Stefan Heule
ETH Zurich

Joint work with: Rustan Leino, Peter Müller, Alexander Summers
Microsoft Research ETH Zurich ETH Zurich

Overview

- Verification of (race-free) concurrent programs using fractional permissions

- Background
- Identify the problem
- Abstract read permissions
- Handling calls, fork/join
- Permission expressions
- Conclusions

Fractional Permissions Boyland, SAS'03

- Provide a way of describing disciplined (race-free) use of shared memory locations
- Many readers ✓ one writer ✓ never both
- Heap locations are managed using *permissions*
- Permission amounts are *fractions* p from [0,1]
  - p=0 (no permission)
  - 0<p<1 (read permission)
  - p=1 (read/write permission)
- Permissions are passed between methods/threads
  - can be split and recombined, never duplicated

Chalice Müller and Leino, ESOP'09

- Verification language/tool for concurrent software
  - race-freedom, deadlock-freedom, functional specs
- Extend first-order logic assertions to additionally include "accessibility predicates", i.e., *permissions*:
  - acc(o.f, p) – we have permission p to location o.f

- Permissions in this talk (not Chalice):
  - acc(o.f,1)
  - acc(o.g,$^1/_2$)

Inhale and Exhale

- Permission transfer (between threads/calls) is modelled using two operations
- exhale p means to
  - *assert* all pure assertions in p (e.g. heap properties)
  - check and give up permissions mentioned in p
- inhale p means to
  - *assume* all pure assertions in p
  - gain permissions mentioned in p
  - remove previous knowledge about newly-accessible locations ("havoc" locations)

Inhale and Exhale - Example

```
method m()
  requires p;
  ensures q;
{
  // inhale our precondition: p
  ...
  // exhale precondition of called method: p
  call m();
  // inhale postcondition of called method: q
  ...
  // exhale our postcondition: q
}
```

Difficulties with Fractional Permissions

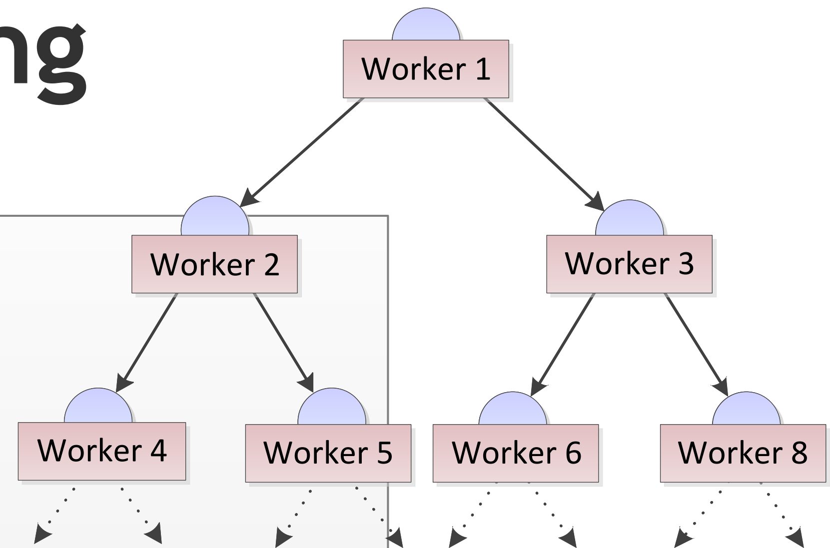
- Permission amounts (*fractions*) always need to be specified explicitly
  - Manual book-keeping is tedious
  - Creates “noise” in specifications and limits reuse
  - Programmers only think in terms of read or write permissions
  - The actual amounts (i.e., fractions) do not matter to the programmer

Goal

- Abstract permission model
    - User doesn’t choose concrete fractions for read permissions
    - Differentiate between *read*, *read/write* and *no* permission

- Also: Unbounded splitting of read permissions

Unbounded Splitting




```
class Node {
  Node l, r

  Outcome method work(Data data)
    requires «permission to data.f»
    ensures «permission to data.f»
  {
    Outcome out := new Outcome()

    if (l != null) left  := fork l.work(data)
    if (r != null) right := fork r.work(data)
    /* perform work on this node, using data.f */
    if (l != null) out.combine(join left)
    if (r != null) out.combine(join right)
    return out
  }
}
```

Abstract Read Permissions

- Introduce *abstract* read permissions: acc(o.f,rd)
  - corresponds to a *fixed, positive,* and *unknown* fraction
  - positive amount: allows reading the location o.f
- Specifications are written using
  - acc(o.f,1) to represent the full permission (read/write)
  - acc(o.f,rd) for read permissions
- In general, different read permissions can correspond to different fractions

Method Calls

- Permission is often required *and* returned later

```
method m(Cell c)
  requires acc(c.f,rd)
  ensures acc(c.f,rd)
{
  /* ... */
}
```

```
method main(Cell c)
  requires acc(c.f,1)
{
  c.f := 0
  call m(c)
  c.f := 1
}
```

- Rule: All read permissions acc(o.f,rd) in pre- and postconditions correspond to the *same* amount

Method Calls (2)

```
method m(Cell c)
  requires acc(c.f,rd)
  ensures acc(c.f,rd)
{



  call m(c)






}
```

Method declaration: Use π_m to interpret any read permission: $0 < \pi_m < 1$
`∀o,f. Mask[o.f] == 0`

