Abstract Read Permissions

Fractional Permissions without the Fractions

Alex Summers
ETH Zurich

Joint work with: Stefan Heule, Rustan Leino, Peter Müller
ETH Zurich, Microsoft Research, 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

Notation

- Examples shown using *Implicit Dynamic Frames* assertions [Smans'09].
- Permissions represented in assertions by "accessibility predicates": acc(x.f, p)
  - means we have permission p to location x.f
- Permissions treated multiplicatively; i.e.,
  - acc(x.f, p) && acc(x.f, p) ≡ acc(x.f, 2p)
- Related to Sep. Logic [Parkinson/Summers'12]
  - Roughly: read acc(x.f,p) as $x.f \overset{p}{\mapsto} \_$
- This work applies to any such program logic
- We use *Chalice* language syntax [Leino/Müller]

Inhale and Exhale

- "inhale P" and "exhale P" are used to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{


}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{

   ...

   call m()

   ...

}
```

Inhale and Exhale

- “inhale P” and “exhale P” are used to encode transfers between threads/calls
- “inhale P” means:
  - *assume* heap properties in p
  - gain permissions in p
- “exhale P” means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{
  // inhale P
  ...

  call m()


  ...

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{
  // inhale P
  ...
  // exhale P
  call m()

  ...

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{
  // inhale P
  ...
  // exhale P
  call m()
  // inhale Q
  ...

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{
  // inhale P
  ...
  // exhale P
  call m()
  // inhale Q
  ...
  // exhale Q
}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions
  - *havoc* heap locations to which no permission is now held

```
void m()
requires P
ensures Q
{
  // inhale P
  ...
  // exhale P
  call m()
  // inhale Q
  ...
  // exhale Q
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee

```
method evaluate(Cell c)
  requires acc(c.f, ?)
  ensures acc(c.f, ?)
{
  /* ... calculations ... */
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee

```
method evaluate(Cell c)
  requires acc(c.f, 2/3)
  ensures acc(c.f, 2/3)
{
  /* ... calculations ... */
}
```

```
method main(Cell c)
  requires acc(c.f, 1/2)
{

  call evaluate(c)  ✗

}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen

```
method equals(Cell c)
  requires acc(this.f, ?) && acc(c.f, ?)
  ensures acc(this.f, ?) && acc(c.f, ?)
{
  /* ... comparisons ... */
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen

```
method equals(Cell c)
  requires acc(this.f, 2/3) && acc(c.f, 2/3)
  ensures acc(this.f, 2/3) && acc(c.f, 2/3)
{
  /* ... comparisons ... */
}
```

What if
this = c ?

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen

```
method equals(Cell c)
  requires acc(this.f, 1/3) && acc(c.f, 1/3) &&
 (this != c ==> acc(this.f, 1/3) && acc(c.f, 1/3))
  ensures acc(this.f, 1/3) && acc(c.f, 1/3) &&
 (this != c ==> acc(this.f, 1/3) && acc(c.f, 1/3))
{
  /* ... comparisons ... */
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen
- Recursive methods require parameterisation

```
method m(Cell c)
  requires acc(c.f, ?)
  ensures acc(c.f, ?)
{
  // do stuff
  call m(c)
  // do more stuff
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen
- Recursive methods require parameterisation

```
method m(Cell c, Perm p)
  requires acc(c.f, ?)
  ensures acc(c.f, ?)
{
  // do stuff
  call m(c)
  // do more stuff
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen
- Recursive methods require parameterisation

```
method m(Cell c, Perm p)
  requires acc(c.f, p)
  ensures acc(c.f, p)
{
  // do stuff
  call m(c)
  // do more stuff
}
```

Difficulties with Fractional Permissions

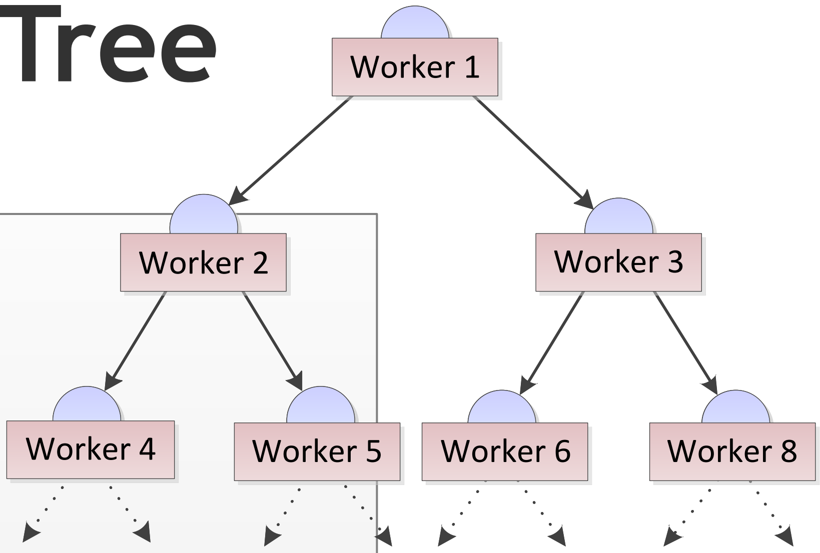
- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen
- Recursive methods require parameterisation

```
method m(Cell c, Perm p)
  requires acc(c.f, p)
  ensures acc(c.f, p)
{
  // do stuff
  call m(c, p/2)
  // do more stuff
}
```

