An introduction to Iris
part 2

Jean-Marie Madiot & François Pottier

JFLA 2026, Oberbronn

References on exercises

- more info on the proof mode here you should get this here: https://gitlab.mpi-sws.org/iris/iris/blob/master/docs/proof_mode.md
  or searching online “iris proof mode”
- more guided examples/exercises in the POPL 2020 Iris tutorial
- on popular demand I *could* do the exercises

An example

Combining progress about a shared resource

A common situation: several threads work on a shared resource protected by a mutex. Once they are done, the resource must satisfy some property accordingly.

Combining progress about a shared resource

A common situation: several threads work on a shared resource protected by a mutex. Once they are done, the resource must satisfy some property accordingly.

```
          let r = ref 0 in
          let l = newlock () in
                (fork)
acquire l;     ||    acquire l;
r := !r + 1;   ||    r := !r + 1;
release l      ||    release l
                (join)
          acquire l;
          assert (!r = 2)
```

Combining progress about a shared resource

A common situation: several threads work on a shared resource protected by a mutex. Once they are done, the resource must satisfy some property accordingly.

```
          let r = ref 0 in
          let l = newlock () in
                (fork)
acquire l;     ||    acquire l;
r := !r + 1;   ||    r := !r + 1;
release l      ||    release l
                (join)
          acquire l;
          assert (!r = 2)
```

Let us prove safety with the lock rules. Reminder of the lock Hoare triples:

$$\{R\}\ \texttt{newlock ()}\ \{\ell.\Box \mathit{isLock}\,\ell\,R\}$$

$$\mathit{isLock}\,\ell\,R \vdash \{\mathit{True}\}\ \texttt{acquire}\ \ell\ \{R\}$$

$$\mathit{isLock}\,\ell\,R \vdash \{R\}\ \texttt{release}\ \ell\ \{\mathit{True}\}$$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{\mathit{isLock}\ l\ R\}$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\} \qquad \| \qquad \{\}$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{\mathit{isLock}\ l\ R\}$ (persistent)

$\{\}$	$\Vert$	$\{\}$
`acquire l;`		`acquire l`

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\}$	$\Vert$	$\{\}$
`acquire l;`	$\Vert$	`acquire l`
$\{R\}$	$\Vert$	$\{R\}$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\}$	‖	$\{\}$
`acquire l;`	‖	`acquire l`
$\{R\}$	‖	$\{R\}$
`r := !r + 1;`	‖	`r := !r + 1;`

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\}$	‖	$\{\}$
`acquire l;`	‖	`acquire l`
$\{R\}$	‖	$\{R\}$
`r := !r + 1;`	‖	`r := !r + 1;`
$\{R\}$	‖	$\{R\}$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\}$	‖	$\{\}$
`acquire l;`	‖	`acquire l`
$\{R\}$	‖	$\{R\}$
`r := !r + 1;`	‖	`r := !r + 1;`
$\{R\}$	‖	$\{R\}$
`release l`	‖	`release l`
$\{\}$	‖	$\{\}$

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\}$	∥	$\{\}$
`acquire l;`	∥	`acquire l`
$\{R\}$	∥	$\{R\}$
`r := !r + 1;`	∥	`r := !r + 1;`
$\{R\}$	∥	$\{R\}$
`release l`	∥	`release l`
$\{\}$	∥	$\{\}$

```
acquire l;
```

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{\mathit{isLock}\ l\ R\}$ (persistent)

$\{\}$	‖	$\{\}$
`acquire l;`	‖	`acquire l`
$\{R\}$	‖	$\{R\}$
`r := !r + 1;`	‖	`r := !r + 1;`
$\{R\}$	‖	$\{R\}$
`release l`	‖	`release l`
$\{\}$	‖	$\{\}$

```
acquire l;
```

$\{R\}$

```
assert (!r = 2)
```

How a proof with invariants would go

$\{\}$

```
let r = ref 0 in
```

$\{r \mapsto 0\}$ so for some clever R,

$\{R\}$

```
let l = newlock ()
```

$\{isLock\ l\ R\}$ (persistent)

$\{\}$	$\{\}$
`acquire l;`	`acquire l`
$\{R\}$	$\{R\}$
`r := !r + 1;`	`r := !r + 1;`
$\{R\}$	$\{R\}$
`release l`	`release l`
$\{\}$	$\{\}$

```
acquire l;
```

$\{R\}$

```
assert (!r = 2)
```

impossible: R is clever but *invariant*

What are invariants lacking?

