Exploiting Purity for Atomicity

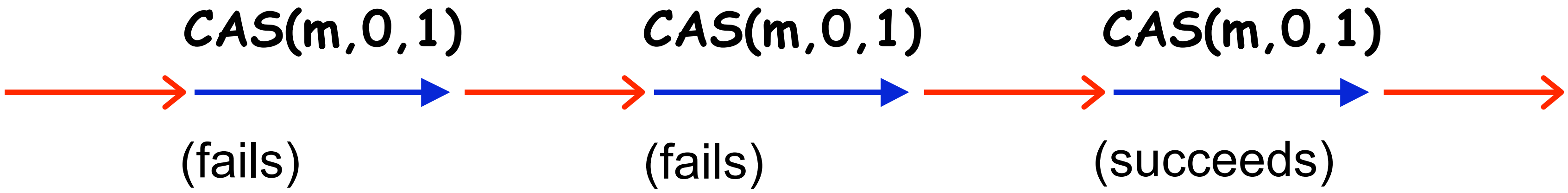
Busy Acquire

```
atomic void busy_acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```

```
if (m == 0) {
  m = 1; return true;
} else {
  return false;
}
```

Busy Acquire

```
atomic void busy_acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```

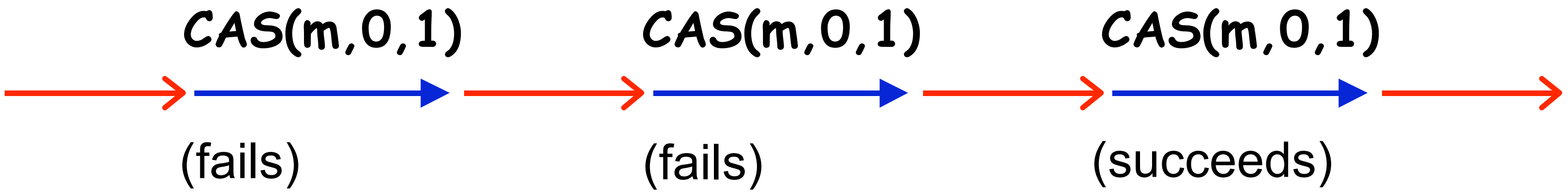


CAS(m,0,1)
(fails)

CAS(m,0,1)
(fails)

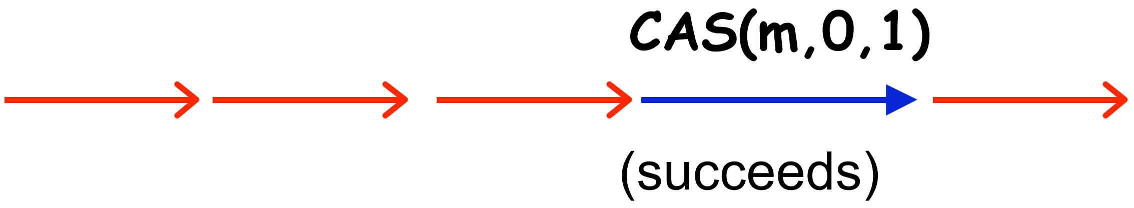
CAS(m,0,1)
(succeeds)

- Non-Serial Execution:



CAS(m,0,1) (fails) **CAS(m,0,1)** (fails) **CAS(m,0,1)** (succeeds)

- Serial Execution:



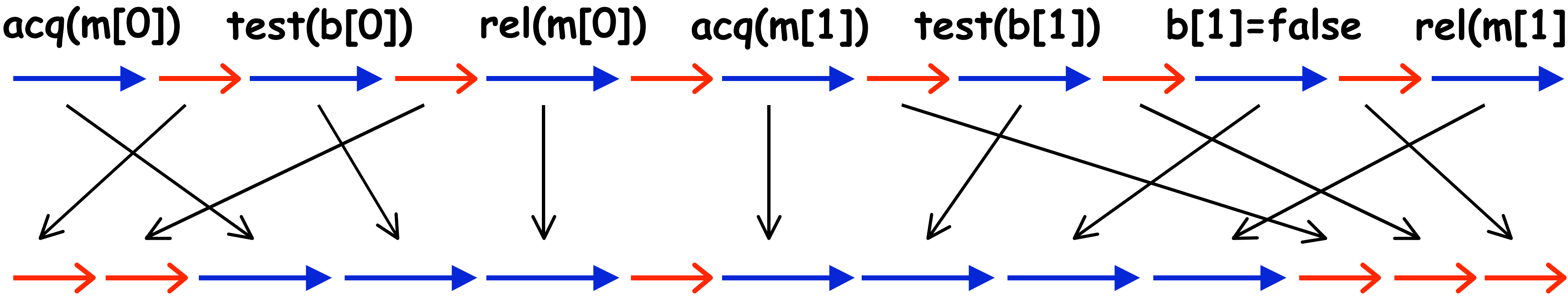
CAS(m,0,1) (succeeds)

