A Practical Lock-Free Queue Implementation and its Verification

Lindsay Groves, Simon Doherty

Victoria University of Wellington

Mark Moir, Victor Luchangco

Sun Microsystems, Boston

(FORTE, Madrid, September, 2004)

Motivation/Outline

- Lock-based concurrency doesn't scale
- *Lock-free/non-blocking* offer better performance, but are very subtle!
  - Ø Michael & Scott's queue
- Techniques for verification
  - Ø Simulation between automata
  - Ø Model checking

After a finite number of steps, some operation completes

Lock-Free Algorithms

- Prepare for operation
- Attempt to update
- If unsuccessful, try again

Lock-Free Algorithms

- Prepare for operation
  - Observe shared variables
  - Work with locals
- Attempt to update
- If unsuccessful, try again

Lock-Free Algorithms

- Prepare for operation
  - Observe shared variables
  - Work with locals
- Attempt to update
  - Update if not changed since last read
  - Needs “strong” synchronisation primitive
- If unsuccessful, try again

Synchronisation Primitives

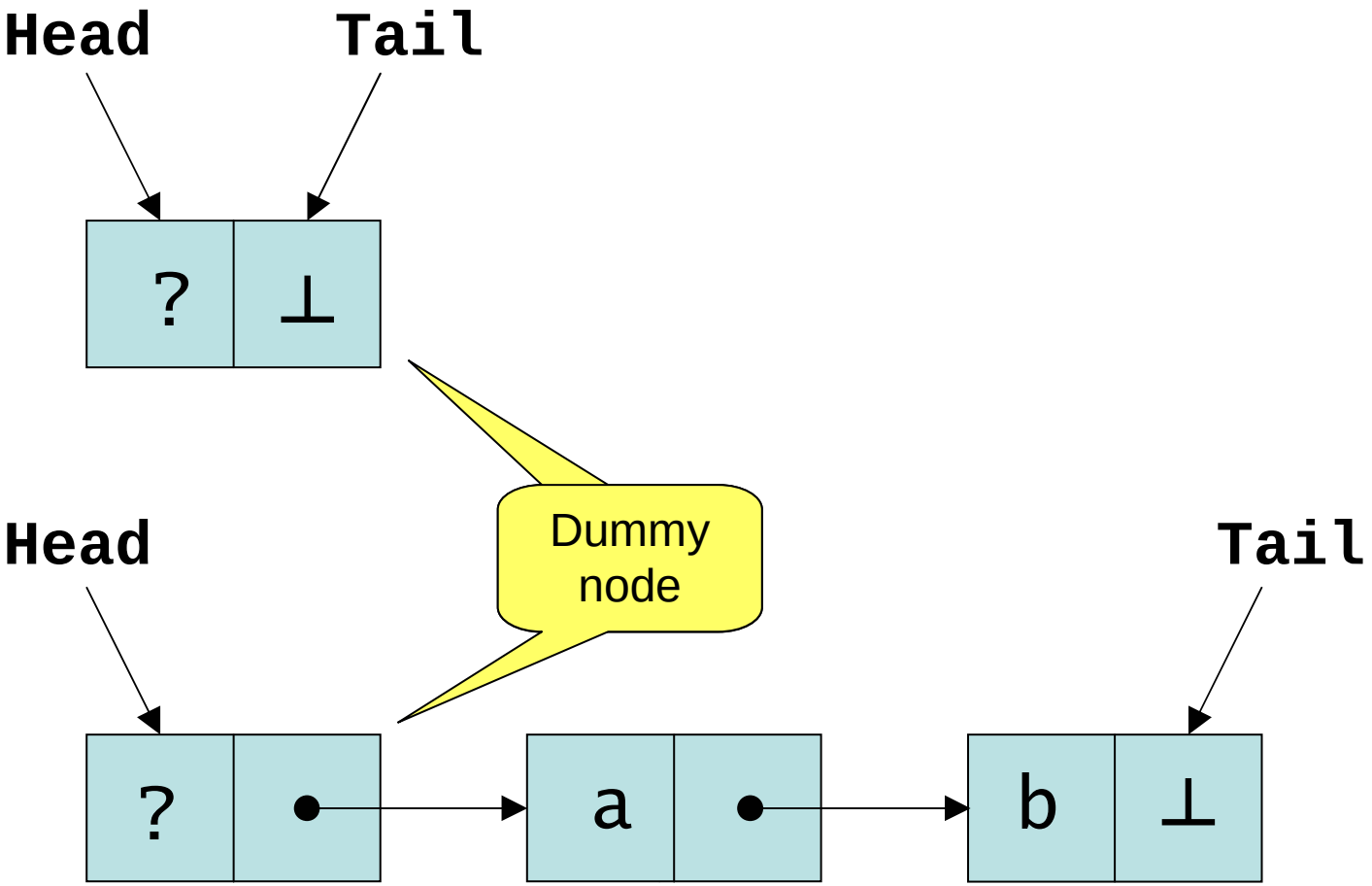
- Load Linked/Store Conditional (LL/SC)
  - LL: Load value
  - SC: Update if not changed since LL
  - Implementations limited
- Compare and Swap (CAS)
  - Update if location contains expected value
  - Doesn't guarantee value hasn't changed!
  - Use version numbers to get around this