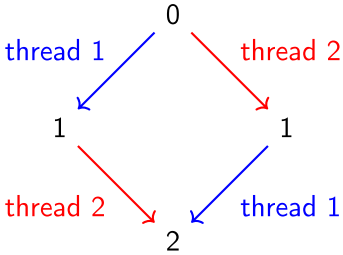
At high-level the program has two interleavings:

```
    let r = ref 0 in
    let l = newlock () in
acquire l;   ||  acquire l;
r := !r + 1; ||  r := !r + 1;
release l    ||  release l
    acquire l;
    assert (!r = 2)
```

What are invariants lacking?

At high-level the program has two interleavings:

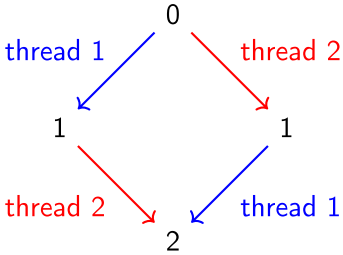
```
    let r = ref 0 in
    let l = newlock () in
acquire l;    ||  acquire l;
r := !r + 1;  ||  r := !r + 1;
release l     ||  release l
    acquire l;
    assert (!r = 2)
```



What are invariants lacking?

At high-level the program has two interleavings:

```
    let r = ref 0 in
    let l = newlock () in
acquire l;   ||  acquire l;
r := !r + 1; ||  r := !r + 1;
release l    ||  release l
    acquire l;
    assert (!r = 2)
```

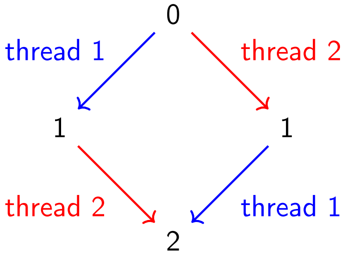


A proof would need need to reflect this somehow.

What are invariants lacking?

At high-level the program has two interleavings:

```
    let r = ref 0 in
    let l = newlock () in
acquire l;   ||  acquire l;
r := !r + 1; ||  r := !r + 1;
release l    ||  release l
    acquire l;
    assert (!r = 2)
```



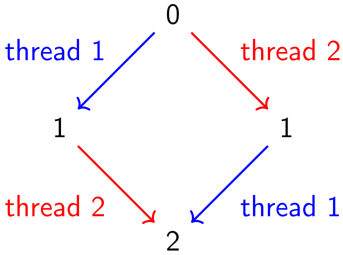
A proof would need need to reflect this somehow.

- some notion of *state* embedded into the separation logic

What are invariants lacking?

At high-level the program has two interleavings:

```
    let r = ref 0 in
    let l = newlock () in
acquire l;    ||  acquire l;
r := !r + 1;  ||  r := !r + 1;
release l     ||  release l
    acquire l;
    assert (!r = 2)
```



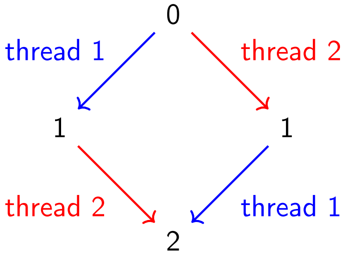
A proof would need need to reflect this somehow.

- some notion of *state* embedded into the separation logic
- *splitting* (and *combining*) states into parts for each thread

What are invariants lacking?

At high-level the program has two interleavings:

```
    let r = ref 0 in
    let l = newlock () in
acquire l;   ||  acquire l;
r := !r + 1; ||  r := !r + 1;
release l    ||  release l
    acquire l;
    assert (!r = 2)
```



A proof would need need to reflect this somehow.