Difficulties with Fractional Permissions

- Concrete fractions cause tension: caller vs callee
  - Reuse can be made difficult
  - Framing may be compromised
- Aliasing information is relevant to values chosen
- Recursive methods require parameterisation
- Manual book-keeping is tedious
  - Creates "noise" in specifications and new mistakes
  - Programmers ideally only need care about:
    - when does a thread have full (write) permission?
    - when does a thread have some (read) permission?
    - … and differences in amounts of permission (…later)

Example: Workers Tree




```
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
  }
}
```

How much permission?

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

Matching rd permissions

- Permission is often required *and* returned later

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

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

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

Encoding Method Calls

We use `Mask[o.f]` to denote the permission amount held to `o.f`

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

  // do stuff

  call m(c)


  // do more stuff


}
```

Encoding Method Calls

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

  // do stuff

  call m(c)

  // do more stuff

}
```

Method initial state: $\forall$`o,f. Mask[o.f] == 0`
Declare fresh constant π_m to interpret rd amounts, and **`assume`** $0 < \pi_m < 1$

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

Declare $0 < \pi_{call} < 1$ (for 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}
- Havoc heap value at `c.f` if no permission (false)

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

Exhale postcondition

- Check permission: **`assert`** `Mask[c.f] >=` π_m
- Remove permission: `Mask[c.f] -=` π_m

Revisiting aliasing

- Recall previous example:

```
method equals(Cell c)
  requires acc(this.f, ?) && acc(c.f, ?)
  ensures acc(this.f, ?) && acc(c.f, ?)
{
  /* ... comparisons ... */
}
```

Revisiting aliasing

- Recall previous example:

```
method equals(Cell c)
  requires acc(this.f, rd) && acc(c.f, rd)
  ensures acc(this.f, rd) && acc(c.f, rd)
{
  /* ... comparisons ... */
}
```

- Consider the encoding of a call to this method:

assert `Mask[this.f] > 0;`

assume π_{call} `< Mask[this.f];`

`Mask[this.f] -=` π_{call}`;`

assert `Mask[c.f] > 0;`

assume π_{call} `< Mask[c.f];`

`Mask[c.f] -=` π_{call}`;`

What if this = c ?

Revisiting aliasing

- Recall previous example:

```
method equals(Cell c)
  requires acc(this.f, rd) && acc(c.f, rd)
  ensures acc(this.f, rd) && acc(c.f, rd)
{
  /* ... comparisons ... */
}
```

- Consider the encoding of a call to this method:

assert Mask[this.f] > 0;

assume π_{call} < Mask[this.f];

Mask[this.f] -= π_{call};

assert Mask[c.f] > 0;

assume π_{call} < Mask[c.f];

Mask[c.f] -= π_{call};

What if this = c ?

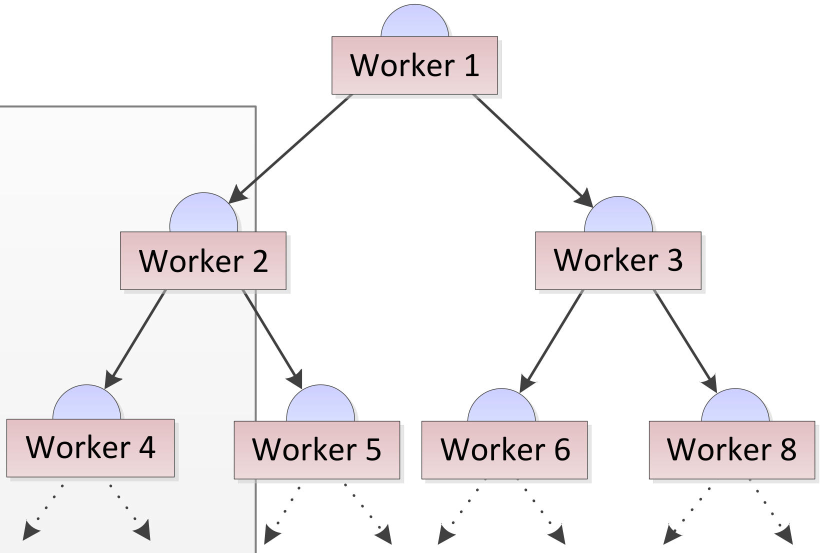
Implicitly, we assume $2 * \pi_{call}$ to be smaller than the amount first held