ABA

Compare and Swap

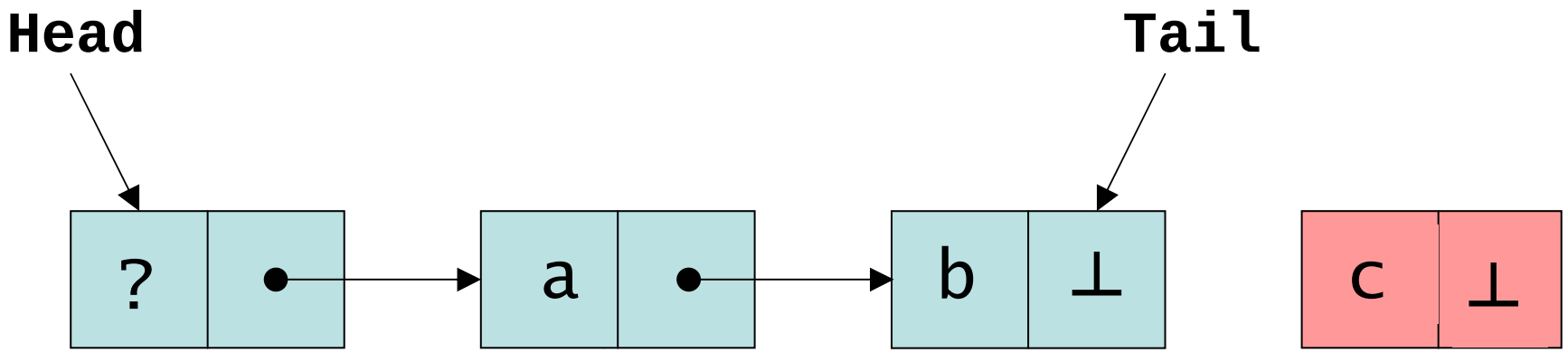
```
boolean CAS(add, old, new) {
  atomically {
    if ( add* == old ) {
      add* = new;
      return true;
    } else
      return false;
  }
}
```

A Linked Queue Implementation



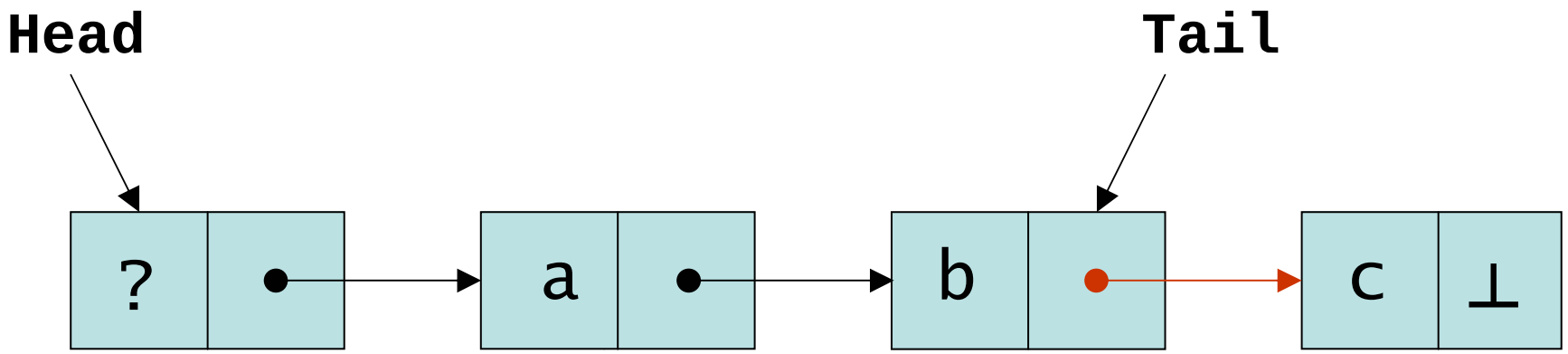
Enqueue

Create and initialise new node:



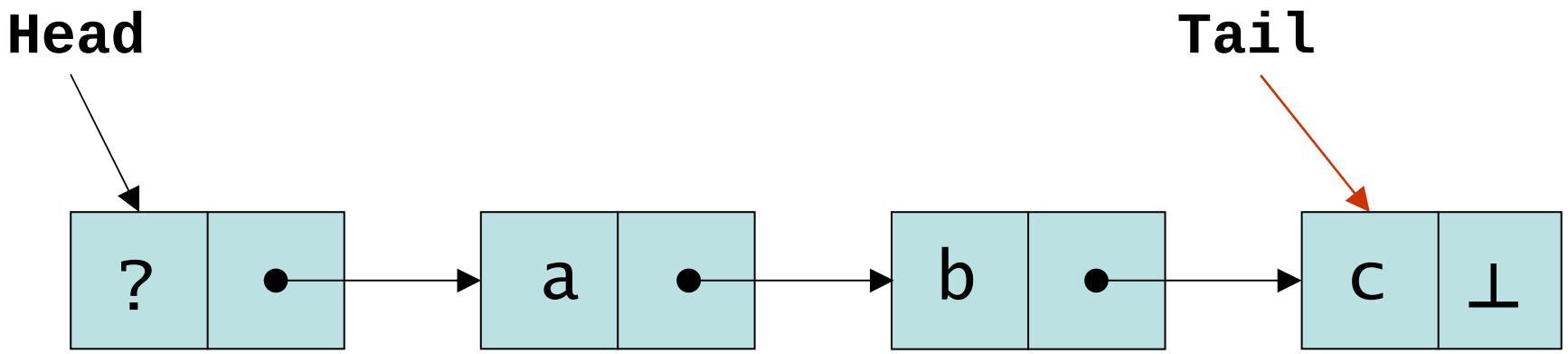
Enqueue

Append new node onto list:



Enqueue

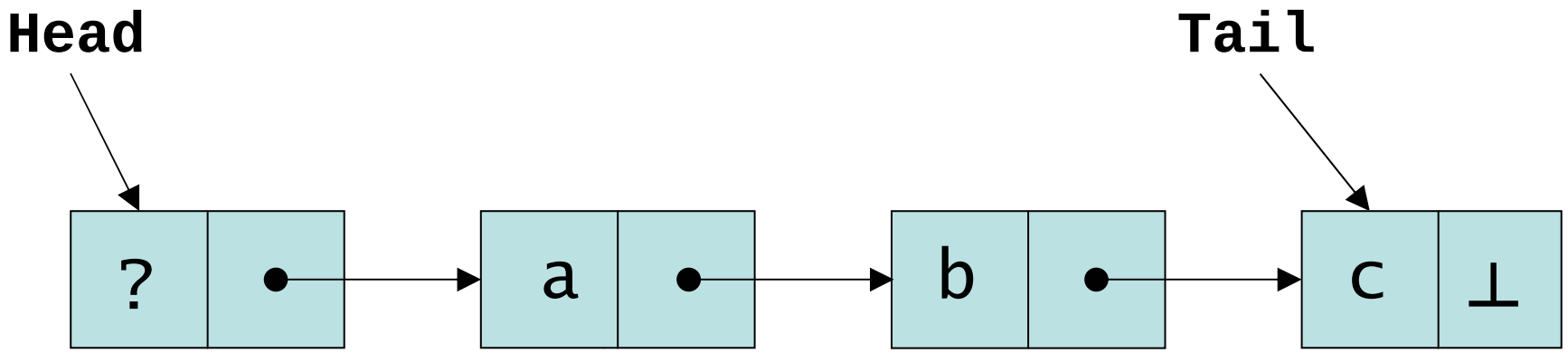
Swing **`Tail`** to point to new node:



Dequeue

If **`Head`** ≠ **`Tail`**,

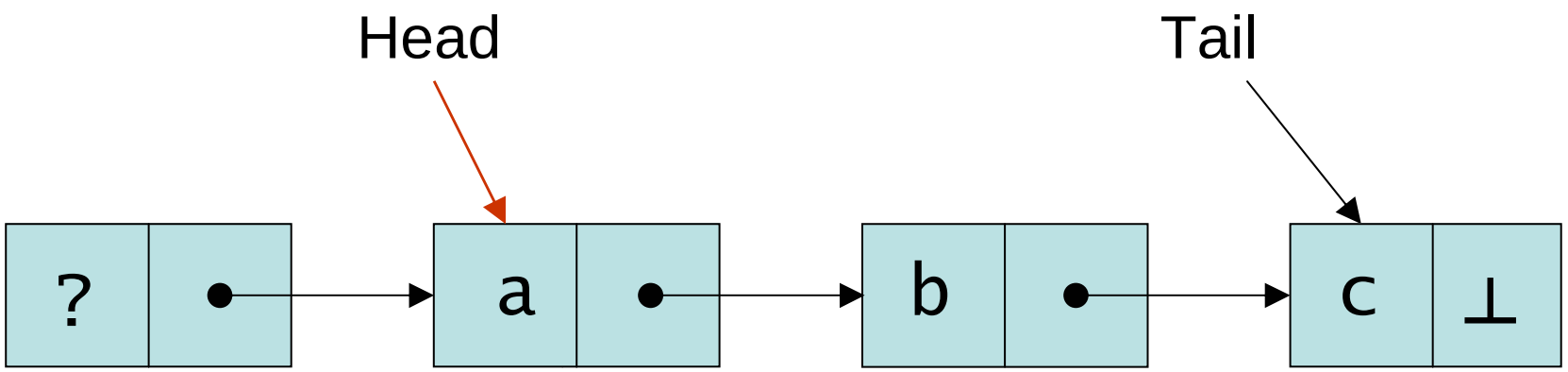
copy value from second node:



Result: a

Dequeue

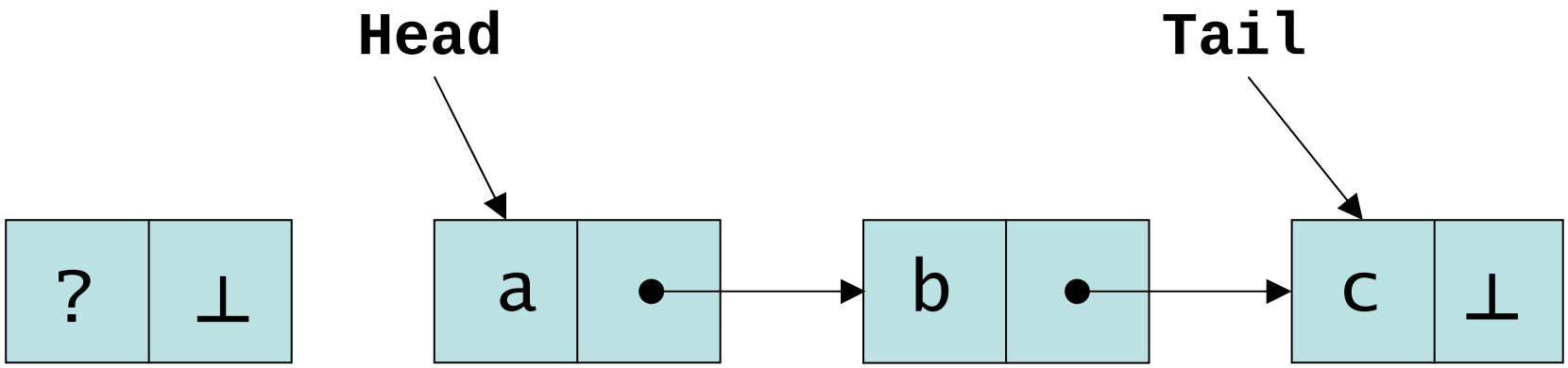
Swing **`Head`** to second node:



Result: a

Dequeue

Unlink old header node and release:



Result: a

Sequential Implementation

Declarations:

```
node = {Data val; node* next};
node* Head, Tail;
```

Initialisation:

```
Head = Tail = new(node);
Head->next  = null;
```

Sequential Implementation

```
void Enqueue(data v) {
  node n = new(node);
  n->val = v;
  n->next = null;
  Tail->next = n;
  Tail = n;
}
```

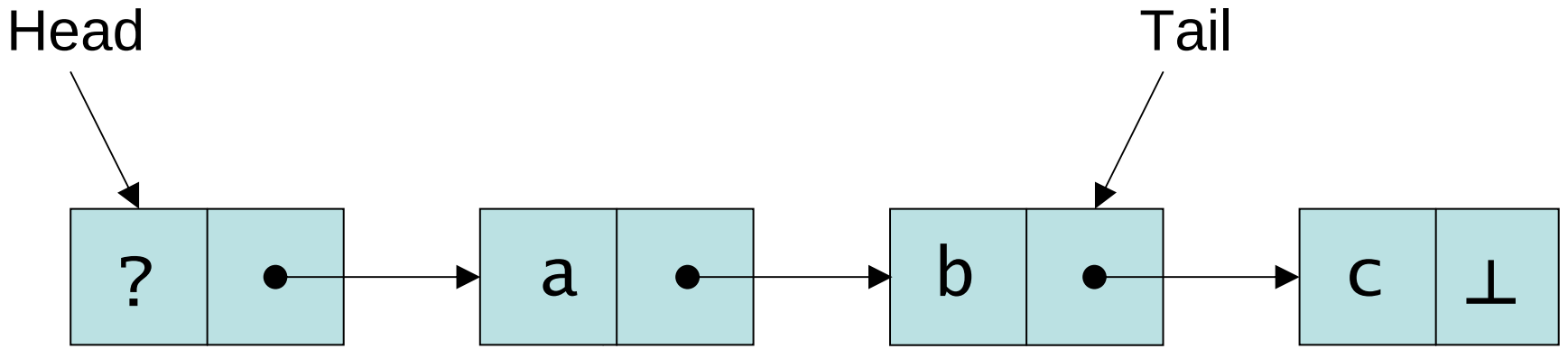
```
data Dequeue() {
  if (Head = Tail)
    return null;
  node h = Head;
  data v = h->next;
  Head = h->next;
  free(h);
  return v;
}
```