- some notion of *state* embedded into the separation logic
- *splitting* (and *combining*) states into parts for each thread
- all possibles orders: combining is *commutative* and *associative* (same reason as for $*$)

One way to add state: auxiliary variables

We could change the program: /

```
        let r = ref 0 in
        let r1 = ref 0 in
        let r2 = ref 0 in
        let l = newlock () in
acquire l;          ||      acquire l;
r := !r + 1         ||      r := !r + 1
r1 := !r1 + 1       ||      r2 := !r2 + 1
release l           ||      release l
        acquire l;
        assert (!r = 2);
        release l
```

One way to add state: auxiliary variables

We could change the program: /

```
          let r = ref 0 in
          let r1 = ref 0 in
          let r2 = ref 0 in
          let l = newlock () in
acquire l;        ||      acquire l;
r := !r + 1       ||      r := !r + 1
r1 := !r1 + 1     ||      r2 := !r2 + 1
release l         ||      release l
          acquire l;
          assert (!r = 2);
          release l
```

Invariant: "the lock *owns* `r` and it is the sum of `r1` and `r2`, which are shared".

One way to add state: auxiliary variables

We could change the program: /

```
                let r = ref 0 in
                let r1 = ref 0 in
                let r2 = ref 0 in
                let l = newlock () in
acquire l;              ||          acquire l;
r := !r + 1             ||          r := !r + 1
r1 := !r1 + 1           ||          r2 := !r2 + 1
release l               ||          release l
                acquire l;
                assert (!r = 2);
                release l
```

Invariant: "the lock *owns* r and it is the sum of r1 and r2, which are shared".

Doable by 'splitting' r1 and r2: fractional permissions (Boyland, 2003):

$$isLock\ l\ (\exists n \exists n_1 \exists n_2\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * n = n_1 + n_2)$$

One way to add state: auxiliary variables

We could change the program: /

```
                let r = ref 0 in
                let r1 = ref 0 in
                let r2 = ref 0 in
                let l = newlock () in
        acquire l;          ||      acquire l;
        r := !r + 1         ||      r := !r + 1
        r1 := !r1 + 1       ||      r2 := !r2 + 1
        release l           ||      release l
                acquire l;
                assert (!r = 2);
                release l
```

Invariant: "the lock *owns* `r` and it is the sum of `r1` and `r2`, which are shared".

Doable by 'splitting' `r1` and `r2`: fractional permissions (Boyland, 2003):

$$isLock\ l\ (\exists n \exists n_1 \exists n_2\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * n = n_1 + n_2)$$

but: 1) not modular, 2) changes the code 3) a special case of ghost

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

```
let l = newlock ()
```

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

```
let l = newlock ()
```

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$ $\quad\|\quad$ $\{\boxed{A'}\}$

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

```
let l = newlock ()
```

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$	$\|$	$\{\boxed{A'}\}$
`acquire l;`	$\|$	`acquire l`
`r := !r + 1;`	$\|$	`r := !r + 1`
`release l;`	$\|$	`release l`

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

```
let l = newlock ()
```

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$	$\{\boxed{A'}\}$
`acquire l;`	`acquire l`
`r := !r + 1;`	`r := !r + 1`
`release l;`	`release l`
$\{\boxed{B}\}$	$\{\boxed{B'}\}$

Desiderata for a proof with ghost state

let r = ref 0 **in**

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

let l = newlock ()

$\{\mathit{isLock}\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$	$\{\boxed{A'}\}$
`acquire l;`	`acquire l`
`r := !r + 1;`	`r := !r + 1`
`release l;`	`release l`
$\{\boxed{B}\}$	$\{\boxed{B'}\}$

$\{\boxed{B} * \boxed{B'}\}$

Desiderata for a proof with ghost state

```
let r = ref 0 in
```

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

```
let l = newlock ()
```

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$	‖	$\{\boxed{A'}\}$
`acquire l;`	‖	`acquire l`
`r := !r + 1;`	‖	`r := !r + 1`
`release l;`	‖	`release l`
$\{\boxed{B}\}$	‖	$\{\boxed{B'}\}$

$\{\boxed{B} * \boxed{B'}\}$

```
acquire l;
```

$\{R * \boxed{B} * \boxed{B'}\}$

Desiderata for a proof with ghost state

`let r = ref 0 in`

$\{r \mapsto 0\}$, find R, $\boxed{A}$, $\boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

