemma 1. *If $P \Rightarrow Q$, then $P \Rightarrow \text{modGhost}(Q)$.*

Proof. By induction on the derivation of the premise. \square

Lemma 2 (Correctness implies validity). *If $\{P\} c \{Q\}$, then $P * R \Rightarrow \text{modGhost}(\text{valid}(c, \text{modGhost}(Q * R)))$.*

Proof. By induction on the derivation of the premise. \square

4.6 Configuration validity

A full ghost heap is a finite map from ghost addresses to values:

$$\text{FullGhostHeaps} = (\mathbb{Z}^+ \times \mathbb{Z}^+) \rightarrow_{\text{fin}} \mathbb{Z}$$

A multiset of atomic invariants is a finite map from atomic invariants to positive integers:

$$\text{Multi}(\text{AtomicInvs}) = \text{AtomicInvs} \rightarrow_{\text{fin}} \mathbb{Z}_0^+$$

An atomic space id is a pair of an atomic invariant and a non-negative integer. An atomic space id (I, k) is valid wrt a multiset of atomic invariants A if $k < A(I)$:

$$\text{ids}(A) = \{(I, k) \mid \exists n \bullet (I, n) \in A \wedge k < n\}$$

A configuration $\langle h, T \rangle$ is *valid* if there exists a full ghost heap g , a multiset A of atomic invariants, a map s_T from the domain of T to bundles, and a map s_A from $\text{ids}(A)$ to bundles, such that

$$\forall \ell \notin \text{dom}(h), \ell' \bullet (\ell, \ell') \notin \text{dom}(g)$$

and

$$\langle h, g, A \rangle = \sum_{t \in \text{dom}(T)} s_T(t) + \sum_{i \in \text{ids}(A)} s_A(i)$$

and

$$\forall (t, \kappa) \in T \bullet \text{valid}(\kappa)(s_T(t))$$

and

$$\forall (I, i) \in \text{ids}(A) \bullet s_A((I, i)) \models I$$

The following lemma states that execution of ghost commands by a thread t_0 preserves validity of a configuration.

Lemma 3. *Consider a configuration $\langle h, T \rangle$, a thread id $t_0 \in \text{dom}(T)$, an assertion Q , a bundle s_0 , and a sequence of ghost commands \bar{c} . If there exists a full ghost heap g , a multiset A of atomic invariants, a map s_T from the domain of T to bundles, and a map s_A from $\text{ids}(A)$ to bundles, and a bundle s' such that*

$$s' = \langle h, g, A \rangle + s_0$$

and

$$s' = \sum_{t \in \text{dom}(T)} s_T(t) + \sum_{i \in \text{ids}(A)} s_A(i)$$

and

$$\text{gsvalid}(\bar{c}, Q)(s_T(t_0))$$

and

$$\forall (t, \kappa) \in T \bullet t \neq t_0 \Rightarrow \text{valid}(\kappa)(s_T(t))$$

and

$$\forall (I, i) \in \text{ids}(A) \bullet s_A((I, i)) \models I$$

, then there exists a full ghost heap g , a multiset A of atomic invariants, a map s_T from the domain of T to bundles, and a map s_A from $\text{ids}(A)$ to bundles, and a bundle s' such that

$$s' = \langle h, g, A \rangle + s_0$$

and

$$s' = \sum_{t \in \text{dom}(T)} s_T(t) + \sum_{i \in \text{ids}(A)} s_A(i)$$

and

$$Q(s_T(t_0))$$

and

$$\forall (t, \kappa) \in T \bullet t \neq t_0 \Rightarrow \text{valid}(\kappa)(s_T(t))$$

and

$$\forall (I, i) \in \text{ids}(A) \bullet s_A((I, i)) \models I$$

.

Proof. By induction on \bar{c} . \square

Lemma 4. *Execution steps preserve configuration validity.*

Proof. By case analysis on the step rule.