Workers 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
  }
}
```



Workers 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

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work
    out1 := call w.ask(task1, d);
    out2 := call w.ask(task2, d);
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

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

How do we know we get back all the permissions we gave away?

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

`do` requires read access to (field `f` of) the shared data

`ask` requires read access to the shared data, and gives some permission to the newly-forked thread

Permission expressions

- We need a way to express (unknown) amounts of read permission held by a forked thread
- We also need to be able to express the *difference* between two permission amounts
- We generalise our permissions: acc(e.f, p)
  - where P is a *permission expression*:
    - 1 (and other concrete fractions)
    - rd (abstract read permission, as before)
    - rd(tk) where tk is a token for a forked thread
    - $p_1 + p_2$ or $p_1 - p_2$ (sums and differences)
- Easy to encode, and is much more expressive…

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work
    out1 := call w.ask(task1, d);
    out2 := call w.ask(task2, d);
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

```
requires acc(d.f, 1)
ensures acc(d.f, 1)
```

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd)
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd - rd(result))
```

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work       // 1
    out1 := call w.ask(task1, d);
    out2 := call w.ask(task2, d);
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

```
requires acc(d.f, 1)
ensures acc(d.f, 1)
```

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd)
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd - rd(result))
```

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work        // 1
    out1 := call w.ask(task1, d);  // 1 - rd(out1)
    out2 := call w.ask(task2, d);
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

```
requires acc(d.f, 1)
ensures acc(d.f, 1)
```

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd)
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd - rd(result))
```

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work      // 1
    out1 := call w.ask(task1, d); // 1 - rd(out1)
    out2 := call w.ask(task2, d); // 1 - rd(out1) - rd(out2)
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

```
requires acc(d.f, 1)
ensures acc(d.f, 1)
```

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd)
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd - rd(result))
```

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work       // 1
    out1 := call w.ask(task1, d); // 1 - rd(out1)
    out2 := call w.ask(task2, d); // 1 - rd(out1) - rd(out2)
    // ... drink coffee
    join out1; join out2;          // 1
    d.f := // modify data
  }
```

```
requires acc(d.f, 1)
ensures acc(d.f, 1)
```

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd)
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd - rd(result))
```

```
class Management {
  Data d; // shared data
  ...
  void method manage(Workers w) {
    // ... make up some work          // 1
    out1 := call w.ask(task1, d);     // 1 - rd(out1)
    out2 := call w.ask(task2, d);     // 1 - rd(out1) - rd(out2)
    // ... drink coffee
    join out1; join out2;             // 1
    d.f := // modify data             // ✓ can write
  }
```

```
requires acc(d.f, 1)
ensures acc(d.f, 1)
```

```
class Workers {
  Outcome method do(Task t, Data d)
  { ... }
  token<do> method ask(Task t, Data d)
  {
    out := fork do(t,d);
    return out;
  }
}
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd)
```

```
requires acc(d.f, rd)
ensures acc(d.f, rd - rd(result))
```

Conclusions

- Presented a specification methodology
  - similar expressiveness to fractional permissions
  - avoids concrete values for read permissions
  - allows the user to reason about read/write abstractly
- Provided an efficient encoding (details in paper)
- Soundness argument also in the paper
- Implemented in the *Chalice* tool
  - fork/join, monitors, channels, loops, predicates
  - underlying type for permissions uses Z3 reals
  - performance similar to with concrete fractions only

Future Work

- We cannot express the permission left over after we fork off an *unbounded* number of threads
  - mathematical sums in permission expressions
  - e.g., `acc(x, 1 - Σi rd(tki))` ($\mathtt{acc(x,\ 1\ -}\ \Sigma_i\ \mathtt{rd(tk}_i\mathtt{))}$)
- Exploit fact that abstract read permissions can be repeatedly constrained from above
  - immutability/frozen objects (work in progress)
- rd amounts encoded as prophecy variables
  - treatment could be generalised to allow more uses
  - e.g., equal split amongst unknown no. of threads

End.

Questions?