Exploiting purity for atomicity
Flanagan, Freund and Qadeer

Informal meeting over an article

Inria Sophia Antipolis

November 16, 2006
Various techniques exist to reason about multi-threaded programs:

- Owicki/Gries

*Atomicity* is build on top of reduction.
Introduction

Various techniques exist to reason about multi-threaded programs:
- Owicki/Gries
- Reduction [Lipton]

"Atomicity" is build on top of reduction.
Introduction

Various techniques exist to reason about multi-threaded programs:
- Owicki/Gries
- Reduction [Lipton]
- Rely/Guarantee [Jones]

\textit{Atomicity} is build on top of reduction.
Various techniques exist to reason about multi-threaded programs:

- Owicki/Gries
- Reduction [Lipton]
- Rely/Guarantee [Jones]
- Separation Logic [Reynold, East London Massive]

*Atomicity* is build on top of reduction.
Reduction

Reducibility

- Every execution (possibly interleaved) is equivalent to a serial one.
Every execution (possibly interleaved) is equivalent to a serial one.
Reduction

Reducibility

- Every execution (possibly interleaved) is equivalent to a serial one.
Syntactically, if it follows the pattern $R^*N^?L^*$:

- $R^*$: zero or more right movers (right commutes).
- $N^?$: zero or one non mover (does not commute).
- $L^*$: zero or more left mover (left commutes).
Reduction

Syntactically, if it follows the pattern $R^* N^? L^*$:

- $R^*$: zero or more right movers (right commutes).
- $N^?$: zero or one non mover (does not commute).
- $L^*$: zero or more left mover (left commutes).

Additionally:

- $B$: a both mover (right and left commutes).
Reduction

If all accesses to $x$ and $y$ are protected by $m$:

```c
void do1() {
    m.acquire();
    x = y;
    y = 2;
    m.release();
}
```

```c
void do2() {
    m.acquire();
    while(x != 0)
        m.wait();
    x = y;
    y = 2;
    m.release();
}
```

- Left one: reducible!
- Right one: not reducible
Reduction

This is very useful for verification purpose. Given, a method $m$, if:

- Every execution $e$ of $m$ reduces to a sequential one $e'$. 
Reduction

This is very useful for verification purpose. Given, a method $m$, if:

- Every execution $e$ of $m$ reduces to a sequential one $e'$.
- $e'$ respects $m$’s specification: $\vdash (e', S)$, then:
This is very useful for verification purpose. Given, a method \( m \), if:

- Every execution \( e \) of \( m \) reduces to a sequential one \( e' \).
- \( e' \) respects \( m \)'s specification: \( \models (e', S) \), then:
- \( \models (e, S) \)!
Reduction

This is very useful for verification purpose. Given, a method $m$, if:

- Every execution $e$ of $m$ reduces to a sequential one $e'$.
- $e'$ respects $m$'s specification: $\vdash (e', S)$, then:
- $\models (e, S)$!
Reduction

This is very useful for verification purpose. Given, a method $m$, if:

- Every execution $e$ of $m$ reduces to a sequential one $e'$.
- $e'$ respects $m$’s specification: $\vdash (e', S)$, then:
- $\models (e, S)$ !

**Independent method**

- Consists only of both movers.
Reduction

In practice, some methods are reducible, but there are many restrictions. Given three regions $e_1, e_2$ and $e_3$:

- $\text{independent}(e_1) \land \text{reducible}(e_2; e_3) \rightarrow \text{reducible}(e_1; e_2; e_3)$
In practice, some methods are reducible, but there are many restrictions. Given three regions \( e_1, e_2 \) and \( e_3 \):

- \( \text{independent}(e_1) \land \text{reducible}(e_2; e_3) \longrightarrow \text{reducible}(e_1; e_2; e_3) \)

For method calls:

- \( \text{independent}(e_1; e_3) \land \text{independent}(e_2) \longrightarrow \text{reducible}(e_1; e_2; e_3) \)
Reduction

In practice, some methods are reducible, but there are many restrictions. Given three regions $e_1$, $e_2$ and $e_3$:

- $\text{independent}(e_1) \land \text{reducible}(e_2; e_3) \rightarrow \text{reducible}(e_1; e_2; e_3)$

For method calls:

- $\text{independent}(e_1; e_3) \land \text{independent}(e_2) \rightarrow \text{reducible}(e_1; e_2; e_3)$
- But, $\text{reducible}(e_1; e_3) \land \text{independent}(e_2) \nrightarrow \text{reducible}(e_1; e_2; e_3)$
Reduction

In practice, some methods are reducible, but there are many restrictions. Given three regions \( e_1, e_2 \) and \( e_3 \):

- \( \text{independent}(e_1) \land \text{reducible}(e_2; e_3) \rightarrow \text{reducible}(e_1; e_2; e_3) \)

For method calls:

- \( \text{independent}(e_1; e_3) \land \text{independent}(e_2) \rightarrow \text{reducible}(e_1; e_2; e_3) \)
- But, \( \text{reducible}(e_1; e_3) \land \text{independent}(e_2) \notightarrow \text{reducible}(e_1; e_2; e_3) \)
  - This case happens often in practice.
In practice, some methods are reducible, but there are many restrictions. Given three regions $e_1$, $e_2$ and $e_3$:

- $\text{independent}(e_1) \land \text{reducible}(e_2; e_3) \rightarrow \text{reducible}(e_1; e_2; e_3)$

For method calls:

- $\text{independent}(e_1; e_3) \land \text{independent}(e_2) \rightarrow \text{reducible}(e_1; e_2; e_3)$
- But, $\text{reducible}(e_1; e_3) \land \text{independent}(e_2) \not\rightarrow \text{reducible}(e_1; e_2; e_3)$
  - This case happens often in practice.
  - Need for a relaxed definition.
Atomicity

Purity

Atomicity

- Reduction where *pure* steps are removed from the trace.
  - A *pure* step is a step that does not write something when it terminates normally.
Atomicity

Purity

Atomicity

- Reduction where *pure* steps are removed from the trace.
  - A *pure* step is a step that does not write something when it terminates normally.

\[
i_1 \rightarrow E \rightarrow i_2 \rightarrow E \rightarrow i_3
\]
Atomicity

Reduction where *pure* steps are removed from the trace.