- **Case S-ATOMIC-LOOKUP.** Validity tells us that there is an atomic space invariant I for which we have some fractional permission π . It follows that $A(I) > 0$, where A is the multiset of atomic invariants. We remove the atomic space with id $(I, A(I) - 1)$ from the multiset and add its bundle to that of the current thread. We also remove the atomic space permission from the thread's bundle. Notice that the resulting augmented configuration is not valid: there is some fraction $1 - \pi$ of atomic space permission left in the threads' bundles, that does not correspond to any atomic spaces that are actually present. However, since ghost commands act only on full atomic space permissions, and no nested atomic commands are allowed, this fraction cannot be used by the program. Lemma 3 is applied using this extraneous permission as the value for s_0 .

After executing the body of the atomic command, we remove some bundle that satisfies I from the current thread's bundle, we add an atomic space for I to the multiset of atomic spaces, and we assign the removed bundle to this atomic space. Finally, we yield the fraction π of the atomic space permission back to the bundle of the current thread.

- The other atomic cases are analogous.
- The other cases are easy. \square

Lemma 5. *A valid configuration is not racy.*

Proof. By contradiction. Consider a configuration C and suppose it is racy. Suppose we have

$$C \xrightarrow{t_1: \ell_1, k_1} C' \xrightarrow{t_2: \ell_2, k_2}$$

Since $t_1 \neq t_2$, the continuation of t_2 in C' is the same as in C . By validity of C , there exists a partition of the resources of the system (the heap, the ghost heap, and the atomic spaces) such that each thread's continuation is valid in its resource bundle and each atomic space's invariant holds in its resource bundle.

$$\begin{array}{l}
\text{fracAlloc } h, \ell, \epsilon, h \quad \frac{\ell \notin \text{dom}(h) \quad \text{fracAlloc } h, \ell + 1, \bar{v}, h'}{\text{fracAlloc } h, \ell, v \cdot \bar{v}, h'[\ell := (1, v)]} \quad \text{gAlloc } g, \ell, k, \epsilon, g \quad \frac{\text{gAlloc } g, \ell, k + 1, \bar{v}, g'}{\text{gAlloc } g, \ell, k, v \cdot \bar{v}, g'[(\ell, k) := (1, v)]} \\
\\
\text{gvalid}(\mathbf{gcons}(v), Q) \triangleq \forall \ell \bullet (0, \ell) \notin \text{dom}(g) \Rightarrow Q(h, g[(0, \ell) := (1, v)], A) \\
\text{gvalid}([\ell_g] := v, Q) \triangleq \exists v_0 \bullet (\ell_g, (1, v_0)) \in g \wedge Q(h, g[\ell_g := (1, v)], A) \\
\text{gvalid}(\mathbf{createAtomic}(I), Q) \triangleq \exists s_I, s' \bullet (h, g, A) = s_I + s' \wedge s_I \models I \wedge Q(s'[A := A[I := A(I) + 1]]) \\
\text{gvalid}(\mathbf{disposeAtomic}(I), Q) \triangleq A(I) \geq 1 \wedge \forall s_I, s' \bullet s' = (h, g, A) + s_I \wedge s_I \models I \Rightarrow Q(s'[A := A[I := A(I) - 1]]) \\
\text{gsvalid}(\epsilon, Q) \triangleq Q(h, g, A) \\
\text{gsvalid}(c \cdot \bar{c}, Q) \triangleq \text{gvalid}(c, \text{gsvalid}(\bar{c}, Q)) \\
\text{modGhost}(Q) \triangleq \exists \bar{c} \bullet \text{gsvalid}(\bar{c}, Q) \\
\\
\text{valid}(\mathbf{cons}(\bar{v}), Q) \triangleq \exists \bar{v}' \bullet \forall \ell, h', g' \bullet \text{fracAlloc } h, \ell, \bar{v}, h' \wedge \text{gAlloc } g, \ell, 0, \bar{v}', g' \Rightarrow Q(\ell)(h', g', A) \\
\text{valid}([\ell], Q) \triangleq \exists v \bullet (\ell, (1, v)) \in h \wedge Q(v)(h, g, A) \\
\text{valid}([\ell] := v, Q) \triangleq \exists v_0 \bullet (\ell, (1, v_0)) \in h \wedge Q(h[\ell := (1, v)], g, A) \\
\text{valid}(\mathbf{let } x := c \mathbf{ in } c'(x), Q) \triangleq \text{valid}(c, \text{modGhost}(\text{valid}(c'(x), Q))) \\
\text{valid}(\mathbf{return } v, Q) \triangleq Q(v)(h, g, A) \\
\text{valid}(\mathbf{atomic}(c), Q) \triangleq \\
\quad \exists I, \pi \in \mathbb{R}_0^+ \bullet A(I) \geq \pi \wedge \\
\quad \forall s_I, s' \bullet s_I \models I \wedge s' = s_I + (h, g, A[I := A(I) - \pi]) \Rightarrow \\
\quad \text{modGhost}(\text{valid}(c, \text{modGhost}(\lambda h, g, A \bullet \\
\quad \exists s_I, s' \bullet s_I \models I \wedge (h, g, A) = s_I + s' \wedge Q(s'[A := A[I := A(I) + \pi]]))) \\
\text{valid}(\mathbf{assert } b, Q) \triangleq b \wedge Q(h, g, A) \\
\text{valid}(\mathbf{fork}(c), Q) \triangleq \exists s_F, s' \bullet (h, g, A) = s_F + s' \wedge \text{modGhost}(\text{valid}(c, (\lambda _ \bullet \mathbf{true}))) (s_F) \wedge Q(s') \\
\\
\text{valid}(\mathbf{done}) \triangleq \mathbf{true} \\
\text{valid}(\mathbf{let } x := c \mathbf{ in } \kappa(x)) \triangleq \text{valid}(c, \text{modGhost}(\text{valid}(\kappa(x))))
\end{array}$$

