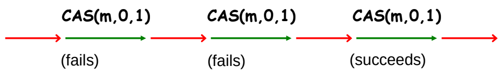
Part IV:
Exploiting Purity for Atomicity

Busy Acquire

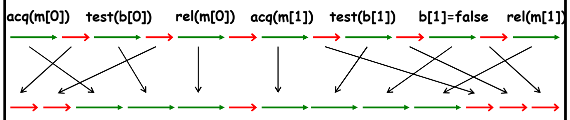
```
atomic void busy_acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```



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



Extending Atomicity

- Atomicity doesn't always hold for methods that are "intuitively atomic"
- Examples
  - initialization
  - resource allocation
  - wait/notify
  - caches
  - commit/retry transactions
- Want to extend reduction-based tools to check atomicity at an abstract level

Pure Code Blocks

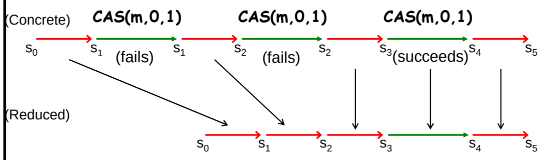
- Pure block: **`pure { E }`**
  - If **E** terminates normally, it does not update state visible outside of **E**
  - **E** is reducible

Busy Acquire

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

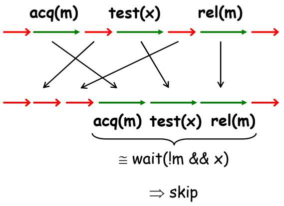
Abstract Execution of Busy Acquire

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



Purity and Abstraction

```
pure {
  acq(m);
  if (x)
    x = false;
  rel(m);
}
```

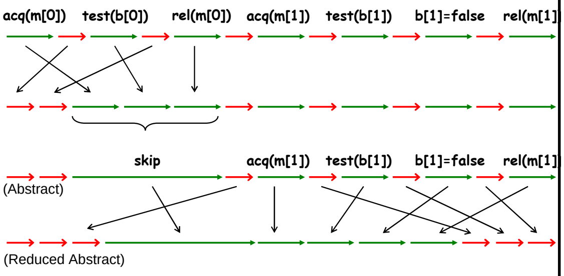


- Abstract execution semantics:
  - **pure** blocks can be skipped

alloc

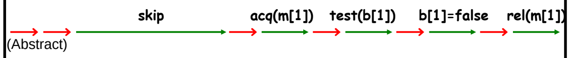
```
atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    pure {
      acquire(m[i]);
      if (b[i]) {
        b[i] = false;
        release(m[i]);
        return i;
      }
      release(m[i]);
    }
    i++;
  }
  return -1;
}
```

Abstract Execution of alloc



Abstraction

- Abstract semantics admits more executions



- Can still reason about most important properties
  - "`alloc` returns either the index of a freshly allocated block or -1"

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↑A

B↑A

(B↑A)* =
B*↑(B*;A) =
B↑A

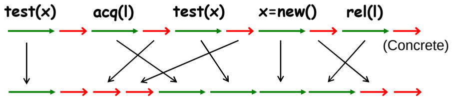
Double Checked Initialization

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

Double Checked Initialization

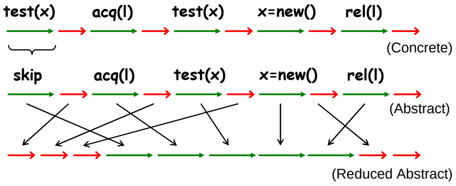
```
atomic void init() {
  if (x != null) return;
  acquire(l);
  if (x == null) x = new();
  release(l);
}
```

conflicting accesses



Double Checked Initialization

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



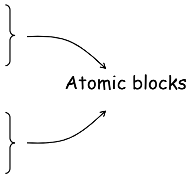
Modifying local variables in pure blocks

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

```
local x;                 pure { acq(m); x = z; rel(m); }
global z;              ≅ pure { wait(m == 0) → x = z; }
                       ≅ x = z0;


local x1, x2;            pure { acq(m); x1 = z; x2 = z; rel(m); }
global z;              ≅ pure { wait(m == 0) → 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);
  }
}
```



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);
    }
  }
}
```

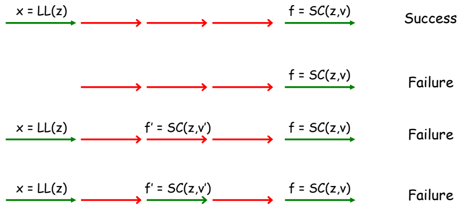
- The pure blocks allow us to prove apply_f abstractly atomic
- We can prove on the abstraction that z is updated to f(z) atomically

---

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



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

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

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

Intuition

- A successful SC operations is a left mover
- The LL operation corresponding to a successful SC operation is a right mover

---

Modeling LL-SC

- Global variable zSet
  - 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
  - initialized to { }

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

Modeling LL-SC

- Global variable zSet
  - 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
  - initialized to { }

```
x = LL(z)    ≅
     if (*)
        LL-Success(z);
     else
        LL-Failure(z);
f = SC(z,v)  ≅
     if (*)
        SC-Success(z,v);
     else
        SC-Failure(z,v);
```

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

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 (*)
         x = LL-Success(z);
      else
         x = z;
      x = x + 1;
      if (SC-Success(z,x))
         break;
   }
}
```

Lock-free atomic increment

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

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
- Additional notions of abstraction
  - unstable reads/writes, weak purity, ...

Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - other applications: model checking [Stoller-Cohen 03, Flanagan-Qadeer 03], procedure summaries [Qadeer et al 04]
- Other reduction-based atomicity checkers
  - Bogor model checker [Hatcliff et al 03]
- Beyond reduction
  - dynamic checking [Wang-Stoller 03]
  - model-checking atomicity requirements [Flanagan 04]
  - view consistency [Artho et al. 03]

Summary

- Atomicity
  - enables sequential analysis
  - common in practice
- Purity enables reasoning about atomicity at an abstract level
  - matches programmer intuition
  - more effective checkers