- Atomic but not reducible

alloc

```
boolean b[MAX]; // b[i]==true iff block i is free
Lock m[MAX];

atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    acquire(m[i]);
    if (b[i]) {
      b[i] = false;
      release(m[i]);
      return i;
    }
    release(m[i]);
    i++;
  }
  return -1;
}
```

alloc



alloc is not Atomic

- There are non-serial executions with no equivalent serial executions

```
m[0] = m[1] = 0; b[0] = b[1] = false;

t = alloc();   ||   free(0); free(1);
```

```
void free(int i) {
  acquire(m[i]);
  b[i] = true;
  release(m[i]);
}
```

```
m[0] = m[1] = 0; b[0] = b[1] = false;

t = alloc();   ||   free(0); free(1);
```

- Non-Serial Execution:

loop for b[0] → free(0) → free(1) → loop for b[1] → t = 1

- Serial Executions:

loop for b[0] → loop for b[1] → free(0) → free(1) → t = -1

free(0) → free(1) → loop for b[0] → t = 0

free(0) → loop for b[0] → free(1) → t = 0

Extending Atomicity

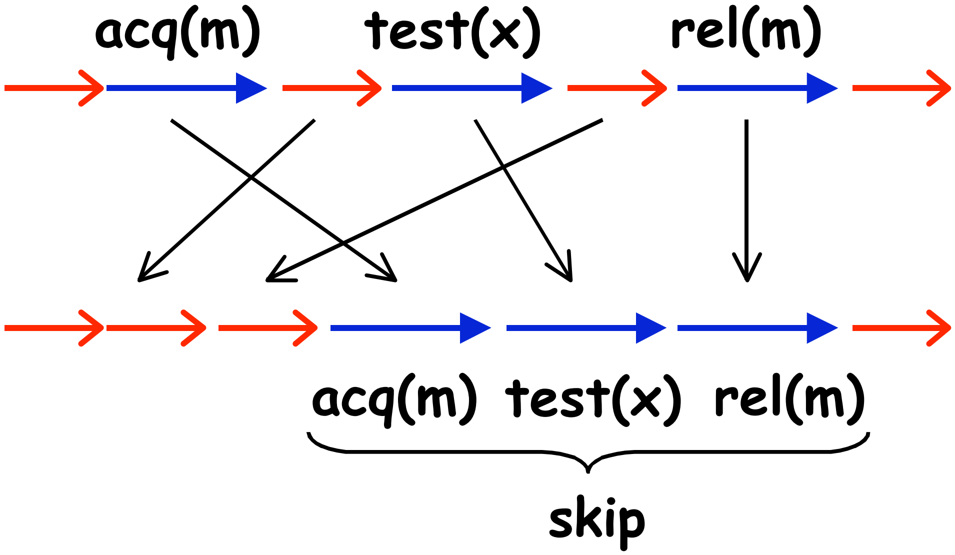
- Atomicity doesn't always hold for methods that are "intuitively atomic"
  - serializable but not reducible (busy_acquire)
  - not serializable (alloc)
- Examples
  - initialization
  - resource allocation
  - wait/notify
  - caches
  - commit/retry transactions

Pure Code Blocks

- Pure block: `pure { E }`
  - `E` is reducible in normally terminating executions
  - If `E` terminates normally, it does not update state visible outside of `E`
- Example

```
while (true) {
  pure {
    acquire(mx);
    if (x == 0) { x = 1; release(mx); break; }
    release(mx);
  }
}
```

Purity and Abstraction



- Abstract execution semantics:
  - treat normal execution of `pure` blocks as the skip statement

Abstraction

- *Abstract semantics* that admits more behaviors
  - pure blocks can be skipped
  - hides "irrelevant" details (ie, failed loop iters)