`let l = newlock ()`

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

Thread 1		Thread 2
$\{\boxed{A}\}$	‖	$\{\boxed{A'}\}$
`acquire l;`	‖	`acquire l`
`r := !r + 1;`	‖	`r := !r + 1`
`release l;`	‖	`release l`
$\{\boxed{B}\}$	‖	$\{\boxed{B'}\}$

$\{\boxed{B} * \boxed{B'}\}$

`acquire l;`

$\{R * \boxed{B} * \boxed{B'}\}$

$\{r \mapsto 2\}$

`assert (!r = 2)`

Desiderata for a proof with ghost state

let r = ref 0 **in**

$\{r \mapsto 0\}$, find $R, \boxed{A}, \boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

let l = newlock ()

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$	$\{\boxed{A'}\}$
acquire l;	acquire l
r := !r + 1;	r := !r + 1
release l;	release l
$\{\boxed{B}\}$	$\{\boxed{B'}\}$

$\{\boxed{B} * \boxed{B'}\}$

acquire l;

$\{R * \boxed{B} * \boxed{B'}\}$

$\{r \mapsto 2\}$

assert (!r = 2)

Ghost updates/triples needed:

1. $r \mapsto 0 \Rrightarrow R * \boxed{A} * \boxed{A'}$
2. $\{R * \boxed{A}\}$ r := !r + 1 $\{R * \boxed{B}\}$
3. $\{R * \boxed{A'}\}$ r := !r + 1 $\{R * \boxed{B'}\}$
4. $R * \boxed{B'} * \boxed{B} \Rrightarrow r \mapsto 2$

Desiderata for a proof with ghost state

let `r = ref 0` **in**

$\{r \mapsto 0\}$, find $R, \boxed{A}, \boxed{A'}$ s.t.:

$\{R * \boxed{A} * \boxed{A'}\}$

let `l = newlock ()`

$\{isLock\ l\ R * \boxed{A} * \boxed{A'}\}$

$\{\boxed{A}\}$	∥	$\{\boxed{A'}\}$
`acquire l;`	∥	`acquire l`
`r := !r + 1;`	∥	`r := !r + 1`
`release l;`	∥	`release l`
$\{\boxed{B}\}$	∥	$\{\boxed{B'}\}$

$\{\boxed{B} * \boxed{B'}\}$

`acquire l;`

$\{R * \boxed{B} * \boxed{B'}\}$

$\{r \mapsto 2\}$

assert `(!r = 2)`

Ghost updates/triples needed:

1. $r \mapsto 0 \Rrightarrow R * \boxed{A} * \boxed{A'}$
2. $\{R * \boxed{A}\}$ `r := !r + 1` $\{R * \boxed{B}\}$
3. $\{R * \boxed{A'}\}$ `r := !r + 1` $\{R * \boxed{B'}\}$
4. $R * \boxed{B'} * \boxed{B} \Rrightarrow r \mapsto 2$

Both write to r so r goes in R. Let's try:

$$R = \exists n\ r \mapsto n * \boxed{P(n)}$$

for some clever P.

Constraint #1: analysis

$$R = \exists n\; r \mapsto n * \boxed{P(n)}$$

Trying to prove the first constraint, derivation-style:

$$\dfrac{\dfrac{\vdots}{\color{green}{True \Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'}}}}{\dfrac{r \mapsto 0 \Rrightarrow r \mapsto 0 * \boxed{P(0)} * \boxed{A} * \boxed{A'}}{r \mapsto 0 \Rrightarrow \exists n\; r \mapsto n * \boxed{P(n)} * \boxed{A} * \boxed{A'}}\;\text{exists-intro}}\;\text{frame}$$