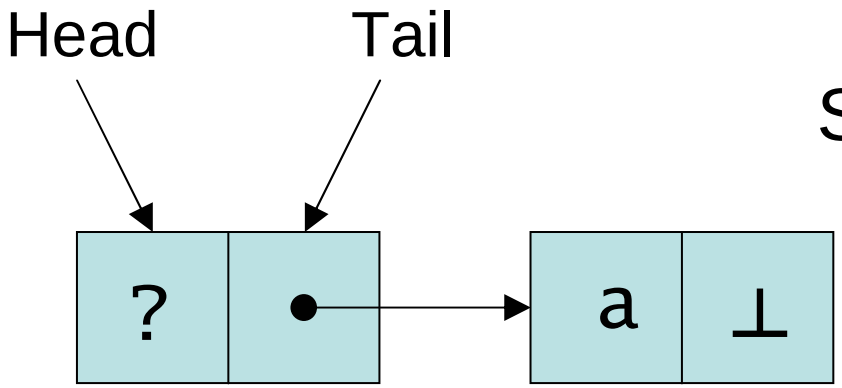
Concurrent Implementation

- Enqueue can't update both **`Tail`** and **`Tail->next`** with single CAS.
- Append new code with CAS.
- Allow **`Tail`** to lag.
- Any enqueuer can advance **`Tail`**.
- Dequeuer has to consider **`Tail`** when queue is empty.

Concurrent Implementation

Tail can lag by at most one





Special case for dequeue

Concurrent Enqueue

```
enqueue(value: data type)
  node = new_node();
  node->value = value; node->next = NULL;
  loop
    tail = Tail; next = tail->next;
    if tail == Tail
      if next == NULL
        if CAS(&tail->next, next, node)
          break;
        endif
      else
        CAS(Tail, tail, next);
      endif
    endif
  endloop
  CAS(Tail, tail, node);
```

Concurrent Enqueue

```
enqueue(value: data type)
  node = new_node();
  node->value = value; node->next = NULL;
  loop
    tail = Tail; next = tail->next;
    if tail == Tail
      if next == NULL
        if CAS(&tail->next, next, node)
          break;
        endif
      else
        CAS(Tail, tail, next);
      endif
    endif
  endloop
  CAS(Tail, tail, node);
```

Allocate new node and initialise

Concurrent Enqueue

```
enqueue(value: data type)
  node = new_node();
  node->value = value; node->next = NULL;
  loop
    tail = Tail; next = tail->next;
    if tail == Tail
      if next == NULL
        if CAS(&tail->next, next, node)
          break;
        endif
      else
        CAS(Tail, tail, next);
      endif
    endif
  endloop
  CAS(Tail, tail, node);
```

Try to update **Tail**, till succeed

Concurrent Enqueue

```
enqueue(value: data type)
  node = new_node();
  node->value = value; node->next = NULL;
  loop
    tail = Tail; next = tail->next;
    if tail == Tail
      if next == NULL
        if CAS(&tail->next, next, node)
          break;
        endif
      else
        CAS(Tail, tail, next);
      endif
    endif
  endloop
  CAS(Tail, tail, node);
```

Try to update **Tail**, till succeed

Concurrent Enqueue

```
enqueue(value: data type)
  node = new_node();
  node->value = value; node->next = NULL;
  loop
    tail = Tail; next = tail->next;
    if tail == Tail
      if next == NULL
        if CAS(&tail->next, next, node)
          break;
        endif
      else
        CAS(Tail, tail, next);
      endif
    endif
  endloop
  CAS(Tail, tail, node);
```

Update **`Tail`** if it's trailing. Can be done at beginning, or by separate process.

Concurrent Dequeue

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head == tail
        if (next == NULL) return FALSE; endif
        CAS(Tail, tail, next);
      else
        *pvalue = next->value;
        if CAS(Head, head, next)
          break;
        endif
      endif
    endif
  endloop
  free(head);
  return TRUE;
```

Keep trying till operation succeeds

Concurrent Dequeue

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head == tail
        if (next == NULL) return FALSE; endif
        CAS(Tail, tail, next);
      else
        *pvalue = next->value;
        if CAS(Head, head, next)
          break;
        endif
      endif
    endif
   endloop
   free(head);
   return TRUE;
```

Return **false** if queue is empty

Concurrent Dequeue

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head == tail
        if (next == NULL) return FALSE; endif
        CAS(Tail, tail, next);
      else
        *pvalue = next->value;
        if CAS(Head, head, next)
          break;
        endif
      endif
    endif
   endloop
   free(head);
   return TRUE;
```

Update **Tail** if it's trailing

Concurrent Dequeue

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head == tail
        if (next == NULL) return FALSE; endif
        CAS(Tail, tail, next);
      else
        *pvalue = next->value;
        if CAS(Head, head, next)
          break;
        endif
      endif
    endif
   endloop
   free(head);
   return TRUE;
```