- Program must still be (sequentially) correct in abstract semantics

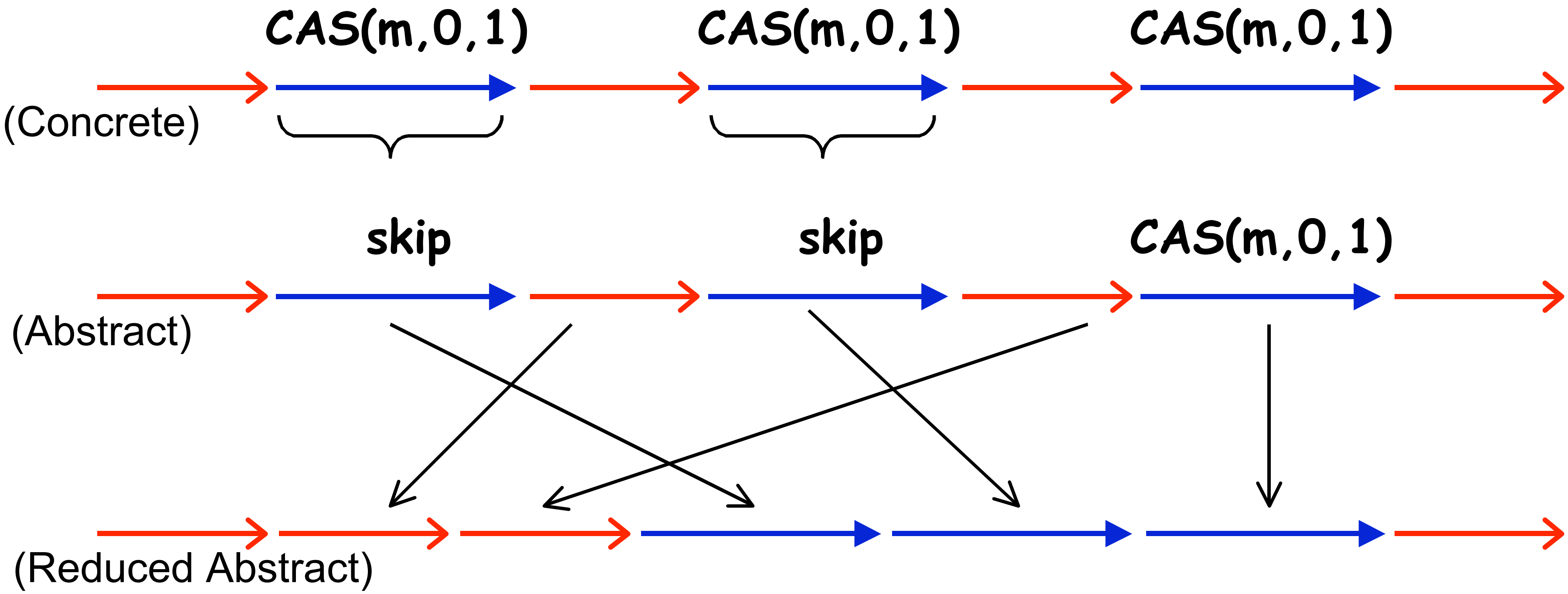
- Abstract semantics make reduction possible

Busy Acquire

```
atomic void busy_acquire() {
  while (true) {
    pure { if (CAS(m,0,1)) break; }
  }
}
```

Abstract Execution of Busy Acquire

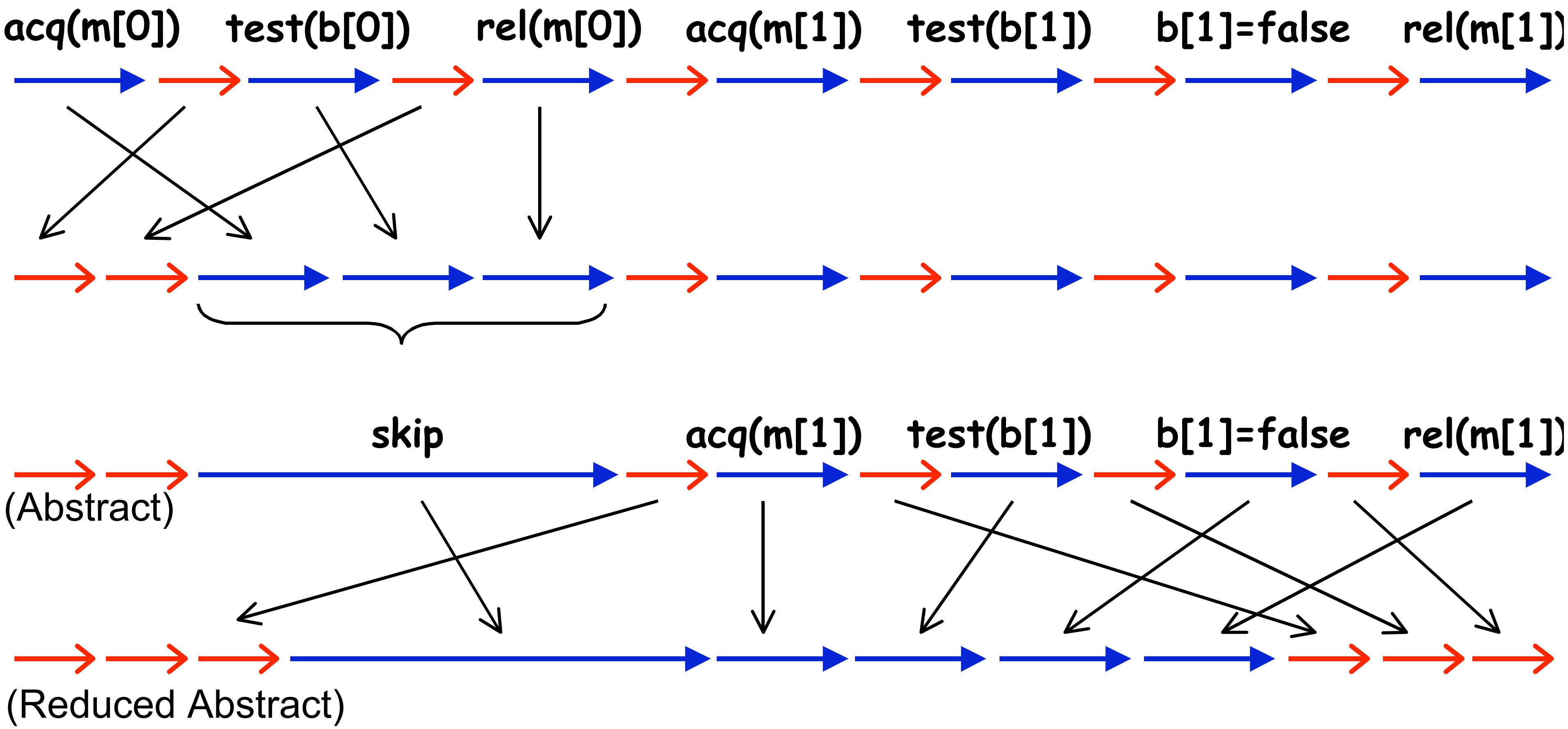
```
atomic void busy_acquire() {
  while (true) {
    pure { if (CAS(m,0,1)) break; }
  }
}
```