Figure 17. Command validity

We know that at least one of the accesses is a non-atomic access. It follows that its thread's resource bundle has full permission on ℓ . If the other access is also a non-atomic access, the contradiction follows immediately.

If the other access is an atomic access, then the proof is complicated by the fact that a sequence of ghost operations (ghost field creations, ghost field mutations, atomic space creations, and atomic space disposals) intervenes between entering the atomic command and performing the access. However, we can prove the invariant, moving backwards in time from the point of the access to before we enter the atomic command, that if we combine the resource bundle of the thread that performs the atomic access and the resource bundles of all atomic spaces in the system, we obtain full permission on ℓ . Since this resulting resource bundle is supposed to be disjoint from the resource bundle of the thread performing the non-atomic access, we obtain the contradiction. \square

Theorem 1 (Soundness). *If a program c is correct ($\{\mathbf{true}\} c \{\mathbf{true}\}$), then it is data-race-free.*

Proof. By Lemma 2, correctness of c implies validity of c . It follows that the initial configuration is valid. By Lemma 4, any reachable configuration is valid. Thus, by Lemma 5, no reachable configuration is racy. \square

5. Verification tool

Modular verification per our approach of lock-free data structure implementations written in the C programming language

is supported by our verification tool prototype VeriFast (Jacobs and Piessens 2008). In particular, we used VeriFast to verify a C implementation of the queue module, as well as a C implementation of the example client program. The tool and the example files may be downloaded from the VeriFast website at <http://www.cs.kuleuven.be/~bartj/verifast/>.

Calling an atomic operation on a lock-free data structure requires a proof that the atomic space invariant contains the data structure's state, and that the operation preserves the atomic space invariant. This is encoded in VeriFast using *lemma function pointers*. *Lemma functions* in VeriFast are like ordinary C functions, except that VeriFast checks that they have no effect on non-ghost state and that they terminate. It follows that a verified lemma function constitutes a proof that its precondition ghost-implies (\Rightarrow) its post-condition. (The current implementation does not directly support ghost-equivalence (\Leftrightarrow), so a separate lemma is needed for each direction of the arrow.) Furthermore, VeriFast supports C function pointers and calls through C function pointers, using a simple form of *predicate families* (Parkinson and Bierman 2005) indexed by function name. We have combined these two mechanisms to yield lemma function pointers. The caller of an atomic operation must supply the required proofs in the form of lemma function pointers.

Ensuring termination of lemma functions that perform calls through lemma function pointers requires some care. The approach adopted in VeriFast is as follows. Performing a statically-bound call requires that the callee appears before the caller in the program text, or that the callee equals the caller and the declared measure decreases. Performing a call on a lemma function pointer requires

that the caller owns a *lemma function pointer call permission*. Such permissions are tracked in the same way as points-to permissions. Furthermore, the permission is temporarily unavailable during the call. A lemma function pointer call permission for a lemma function f may be generated only by a function that appears after f in the program text. This function must then pass this permission to the function that will perform the dynamically-bound call.

