Fractional Permissions without the Fractions

Alex Summers

ETH Zurich

Joint work with: Stefan Heule, Rustan Leino, Peter Müller
ETH Zurich MSR Redmond ETH Zurich

Overview

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

- Background
- Problem: picking rational values
- Abstract read permissions
- Handling calls, fork/join, monitors
- Permission expressions
- Conclusions

Fractional Permissions (Boyland)

- Provide a way of describing disciplined (race-free) use of shared memory locations.
- Many readers ✓ One writer ✓ Not both.
- Heap locations are managed using *permissions*
  - passed between threads, but never duplicated
- Permission *amounts* are rationals p from $[0,1]$
  - $p=0$ (no permission)
  - $0<p<1$ (read permission)
  - $p=1$ (read/write permission)
- Permissions can be split and recombined

Implicit Dynamic Frames (Smans)

- Uses permissions as assertions to control which threads can read/write to heap locations
- Permissions can be fractional
- Extend first-order logic assertions to additionally include “accessibility predicates”:
  acc(x.f, p) ; we have permission p to location x.f
- For example, acc(x.f,1) && x.f == 4 && acc(x.g,1)
- Permissions treated multiplicatively; i.e.,
  - acc(x.f,p) && acc(x.f,p) ≡ acc(x.f,2p)
- Related to Sep. Logic * [Parkinson/Summers’11]

Chalice (Leino & Müller)

- Verification tool for concurrent programs
  - race-freedom, deadlock-freedom, functional specs
- Specification logic : Implicit Dynamic Frames
- Supports weak fractional permissions
  - acc(e.f, n%) – integer percentages ($0<n\leq 100$)
- Also counting permissions (not discussed here)
- Verification conditions are generated in terms of
  - Heap variable – tracks information about heap
  - Mask variable – tracks permissions currently held
- Modular verification – per method declaration.

Inhale and Exhale

- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{


}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

```
void m()
requires p
ensures q
{

  ...

  call m()

  ...

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

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

  call m()

  ...

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

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


  ...

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

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

}
```

Inhale and Exhale

- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

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

Inhale and Exhale

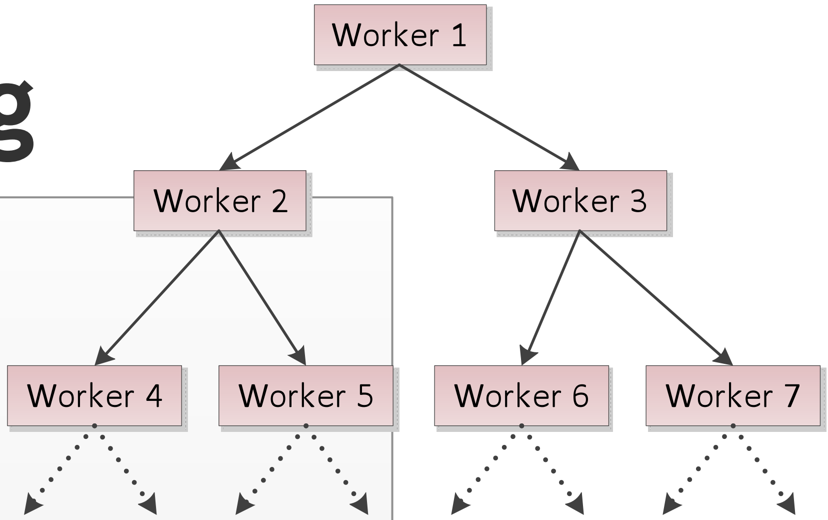
- "inhale P" and "exhale P" are used in Chalice to encode transfers between threads/calls
- "inhale P" means:
  - *assume* heap properties in p
  - gain permissions in p
  - havoc newly-readable locations
- "exhale P" means:
  - *assert* heap properties in p
  - check and give up permissions

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

Problem / Aims

- *We always need to specify fractional (read) permissions using precise (rational) values.*
  - Manual book-keeping is tedious
  - Creates ‘noise’ in specifications, and limits re-use
  - User only cares about read or write permissions
- Aim: abstract over concrete permission amounts
  - User doesn’t choose amounts for read permissions
- Want decent performance from theorem provers
- Also, unbounded splitting of permissions…

Permission splitting




```
class Node {
  Node l,r;

  Outcome work(Data d)
    requires «permission to d.f»;
    ensures «permission to d.f»;
  {
    if (l != null) fork outL := l.work(d);
    if (r != null) fork outR := r.work(d);
    Outcome out := /* work on this node, using d.f */
    if (l != null) out.combine(join outL);
    if (r != null) out.combine(join outR);
    return out;
  }
}
```

How much permission?