- A *pure* step is a step that does not write something when it terminates normally.

```
\[ i_1 \xrightarrow{E} \]  
\[ i_2 \xrightarrow{E} \]  
\[ i_3 \xrightarrow{E} \]
```

```
\[ i_1 \xrightarrow{E} \]  
\[ \text{skip} \]  
\[ i_3 \xrightarrow{E} \]
```

pure removed
Atomicity

Purity

Atomicity

- Reduction where pure steps are removed from the trace.
  - A pure step is a step that does not write something when it terminates normally.

```
  i_1  E  i_2  E  i_3  E
    ↓        ↓        ↓
  i_1  E  skip  i_3  E
```

pure removed

```
  E  i_1  E  i_3  E
```

reduced
Atomicity

Purity

- Lines 3 and 4 are pure.
- Removing them from the trace reduces to traces of do1().
- Thus, it is atomic!

```c
void do2() {
    m.acquire();
    while(x != 0) {
        m.wait();
        x = y;
        y = 2;
    }
    m.release();
}
```
An *unstable* variable is a variable which does not affect program correctness.

```c
int packetCount;
Queue packets;

// atomic
void enqueue(Packet p);

void receive(Packet p){
    packetCount++;
    enqueue(p);
}
```
 Atomicity

Instability

- An unstable variable is a variable which does not affect program correctness.
- Read and write to unstable variables are considered as both movers ($B$).

```c
int packetCount;
Queue packets;

// atomic
void enqueue(Packet p);

void receive(Packet p){
    packetCount++;
    enqueue(p);
}
```
Atomicity

Instability

- An *unstable* variable is a variable which does not affect program correctness.
- Read and write to unstable variables are considered as both movers ($B$).
- Thus, the method on the right is atomic.

```c
int packetCount;
Queue packets;

// atomic
void enqueue(Packet p);

void receive(Packet p) {
    packetCount++;
    enqueue(p);
}
```
Atomicity
Weak purity and abstract semantic

*Weak purity*:  
- Same as purity but writes to local variables are allowed.
Atomicity
Weak purity and abstract semantic

*Weak purity:*
- Same as purity but writes to local variables are allowed.

*Abstract semantic:*
- All along the papers, it is mentioned: atomicity ⇔ “good behavior”.
Atomicity
Weak purity and abstract semantic

Weak purity:
- Same as purity but writes to local variables are allowed.

Abstract semantic:
- All along the papers, it is mentioned: atomicity ≜ “good behavior”.
  - This “good behavior” isn’t further explained.
Atomicity
Weak purity and abstract semantic

*Weak purity:*
- Same as purity but writes to local variables are allowed.

*Abstract semantic:*
- All along the papers, it is mentioned: atomicity $\Leftrightarrow$ “good behavior”.
  - This “good behavior” isn’t further explained.
- They also speak of “abstract semantic”.
Atomicity
Weak purity and abstract semantic

**Weak purity:**
- Same as purity but writes to local variables are allowed.

**Abstract semantic:**
- All along the papers, it is mentioned: atomicity $\iff$ “good behavior”.
  - This “good behavior” isn’t further explained.
- They also speak of “abstract semantic”.
  - This semantic isn’t formalized.
Atomicity
Abstract semantic

Authors claim:

- This method is atomic.
- However, `compute` can be called multiple times for the same key!

```java
void cachePut(String k, Object val);
Object cacheGet(String k);

Object compute(String k);

Object lookup(String k) {
    //@ pure{
    Object r = cacheGet(k);
    if (r != null)
        return r;
    //@}

    Object r = compute(k);
    cachePut(k, r);
    return r;
}
```
Atomicity
Abstract semantic

Authors claim:
- Independently from other threads, this method always returns true or false.
- Thus, it is atomic under a weaker semantic.

```c
int alloc()
{
    int i = 0;
    while (i < max)
    {
        acquire(l[i]);

        if (free[i]) {
            free[i] = false;
            release(l[i]);
            return true;
        }

        release(l[i]);
        i++;
    }

    return false;
}
```
This paper contains great ideas. However, in my opinion, it isn’t precise enough.
Conclusion

- This paper contains great ideas.
- However, in my opinion, it isn’t precise enough.
- Weaknesses can be reduced:
  - The abstract semantic can be formally captured by a specification.
  - With this idea, the concept of atomicity is more precise and more general.
In practice,

- **R**: acquiring a lock.
- **N**: a write to shared memory.
- **L**: releasing a lock.
- **B**: read/write to a locked variable or a local one.
In practice,

- **R**: acquiring a lock.
- **N**: a write to shared memory.
- **L**: releasing a lock.
- **B**: read/write to a locked variable or a local one.

In the effect system:

\[
\begin{array}{c}
\vdash e : a,b \quad \text{pure } e \\
\hline
\vdash e : B, b \\
\end{array}
\]