alloc

```
atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    pure {
      acquire(m[i]);
      if (b[i]) {
        b[i] = false;
        release(m[i]);
        return i;
      }
      release(m[i]);
    }
    i++;
  }
  return -1;
}
```

Abstract Execution of alloc



acq(m[0]) test(b[0]) rel(m[0]) acq(m[1]) test(b[1]) b[1]=false rel(m[1]

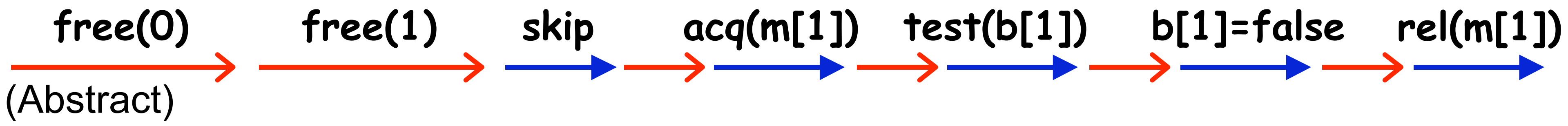
skip acq(m[1]) test(b[1]) b[1]=false rel(m[1]

(Abstract)

(Reduced Abstract)

Abstraction

- Abstract semantics admits more executions



free(0) free(1) skip acq(m[1]) test(b[1]) b[1]=false rel(m[1])

(Abstract)

- Can still reason about important properties
  - "`alloc` returns either the index of a freshly allocated block or -1"
  - cannot guarantee "`alloc` returns smallest possible index"
    - but what does this really mean anyway???

Type Checking

```
atomic void deposit(int n) {
  acquire(this);          R
  int j = bal;            B
  bal = j + n;            B
  release(this);          L
}
```

((R;B);B);L =

(R;B);L =

R;L =

A

```
atomic void depositLoop() {
  while (true) {
    deposit(10);          A
  }
}
```

(*A*)* = *C* ⇒ ERROR

alloc

```
boolean b[MAX];
Lock m[MAX];

atomic int alloc() {
   int i = 0;
   while (i < MAX) {
     acquire(m[i]);
     if (b[i]) {
       b[i] = false;
       release(m[i]);
       return i;
     }
     release(m[i]);
     i++;
   }
   return -1;
 }
```

A

A* = C

Type Checking with Purity

```
atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    pure {
      acquire(m[i]);
      if (b[i]) {
        b[i] = false;
        release(m[i]);
        return i;
      }
      release(m[i]);
    }
    i++;
  }
  return -1;
}
```