Save node value to return

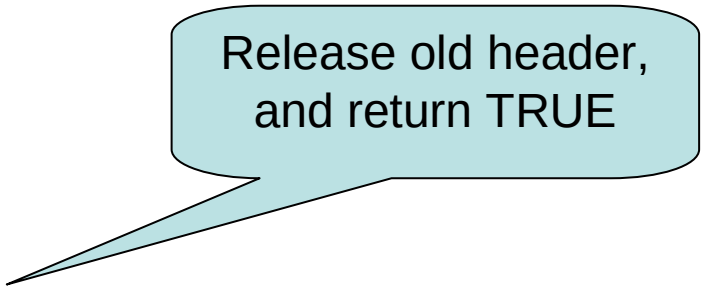
Concurrent Dequeue

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head == tail
        if (next == NULL) return FALSE; endif
        CAS(Tail, tail, next);
      else
        *pvalue = next->value;
        if CAS(Head, head, next)
          break;
        endif
      endif
    endif
   endloop
   free(head);
   return TRUE;
```

Update Head if it hasn ‘t change

Concurrent Dequeue

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head == tail
        if (next == NULL) return FALSE; endif
        CAS(Tail, tail, next);
      else
        *pvalue = next->value;
        if CAS(Head, head, next)
          break;
        endif
      endif
    endif
   endloop
   free(head);
   return TRUE;
```



Handling Dynamic Storage

- Can't release dequeued node to system – another process may still have a pointer to it.
- Dequeue returns nodes to a free-list.
- Enqueue allocates nodes from the free-list if possible.
- Implementation is not *population oblivious*.

ABA and Version Numbers

- CAS doesn't guarantee that value hasn't changed - A B A.
- The rest of queue may be different.
- Attach version numbers to pointers and increment every time pointer is modified.
- Can store pointer and count in 64 bit word.
- Likelihood of mistake is minute.
- We assume version numbers unbounded!

Enqueue with Counts

```
enqueue(value: data type)
  node = new_node();
  node->value = value; node->next.ptr = NULL;
  loop
    tail = Tail; next = tail.ptr->next;
    if tail == Tail
      if next.ptr == NULL
        if CAS(&tail.ptr->next, next,
               <node, next.count+1>)
          break;
        endif
      else
        CAS(Tail, tail, <next.ptr, tail.count+1>);
      endif
    endif
  endloop
  CAS(Tail, tail, <node, tail.count+1>);
```

Michael & Scott, 1996

Dequeue with Counts

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; tail = Tail;
    next = head->next;
    if head == Head
      if head.ptr == tail.ptr
        if (next.ptr == NULL) return FALSE; endif
        CAS(Tail, tail, <next.ptr, tail.count+1>);
      else
        *pvalue = next.ptr->value;
        if CAS(Head, head, <next.ptr, head.count+1>)
          break;
        endif
      endif
    endif
  endloop
  free(head.ptr);
  return TRUE;
```

A Small Optimisation