Idea: abstract read permissions

- Introduce new read permissions: acc(e.f, rd)
  - Represents an (a priori unknown), positive fractional permission
  - Positive amount: allows reading of location e.f
- Fractions are never expressed precisely
  - We generate (satisfiable) constraints on them
  - Specifications written using just:
    - read permissions: acc(e.f, rd) or simply rd(e.f)
    - write permissions: acc(e.f, 100%) or simply acc(e.f)
  - Different read permissions can refer to different amounts. But, sometimes we want them to match..

Matching rd permissions

- For example, method calls often take some permission and then return it to the caller:

```
method m(c: Cell)
  requires rd(c.val);
  ensures rd(c.val);
{
  /* do something fun... */
}
```

```
method main(c: Cell)
  requires acc(c.val);
{
  c.value := 0;
  call m(c);
  c.value := 1;
}
```

- Rule: *for a given method call*, every rd permission in a method specification is interpreted by the same permission amount

A recursive method …

```
method m(c: Cell)
  requires rd(c.val);
  ensures rd(c.val);
{

  // do stuff

  call m(c);

  // do stuff


}
```

Declare fraction f_m ; used to interpret rd in current method specification: $0 < f_m \leq 1$

Inhale precondition
`Mask[c.val] += f_m`

Exhale precondition for recursive call
- Declare $0 < f_{call} \leq 1$ (rd amounts in recursive call)
- Check that we have *some* permission amount
  assert `Mask[c.val] > 0`
- Constrain f_{call} *to be smaller than permission we have*
  assume f_{call} `< Mask[c.val]`
- Give away this amount: `Mask[c.val] -=` f_{call}

Inhale postcondition: `Mask[c.val] +=` f_{call}

Exhale postcondition
- Check available permission
  assert `Mask[c.val] >=` f_m
- Remove permission from mask
  `Mask[c.val] -=` f_m

Losing permission

- What if we don't intend to return same amount?

```
method m(c: Cell)
  requires rd(c.val);
  ensures rd(c.val);
{
  fork tk := m(c);
}
```

exhale post-condition:
- Check available permission
  `assert Mask[c.val] >= f`$_m$ ✗

- Introduce **rd***

```
method m(c: Cell)
  requires rd(c.val);
  ensures rd*(c.val);
{
  fork tk := m(c);
}
```

represents a different (positive) fraction – with no other information

exhale post-condition:
- Check *some* available permission
  `assert Mask[c.val] > 0` ✓
- Unknown amount returned to caller

Monitors

- Locks are associated with monitor invariants
  - inhale monitor invariant on acquire of lock
  - exhale monitor invariant on release of lock
- How should read permission in monitor invariants be interpreted?
- Recall: for methods, we “choose” a value that is convenient at each call site.
- Can we do the same when we transfer read permission into a monitor?

Monitors

```
class Lock {
  var x: int;
  invariant rd(x);
}
```

- Analogous idea: fix fraction at release

Thread 1

```
/* ... */
release lock;


/* ... */


acquire lock;
/* ... */
```

Fix fraction f_1
Store f_1 in monitor

Get fraction f_1

Monitors

```
class Lock {
  var x: int;
  invariant rd(x);
}
```

- Analogous idea: fix fraction at release

Thread 1

```
/* ... */
release lock;

/* ... */

acquire lock;
/* ... */
```

Fix fraction f_1
Store f_1 in monitor

Get fraction f_1

Thread 2

```
acquire lock;
/* ... */
release lock;
```

Get some fraction f_1

Fix fraction $f_2 \neq f_1$
Store f_2 in monitor

- Fraction needs to be fixed at object creation
  - Not possible at `share` for similar reasons

Monitors

- We need to fix f_{monitor} at object creation
  - No useful information available at this point
  - $0 < f_{\text{monitor}} < 1$

- Less flexible than method calls

```
method main(lock: Lock)
  requires rd(x);
{
  release lock;
}
```

```
class Lock {
  var x: int;
  invariant rd(x);
}
```

Is fraction $f_{\text{monitor}} \leq f_{\text{main}}$?

Monitors

- Solution 1: Use **rd*** (x) in monitor

```
method main(lock: Lock)
  requires rd(x);
{
  release lock;

  acquire lock;
}
```

```
class Lock {
  var x: int;
  invariant rd*(x);
}
```

Only need to check that we have some permission.