It can be shown easily that this ensures lemma function termination. By contradiction. Consider an infinite lemma function execution. Since loops are not allowed in lemma functions, the call stack must be infinite. Since there are finitely many lemma functions, there must be a cycle in the call graph. Consider the node n in the graph that appears latest in the program text. The incoming edges into this node must be performed through a lemma function pointer. Therefore, they must consume a lemma function pointer call permission for n . However, these permissions may be generated only by functions that appear after n , so they must have been generated before the cycle was entered. But when the cycle was entered, only a finite number of lemma function call permissions could have been generated.

6. Related work

In this section, we provide a preliminary survey of related work. Pending a more thorough survey, to the best of our knowledge, we show here for the first time an approach for performing modular full functional specification and verification of partial correctness of dynamically allocated lock-free data structures and their client code, that achieves information hiding and that enables building lock-free data structures on top of other lock-free data structures. Furthermore, we describe how we implemented a verification tool for the approach.

In his PhD thesis, Vafeiadis (2007) proposes an approach for verifying full functional partial correctness of lock-free data structures using a combination of RGSep and linearizability proofs. However, he does not show how to verify full functional partial correctness of client code that uses these data structures. Furthermore, our approach seems simpler, in that it is based only on simple atomic spaces, ghost fields, fractional permissions, and ghost implication. On the other hand, we have not yet validated our approach as widely as was done by Vafeiadis, so it is not clear that our approach applies as widely as his.

Work on combining separation logic and rely-guarantee reasoning (Vafeiadis and Parkinson 2007; Feng et al. 2007; Feng 2009; Dodds et al. 2009) deals with lock-free data structures, but does not show how to specify full functional partial correctness of such data structures, in a way that preserves information hiding. However, we believe an approach like Local Rely-Guarantee (Feng 2009) may usefully be combined with our approach in cases where the Owicki-Gries-like approach with ghost fields by itself is not sufficiently powerful or convenient. Specifically, we would use LRG inside the atomic space invariant to describe the constraints on the ghost fields (rather than applying LRG to the entire atomic space, which would complicate the operations' specifications).

Vafeiadis et al. (2006) prove by hand a number of highly concurrent set implementations using rely-guarantee.

Parkinson et al. (2007) specify and verify a non-blocking stack implementation and verifies a client program as well; however, the specification does not specify full functionality. On the other hand, it does hide the presence of inter-thread interference. In our approach, this can be achieved by including the atomic space within the abstract predicate that describes the data structure, but then the specification is no longer precise and no longer enables building higher-level lock-free data structures.

Gotsman et al. (2007) extend separation logic with support for dynamic locks and threads. They support locks that contain

lock permissions themselves, by introducing a level of indirection through lock invariant names. For simplicity, in this paper atomic space invariants are inline and cannot contain atomic space permissions; however, by introducing named atomic space invariants, we believe we can lift this limitation.

We do not address verification of progress properties in this work; existing work in this area includes Gotsman et al. (2009).

7. Conclusion

We proposed an approach for specifying and verifying full functional partial correctness of lock-free data structures, in a way that achieves modularity, compositionality, and information hiding. We demonstrated the approach by showing a specification, a verified implementation, and a verified client program for a lock-free cell data structure and a lock-free queue data structure. We proved the soundness of the program logic used. We described how we implemented the approach in VeriFast, our prototype C program verification tool.

Immediate future work includes performing a more thorough survey of related work and validation on more examples.

References

- Sarita V. Adve and Mark D. Hill. Weak ordering—A new definition. In *Proc. 17th Intl. Symp. Computer Architecture*, 1990.
- Hans Boehm and Sarita Adve. Foundations of the C++ concurrency model. In *PLDI*, 2008.
- Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
- Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, March 2009.
- Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, January 2009.
- Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
- Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzy, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS 2007*, 2007.
- Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that nonblocking algorithms don’t block. In *POPL*, 2009.
- Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
- Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL 2005*, 2005.
- Matthew Parkinson, Richard Bornat, and Peter O’Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007.
- Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, Computer Laboratory, University of Cambridge, 2007.
- Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.
- Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.