```
dequeue(pvalue: pointer to data type): boolean
  loop
    head = Head; next = head->next;
    if head == Head
      if (next.ptr == NULL) return FALSE;
      else
        *pvalue = next.ptr->value;
        if CAS(Head, head, <next.ptr, head.count+1>)
          tail = Tail;
          if head.ptr == tail.ptr
            CAS(Tail, tail, <next.ptr, tail.count+1>);
          endif
          break;
        endif
      endif
    endif
  endloop
  free(head.ptr);
  return TRUE;
```

Only check special case after swinging **Head** successfully

Verification

For sequential implementation:

- Define function from representation to abstract queue values:

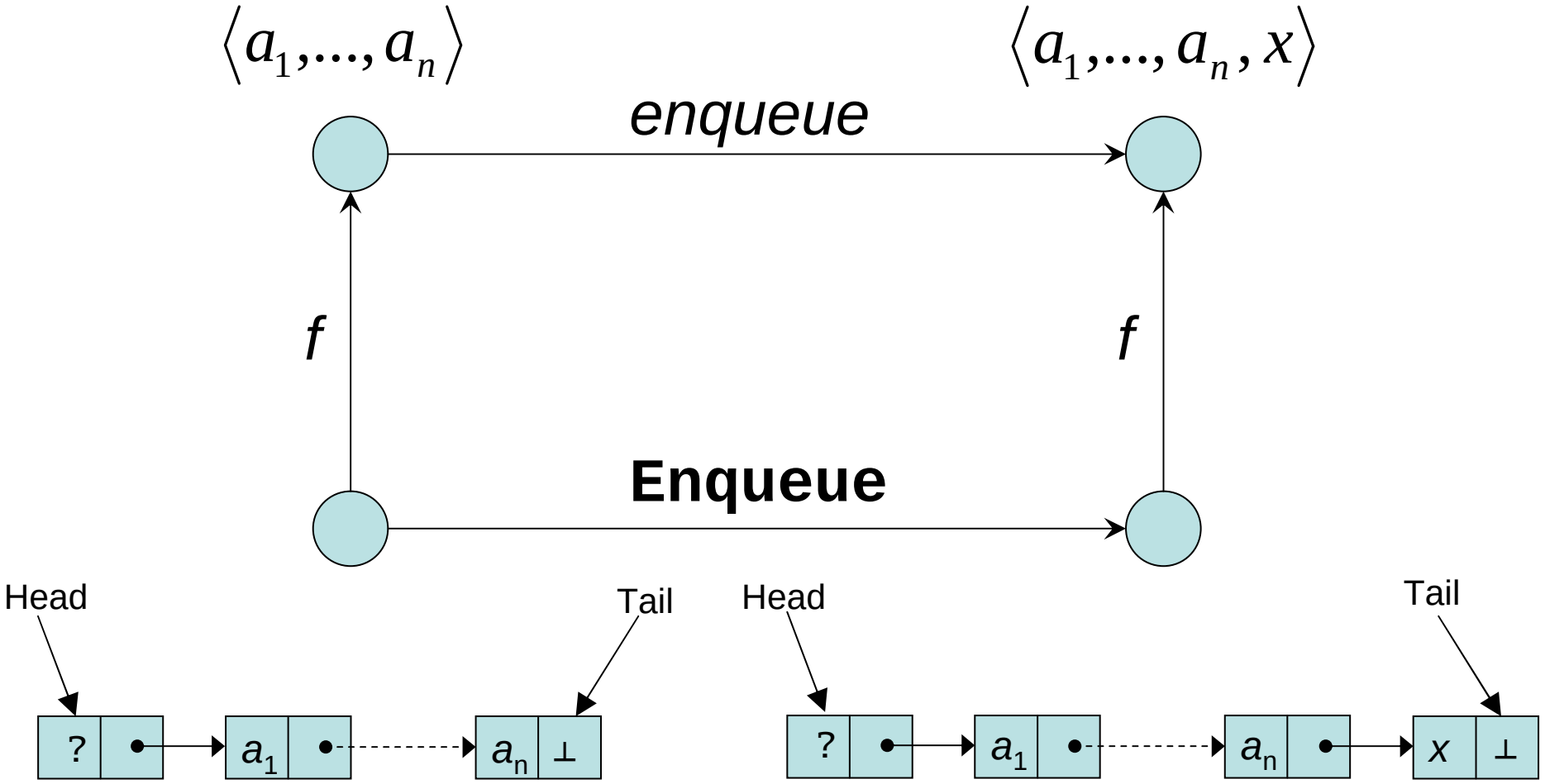
  f: *List* → *Queue*

  Hoare, 1972
  In general, may need relation, but function works here.

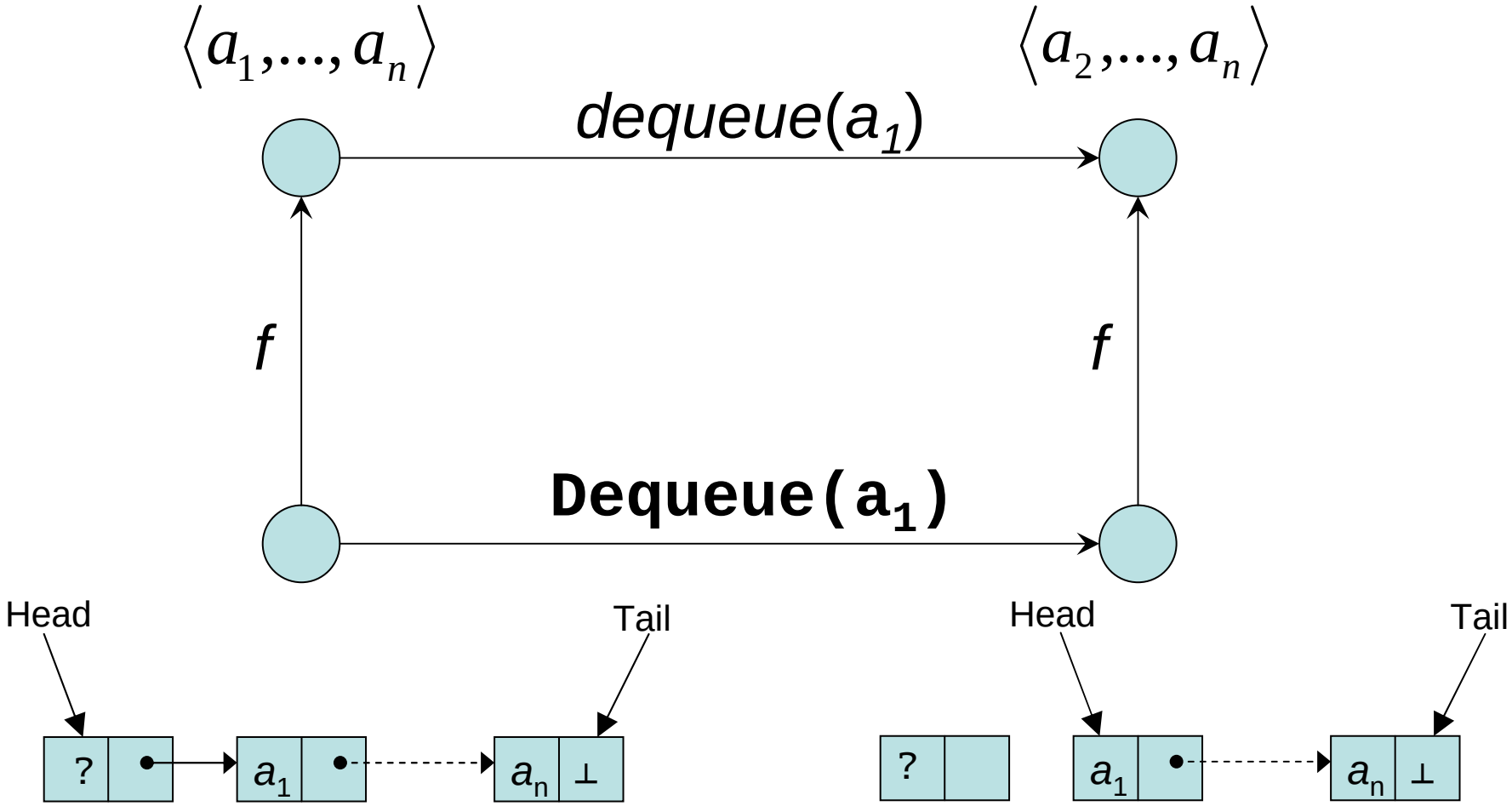
- Initial state represents empty queue:

  $f(InitList) = emptyQueue$

Verification: Enqueue preserves *f*



Verification: Dequeue preserves *f*



Verification

- For concurrent versions, must consider all possible interleavings of atomic actions.
- Prove *linearisability*: Every operation appears to occur at some point between its invocation and its response.
- Model specification and implementation as *Input/Output Automata*.
- Prove trace inclusion by simulation.

Abstract Automaton (AbsAut)

External actions:

- $enq\_inv_p(v)$
- enq_resp_p
- deq_inv_p
- $deq\_res_p\ (r)$

Internal actions:

- do_enq_p
- do_deq_p

State variables:

- Abstract queue

  Updated by do_enq_p and do_deq_p
- Local variables

Generates all linearisable executions

Concrete Automaton (ConcAut)

External actions:

- As for Abstract

Internal actions:

- One for each assignment
- Two for each test

State variables:

- Heap model
- Local variables

Generates all possible executions