$A \uparrow A$

$B \uparrow A$

$(B \uparrow A)^* =$
$B^* \uparrow (B^*;A) =$
$B \uparrow A$

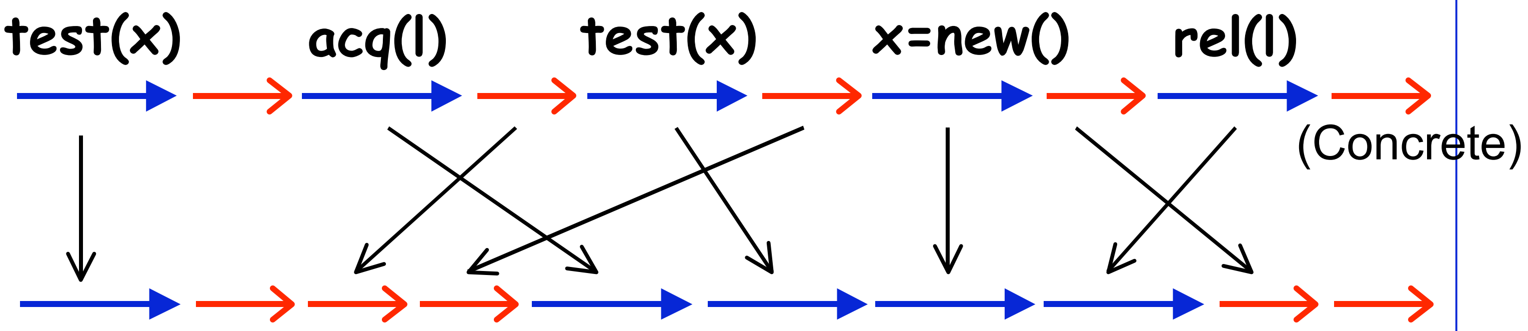
Double Checked Initialization

```
atomic void init() {
  if (x != null) return;
  acquire(l);
  if (x == null) x = new();
  release(l);
}
```

Double Checked Initialization

```
atomic void init() {
  if (x != null) return;
  acquire(l);
  if (x == null) x = new();
  release(l);
}
```

conflicting accesses

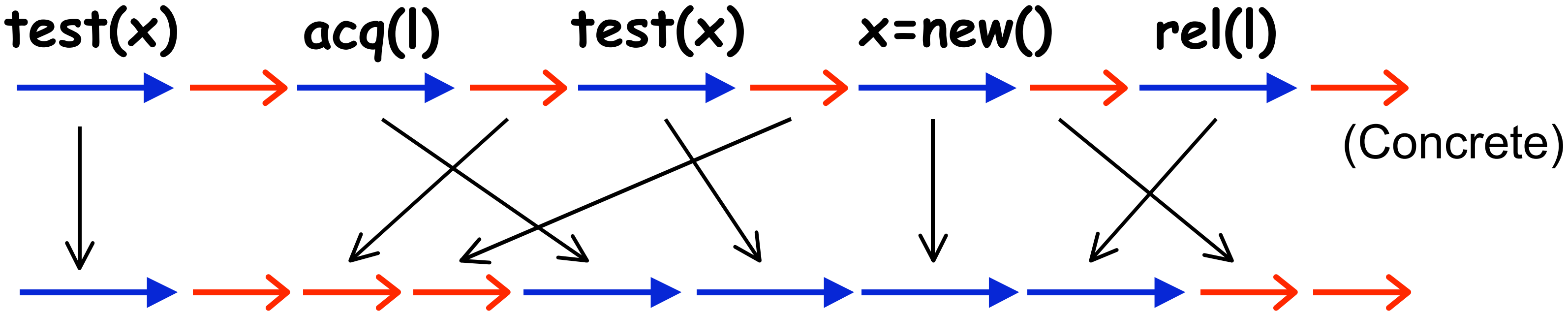


Double Checked Initialization

```
atomic void init() {
  if (x != null) return;
  acquire(l);
  if (x == null) x = new();
  release(l);
}
```

