A Foundation for Verifying Concurrent Programs

K. Rustan M. Leino
RiSE, Microsoft Research, Redmond

joint work with Peter Müller and Jan Smans

Lecture 0
1 September 2009
FOSAD 2009, Bertinoro, Italy
Program verification

Prove program correctness for all possible inputs and behaviors
Modular verification

- Prove parts of a program separately
- Correctness of every part implies correctness of whole program
Specifications

- Record programmer design decisions
- Describe usage of program constructs
- Provide redundancy
- Enable modular verification
Specification style

Specification and verification methodology
Describes properties of the heap
Active area of research

Ownership
- Spec#, Java+JML, vcc, type systems, …

Dynamic frames
- VeriCool, Dafny

Permissions (capabilities)
- Effect systems, separation logic, VeriCool 3, Chalice, …
Concurrent programs

- Interleaving of thread executions
- Unbounded number of: threads, locks, ...
- We need some basis for doing the reasoning
- A way of thinking!
Concurrent programs
  - Features like: threads, monitors, abstraction as well as: objects, methods, loops, ...
  - Avoid errors like: race conditions, deadlocks
Specifications with permissions
Building a program verifier
Square

Pre- and postconditions
Cube

Loop invariants
Specifications at run time

- Helps testing find bugs more quickly
- Optional, they can be treated as ghosts
- If they are to be ghosted, specifications must have no side effects (on non-ghost state)
Dealing with memory (the heap)

- Access to a memory location requires permission
- Permissions are held by activation records
- Syntax for talking about permission to y: $\text{acc}(y)$
method Main()
{
  var c := new Counter;
  call c.Inc();
}

method Inc()
  requires acc(y)
  ensures acc(y)
{
  y := y + 1;
}
A specification expression can mention a memory location only if it also entails the permission to that location.

- `acc(y) && y < 100` - ✔️
- `y < 100` - ❌
- `acc(x) && y < 100` - ❌
- `acc(o.y) && p.y < 100` - ❌
- `o == p && acc(o.y) && p.y < 100` - ✔️
- `x / y < 20` - ❌
- `y ≠ 0 && x / y < 20` - ✔️
A loop iteration is like its own activation record

```java
Before;
while (B) invariant J { S; }
After;
```

is like

```java
Before;
call MyLoop(...);
After;
```

```java
method MyLoop(...) 
  requires J 
  ensures J 
  {
    if (B) {
      S;
      call MyLoop(...);
    }
  }
```
method M()
    requires acc(x) && acc(y) && x <= 100 && y <= 100
{
    while (y < 100)
        invariant acc(y) && y <= 100
        {
            y := y + 1;
            x := x + 1;  // error: no permission to access x
        }
    assert x <= y;
}
method M()
    requires acc(x) && acc(y) && x <= 100 && y <= 100
{
    while (y < 100)
        invariant acc(y) && y <= 100
        { 
            y := y + 1;
        }
    assert x <= y;
}
ISqrt with fields

Loop invariants with permissions
Threads run concurrently

A new thread of control is started with the `fork` statement

A thread can wait for another to complete with the `join` statement

Permissions are transferred to and from a thread via the starting method’s pre- and postconditions
ForkInc

Fork and join
The two halves of a call

call == fork + join

\[
\text{call } x,y := o.M(E, F); \\
\text{fork } tk := o.M(E, F); \\
\text{join } x,y := tk;
\]

... but is implemented more efficiently
TwoSqrts

Parallel computation
Recall:
A specification expression can mention a memory location only if it also entails some permission to that location.

Example: \texttt{acc}(y) \&\& y < 100

Without any permission to \texttt{y}, other threads may change \texttt{y}, and then \texttt{y} would not be stable.
Read permissions

- $\text{acc}(y)$ write permission to $y$
- $\text{rd}(y)$ read permission to $y$

At any one time, at most one thread can have write permission to a location
VideoRental

Parallel reads
Fractional permissions

- $\text{acc}(y)$: 100% permission to $y$
- $\text{acc}(y, p)$: $p\%$ permission to $y$
- $\text{rd}(y)$: read permission to $y$
- Write access requires 100%
- Read access requires $>0\%$

\[
\text{acc}(y) = \text{acc}(y, 69) + \text{acc}(y, 31)
\]

\[
\text{rd}(y) \approx \text{acc}(y, \varepsilon)
\]
Implicit dynamic frames

method M() requires acc(y) ensures acc(y) can change y

Can
method P() requires rd(y) ensures rd(y) change y?

That is, can we prove:

```java
method Q()
    requires rd(y) && y == 5
{
    call P();
    assert y == 5;
}
```

Demo: NoPerm
What if two threads want write access to the same location?

class Fib {
    var y: int;
    method Main() {
        var c := new Fib;
        fork c.A();
        fork c.B();
    }
}

method A() {
    y := y + 21;
}

method B() {
    y := y + 34;
}
class Fib {
    var y: int;
    invariant acc(y);
    method Main()
    {
        var c := new acc(c.y)
        share c;
        fork c.A();
        fork c.B();
    }
}

method A() ...
{
    acquire this;
    y := y + 21;
    release this;
}

method B() ...
{
    acquire this;
    y := y + 34;
    release this;
}
Monitor invariants

- Like other specifications, can hold both permissions and conditions
- Example: invariant acc(y) && 0 <= y
Object life cycle

- new
- thread_local
- share
- shared, available
- release
- acquire
- shared, locked
The concepts holding a lock, and having permissions are orthogonal to one another. In particular:

- Holding a lock does not imply any right to read or modify shared variables

Their connection is:

- Acquiring a lock obtains some permissions
- Releasing a lock gives up some permissions
Thread-safe libraries

- Server-side locking
  - "safer" (requires less thinking)

```
invariant acc(y);
method M()
  requires true
  {
    acquire this;  y := …;  release this;
  }
```

- Client-side locking
  - more efficient

```
method M()
  requires acc(y)
  {
    y := …;
  }
```