Inhale precondition: `Mask[c.f] +=` π_m

Declare $0 < \pi_{call} < 1$ (rd in recursive call)
Exhale precondition for recursive call

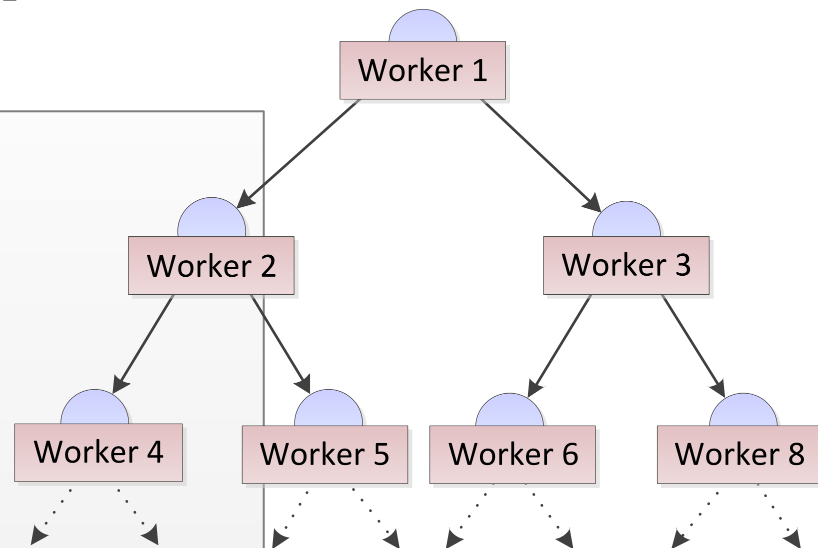
- Check that we have *some* permission
  `assert Mask[c.f] > 0`
- Constrain π_{call} to be smaller than what we have
  `assume` π_{call} `< Mask[c.f]`
- Give away this amount: `Mask[c.f] -=` π_{call}

Inhale postcondition: `Mask[c.f] +=` π_{call}

Exhale postcondition

- Check available permission
  `assert Mask[c.f] >=` π_m
- Remove permission from mask
  `Mask[c.f] -=` π_m

Introductory example revisited




```
class Node {
  Node l,r

  Outcome method work(Data data)
    requires «permission to data.f»
    ensures «permission to data.f»
  {
    Outcome out := new Outcome()

    if (l != null) left  := fork l.work(data)
    if (r != null) right := fork r.work(data)
    /* perform work on this node, using data.f */
    if (l != null) out.combine(join left)
    if (r != null) out.combine(join right)
    return out
  }
}
```

Introductory example revisited

```
class Node {
  Node l,r

  Outcome method work(Data data)
    requires acc(data.f,rd)
    ensures acc(data.f,rd)
  {
    Outcome out := new Outcome()

    if (l != null) left  := fork l.work(data)
    if (r != null) right := fork r.work(data)
    /* perform work on this node, using data.f */
    if (l != null) out.combine(join left)
    if (r != null) out.combine(join right)
    return out
  }
}
```

- rd-permission sufficient for this example

Some (unknown) amount(s) are given away

And retrieved again later on

More complex example

```
class Client {
  int method main(Problem p, Solver s)
  {
    tk1 := call s.handle(p)
    tk2 := call s.handle(p)

    r1 := join tk1
    r2 := join tk2
    r := max(r1,r2)
  }
}
```

Intuitively, `handle` returns the permission it was passed *minus* the permission held by the forked thread

How can we express that the we get back all the permission given away?

```
class Solver {
  int method solve(Problem p, Data d)
  { /* ... */ }

  token<Solver.solve> method handle(Problem p)
  { /* initialize, etc. */ tk := fork solve(p, d)
    return tk; }
}
class Problem { int f; }
```

`solve` requires read access to the problem

`handle` requires read access to the problem and gives some of it to a newly-forked thread

Permission Expressions

- We need a way to express the (unknown) fractions held by a forked thread
- We also need to express *differences*

- We generalize our permissions to acc(o.f,P) where P is a permission expression
  - 1 and rd as before (full and read permission)
  - rd(tk) where tk is a token for a forked thread
  - P_1+P_2 and P_1-P_2

```
class Client {
  int method main(Problem p, Solver s)
    requires acc(p.f,1)
    ensures acc(p.f,1)
  {
    tk1 := call s.handle(p)   // 1
    tk2 := call s.handle(p)   // 1 - rd(tk1)
                              // 1 - rd(tk1) - rd(tk2)

    r1 := join tk1
    r2 := join tk2            // 1 - rd(tk2)
    r := max(r1,r2)           // 1
  }
}
```

```
class Solver {
  int method solve(Problem p, Data d)
    requires acc(p.f,rd); ensures acc(p.f,rd)
  { /* ... */ }

  token<Solver.solve> method handle(Problem p)
    requires acc(p.f, rd)
    ensures acc(p.f, rd-rd(result))
  { /* initialize, etc. */ tk := fork solve(p, d)
    return tk; }
}
class Problem { int f; }
```

Conclusions

- Presented a specification methodology
  - similar expressiveness as fractional permissions
  - avoids concrete values for read permissions
  - allows the user to reason about read/write abstractly
- Support for full Chalice language
  - fork/join, monitors, channels, loop invariants
- Methodology is implemented
  - backwards-compatible with few simple edits
  - performance comparable with previous encoding

End.

Are there any questions?