$A \uparrow A$

$A \uparrow \_$



Double Checked Initialization

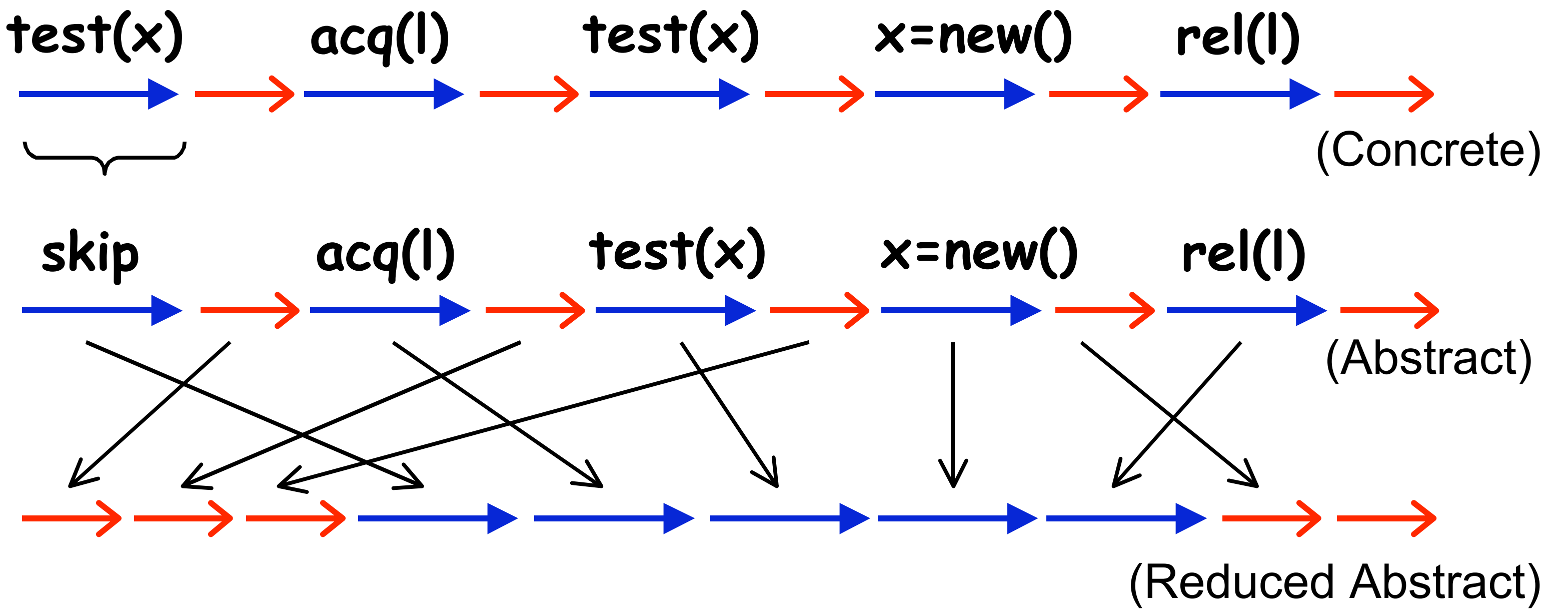
```
atomic void init() {
  pure { if (x != null) return; }
  acquire(l);
  if (x == null) x = new();
  release(l);
}
```

$B \uparrow A$

$A \uparrow \_$

$B;A \uparrow A$

$= A \uparrow A$



Modifying local variables in pure blocks

- Partition variables into global and local variables
- Allow modification of local variables

```
                   pure { acq(m); rel(m); }
                 ≅ skip
local x;           pure { acq(m); x = z; rel(m); }
global z;        ≅ pure { x = z; }
                 ≅ x = z0;
local x1, x2;      pure { acq(m); x1 = z; x2 = z; rel(m); }
global z;        ≅ pure { x1 = z; x2 = z; }
                 ≅ x1 = z0; x2 = z0
```

Transaction retry

```
atomic void apply_f() {
   int x, fx;
   while (true) {
      acq(m);
      x = z;
      rel(m);

      fx = f(x);

      acq(m);
      if (x == z) { z = fx; rel(m); break; }
      rel(m);
   }
}
```