Simulation Proof

- Define *simulation relation R* between concrete and abstract IOA.
  - Relates concrete and abstract values (cf *f*).
  - Also relates pc values for both IOAs.
- Show that for every execution of *ConcAut*, there is an execution of *AbsAut* with the same trace (external actions).

Simulation Proof

- Show that for every step of *ConcAut* there is a corresponding sequence of steps of *AbsAut* with the same trace:
  - Match external actions.
  - Identify point in ConcAut where operation takes effect (*linearisation point*), and match with internal action.
  - Rest are stuttering steps.

Simulation Proof

- When dequeue observes **Head** in *ConcAut*, we can't tell whether the dequeue will return "empty".
- Verification requires two steps:
  - *Forward simulation* from *ConcAut* to *IntAut.*
  - *Backward simulation* from *IntAut* to *AbsAut.*
- Intermediate automaton *IntAut* just handles dequeue on empty queue.

Other Verifications

- Michael & Scott, 1996
  - "Prove" several safety properties
  - Note clear what they guarantee
- Yahva & Sagiv, 2003
  - Reduce to finite system and model check
  - Check same properties as Michael and Scott
- Abrial & Cansell, 2004
  - Describe a "formal construction"
  - Lecture slides only – not clear what's done

Other Experience

- Proofs of several algorithms:
  - Stacks, queues and deques
  - Array-based and pointer-based
  - Using CAS and DCAS
- Manual and mechanical proofs (PVS)
- Errors found in two published algorithms
- Optimisation found for one
- Used SPIN to verify bugs and explore variants

What Next?

- Other algorithms
- Composition
  - IOAs limited
  - TLA maybe?
- Other theorem provers: HOL, ...
- Model checking
  - Good for initial testing and exploration
  - Can abstraction give finite systems?