- No guarantee that permission we get back is the same, when we re-acquire monitor

Example Revisited

```
class Node {
  Node l,r;

  Outcome work(Data d)
    requires «permission to d.f»;
    ensures «permission to d.f»;
  {
    if (l != null) fork outL := l.work(d);
    if (r != null) fork outR := r.work(d);
    Outcome out := /* work on this node, using d.f */
    if (l != null) out.combine(join outL);
    if (r != null) out.combine(join outR);
    return out;
  }
}
```

Example Revisited

```
class Node {
  Node l,r;

  Outcome work(Data d)
    requires rd(d.f);
    ensures rd(d.f);
  {
    if (l != null) fork outL := l.work(d);
    if (r != null) fork outR := r.work(d);
    Outcome out := /* work on this node, using d.f */
    if (l != null) out.combine(join outL);
    if (r != null) out.combine(join outR);
    return out;
  }
}
```

Some amount(s) given away, but not all

Same amount(s) are retrieved

- **`rd`** permissions sufficient to specify the example

```
class Management {
  Data d; // shared data
  ...
  void 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 do(Task t, Data d, Plan p)
  { ... }
  token<Outcome> ask(Task t, Data d) {
    fork out := call do(t,d,plan);
    return out;
  }
}
```

`do` requires rd access to the shared data

`ask` requires rd access to the shared data, and gives some of this 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*:
    - 100% or rd (as before)
    - rd(tk) where tk is a token for a forked thread
    - rd(m) where m is a monitor
    - $P_1 + P_2$ or $P_1 - P_2$
- Easy to encode, and is much more expressive…

```
class Management {
  Data d; // shared data
  ...
  void 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, 100%)
ensures acc(d.f, 100%)
```

```
class Workers {
  Outcome do(Task t, Data d, Plan p)
  { ... }
  token<Outcome> ask(Task t, Data d) {
    fork out := call do(t,d,plan);
    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 manage(Workers w) {
    // ... make up some work        // 100%
    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, 100%)
ensures acc(d.f, 100%)
```

```
class Workers {
  Outcome do(Task t, Data d, Plan p)
  { ... }
  token<Outcome> ask(Task t, Data d) {
    fork out := call do(t,d,plan);
    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 manage(Workers w) {
    // ... make up some work          // 100%
    out1 := call w.ask(task1, d);    // 100% - rd(out1)
    out2 := call w.ask(task2, d);
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

```
requires acc(d.f, 100%)
ensures acc(d.f, 100%)
```

```
class Workers {
  Outcome do(Task t, Data d, Plan p)
  { ... }
  token<Outcome> ask(Task t, Data d) {
    fork out := call do(t,d,plan);
    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 manage(Workers w) {
    // ... make up some work          // 100%
    out1 := call w.ask(task1, d);   // 100% - rd(out1)
    out2 := call w.ask(task2, d);   // 100% - rd(out1) - rd(out2)
    // ... drink coffee
    join out1; join out2;
    d.f := // modify data
  }
```

```
requires acc(d.f, 100%)
ensures acc(d.f, 100%)
```

```
class Workers {
  Outcome do(Task t, Data d, Plan p)
  { ... }
  token<Outcome> ask(Task t, Data d) {
    fork out := call do(t,d,plan);
    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 manage(Workers w) {
    // ... make up some work          // 100%
    out1 := call w.ask(task1, d);     // 100% - rd(out1)
    out2 := call w.ask(task2, d);     // 100% - rd(out1) - rd(out2)
    // ... drink coffee
    join out1; join out2;             // 100%
    d.f := // modify data
  }
```

```
requires acc(d.f, 100%)
ensures acc(d.f, 100%)
```

```
class Workers {
  Outcome do(Task t, Data d, Plan p)
  { ... }
  token<Outcome> ask(Task t, Data d) {
    fork out := call do(t,d,plan);
    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 manage(Workers w) {
    // ... make up some work        // 100%
    out1 := call w.ask(task1, d);   // 100% - rd(out1)
    out2 := call w.ask(task2, d);   // 100% - rd(out1) – rd(out2)
    // ... drink coffee
    join out1; join out2;           // 100%
    d.f := // modify data           // ✓ can write
  }
```

```
requires acc(d.f, 100%)
ensures acc(d.f, 100%)
```

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

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

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

Monitors

- Recall the awkward situation with monitors:

```
method main(Lock: lock)
  requires rd(x);
{
  release lock;

  acquire lock;
}
```

```
class Lock {
  int x;
  invariant rd(x);
}
```

Monitors

- Solution 2: Using the permission expressions

```
method main(Lock lock)
  requires acc(x,rd(lock));
{
  release lock;

  acquire lock;
}
```

```
class Lock {
  int x;
  invariant rd(x);
}
```

- Now we can express exactly the amount of permission we need to exhale to the monitor.

Summary and Extras

- Presented a specification methodology:
  - similar expressiveness to fractional permissions
  - avoids explicit "values" for read permissions
  - allows user to reason about read/write abstractly
- Supports full Chalice language
  - fork/join, channels, predicates, loop invariants
- Methodology is implemented
  - backwards-compatible with a few easy edits
  - permission encoding uses only integer-typed data
  - performance comparable with existing encoding

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, 100% -`** Σ_i **`rd(tk`**$_i$**`))`**
  - some careful encoding is required to perform well
- In some obscure cases, permission *multiplication* arises
  - non-linear arithmetic tends to perform badly
- Experiment with encoding harder fractional examples using abstract permission expressions

End.

Are there any questions?