$A \uparrow \_$

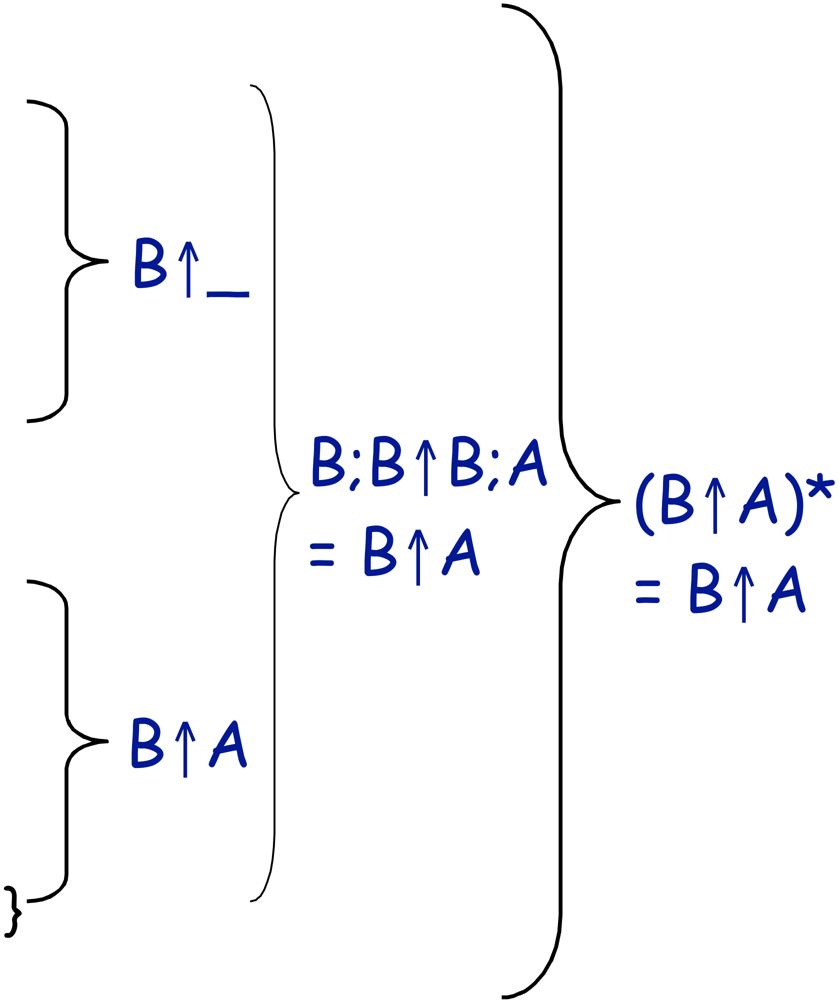
$A \uparrow A$

$A;A \uparrow A;A$

$= C \uparrow C$

Transaction retry

```
atomic void apply_f() {
    int x, fx;
    while (true) {
        pure {
            acq(m);
            x = z;
            rel(m);
        }

        fx = f(x);

        pure {
            acq(m);
            if (x == z) { z = fx; rel(m); break; }
            rel(m);
        }
```



$B \uparrow \_$

$B \uparrow A$

$B;B \uparrow B;A$
$= B \uparrow A$

$(B \uparrow A)^*$
$= B \uparrow A$

Transaction retry

```
atomic void apply_f() {
    int x, fx;
    while (true) {
        pure {
            acq(m);
            x = z;
            rel(m);
        }

        fx = f(x);

        pure {
            acq(m);
            if (x == z) { z = fx; rel(m); break; }
            rel(m);
        }
```

- The pure blocks allow us to prove apply_f abstractly atomic
- We can prove on the abstraction that z is updated to f(z) atomically

Transaction retry

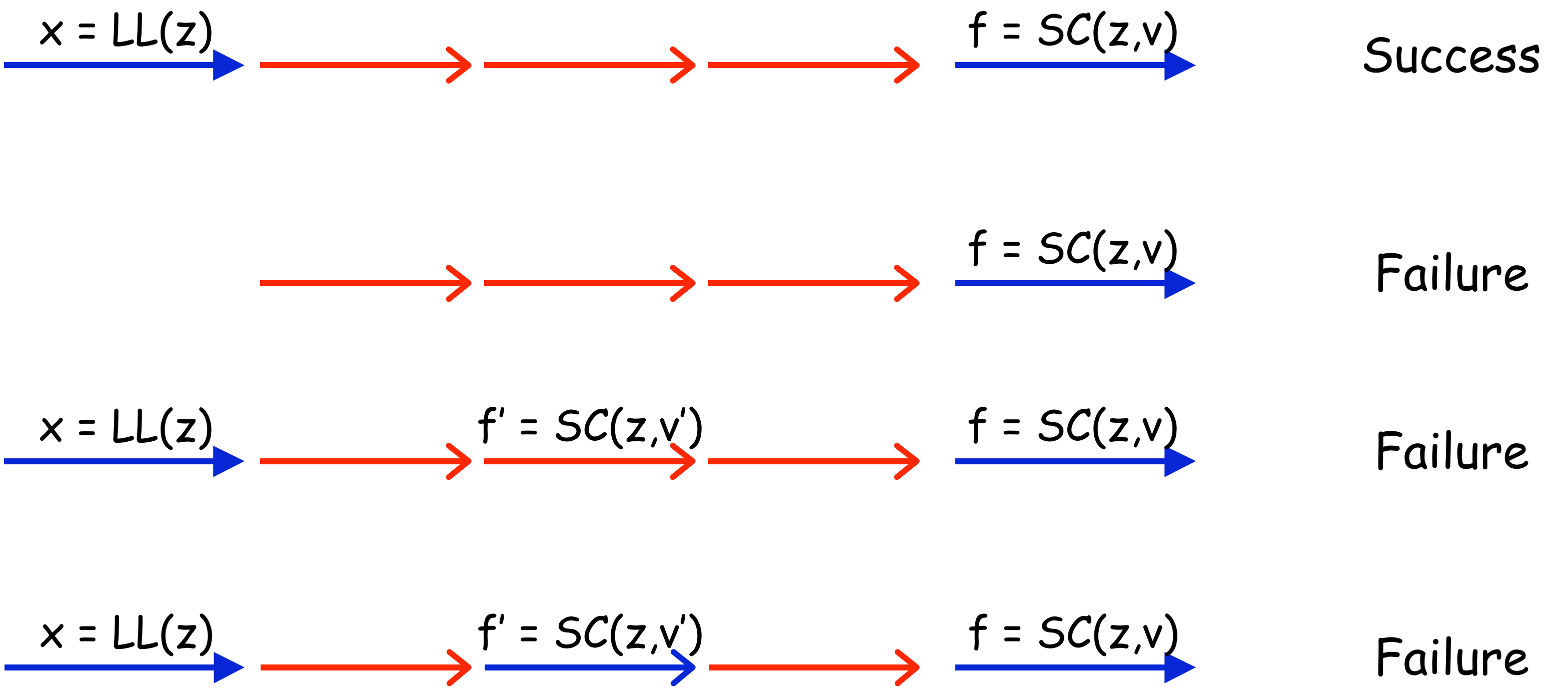
```
atomic void apply_f() {
    int x, fx;
    while (true) {
        pure {
            acq(m);
            x = z;
            rel(m);
        }

        fx = f(x);

        pure {
            acq(m);
            if (x == z) { z = fx; rel(m); break; }
            rel(m);
        }
    }
}
```

```
atomic void apply_f() {
    int x, fx;
    while (true) {


            skip;



        fx = f(x);

        if (*)
            // normal execution
            skip;
        else
            // exceptional execution
            acq(m); assume(x==z);
            z =fx; rel(m); break;
```

Lock-free synchronization

- Load-linked: x = LL(z)
  - loads the value of z into x
- Store-conditional: f = SC(z,v)
  - if no SC has happened since the last LL by this thread
    - store the value of v into z and set f to true
  - otherwise
    - set f to false

Scenarios



x = LL(z) f = SC(z,v) Success

f = SC(z,v) Failure

x = LL(z) f' = SC(z,v') f = SC(z,v) Failure

x = LL(z) f' = SC(z,v') f = SC(z,v) Failure

Lock-free atomic increment

```
atomic void increment() {
    int x;
    while (true) {
        x = LL(z);
        x = x + 1;
        if (SC(z,x)) break;
    }
}
```

Modeling LL-SC

- Global variable zSet initialized to { }
  - contains ids of threads who have performed the operation LL(z) since the last SC(z,v)

```
x = LL(z)     ≅  x = z; zSet = zSet ∪ { tid };

f = SC(z,v)   ≅  if (tid ∈ zSet)
                     { z = v; zSet = { }; f = true; }
                  else
                     { f = false; }
```

Modeling LL-SC

- Global variable zSet initialized to { }
  - contains the id of the unique thread that has performed the operation LL(z) since the last SC(z,v) and whose SC(z,v') is destined to succeed

```
x = LL(z)      ≅
                 if (*)
LL-Success(x,z)     { assume(zSet = {}); zSet = { tid }; x = z; }
                 else
                    { x = z; }
f = SC(z,v)    ≅
                 if (tid ∈ zSet)
                    { z = v; zSet = { }; f = true; }
                 else
                    { f = false; }
```

Modeling LL-SC

- Global variable zSet initialized to { }
  - contains the id of the unique thread that has performed the operation LL(z) since the last SC(z,v) and whose SC(z,v') is destined to succeed

```
x = LL(z)     ≅
        if (*)
            LL-Success(x,z);
        else
            x = z;
f = SC(z,v)  ≅
        if (tid ∈ zSet)
            { z = v; zSet = { }; f = true; }
        else
```

- LL-Success(x,z) is a right mover
- SC(z,v) is a left mover

… provided stores to z performed only through SC [Wang-Stoller 2005]

Lock-free atomic increment

```
atomic void increment() {
    int x;
    while (true) {
        x = LL(z);
        x = x + 1;
        if (SC(z,x)) break;
    }
}
```

```
atomic void increment() {
    int x;
    while (true) {
        if (*)
            LL-Success(x,z);
        else
            x = z;
        x = x + 1;
        if (SC(z,x)) break;
    }
}
```

Lock-free atomic increment

```
atomic void increment() {
    int x;
    while (true) {
        pure {
            if (*)
                LL-Success(x,z);
            else
                x = z;
            x = x + 1;
            if (SC(z,x)) break;
        }
    }
}
```

```
atomic void increment() {
    int x;
    while (true) {
        pure {
            if (*)
                LL-Success(x,z);
                x = x + 1;
                // SC succeeds
                SC(z,x); break;
            else
                x = z; x = x + 1;
                // SC fails
        }
    }
}
```

$B\uparrow(R;B;L)$
$=B\uparrow A$

$(B\uparrow A)^*$
$= B\uparrow A$

Atomicity and Purity Effect System

- Enforces properties for abstract semantics
  - pure blocks are reducible and side-effect free
  - atomic blocks are reducible
- Leverages other analyses
  - race-freedom
  - control-flow
  - side-effect

Summary

- Atomicity
  - enables sequential analysis
  - common in practice

- Purity enables reasoning about atomicity at an abstract level
  - matches programmer intuition
  - more effective checkers