Constraint #2 and #3

$$R = \exists n\; r \mapsto n * \boxed{P(n)}$$

$$\vdots$$

$$\boxed{P(n)} * \boxed{A} \Rrightarrow \boxed{P(n+1)} * \boxed{B}$$

frame

$$r \mapsto n+1 * \boxed{P(n)} * \boxed{A} \Rrightarrow r \mapsto n+1 * \boxed{P(n+1)} * \boxed{B}$$

exists-right

$$r \mapsto n+1 * \boxed{P(n)} * \boxed{A} \Rrightarrow \exists n'\; r \mapsto n' * \boxed{P(n')} * \boxed{B}$$

incr+seq

$$\{r \mapsto n * \boxed{P(n)} * \boxed{A}\}\ \texttt{r := !r + 1}\ \{\exists n'\; r \mapsto n' * \boxed{P(n')} * \boxed{B}\}$$

exists-left

$$\{R * \boxed{A}\}\ \texttt{r := !r + 1}\ \{R * \boxed{B}\}$$

acquire+seq+release

$$\{\boxed{A}\}\ \texttt{acquire r; r := !r + 1; release r}\ \{\boxed{B}\}$$

Constraint #4

Now to conclude the program:

$$
\dfrac{\dfrac{\dfrac{\vdots}{\boxed{P(n)} * \boxed{B} * \boxed{B'} \;\Rrightarrow\; n = 2}\;\text{frame}}{r \mapsto n * \boxed{P(n)} * \boxed{B} * \boxed{B'} \;\Rrightarrow\; r \mapsto n * n = 2}\;\text{conseq}}{\dfrac{r \mapsto n * \boxed{P(n)} * \boxed{B} * \boxed{B'} \;\Rrightarrow\; r \mapsto 2}{R * \boxed{B} * \boxed{B'} \;\Rrightarrow\; r \mapsto 2}\;\text{exists-intro}}
$$

Ghost updates needed for our example

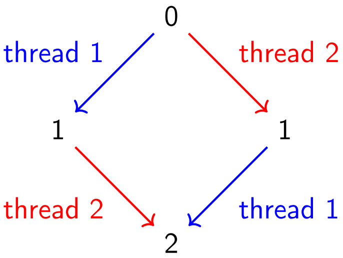
Purely in terms of ghost, our constraints are, for all n,

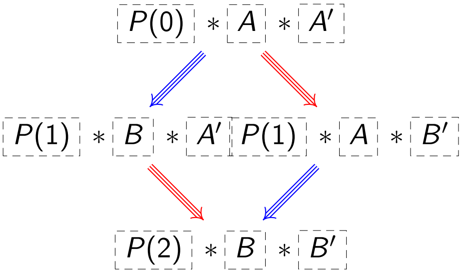
$$\begin{aligned}
\mathit{True} &\Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'} \\
\boxed{P(n)} * \boxed{A} &\Rrightarrow \boxed{P(n+1)} * \boxed{B} \\
\boxed{P(n)} * \boxed{A'} &\Rrightarrow \boxed{P(n+1)} * \boxed{B'} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} &\Rightarrow n = 2
\end{aligned}$$

Ghost updates needed for our example

Purely in terms of ghost, our constraints are, for all n,

$$
\begin{aligned}
\mathit{True} &\Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'} \\
\boxed{P(n)} * \boxed{A} &\Rrightarrow \boxed{P(n+1)} * \boxed{B} \\
\boxed{P(n)} * \boxed{A'} &\Rrightarrow \boxed{P(n+1)} * \boxed{B'} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} &\Rrightarrow n = 2
\end{aligned}
$$

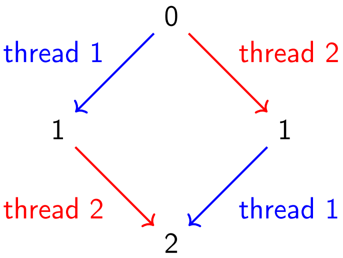


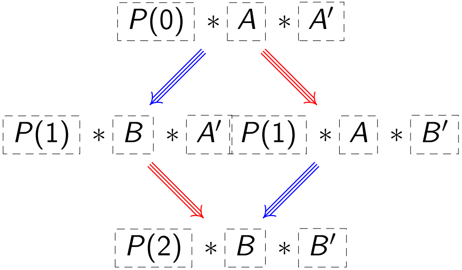


Ghost updates needed for our example

Purely in terms of ghost, our constraints are, for all n,

$$\begin{aligned}
\mathit{True} &\Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'} && \textit{ghost allocation} \\
\color{blue}{\boxed{P(n)} * \boxed{A}} &\color{blue}{\Rrightarrow} \color{blue}{\boxed{P(n+1)} * \boxed{B}} \\
\color{red}{\boxed{P(n)} * \boxed{A'}} &\color{red}{\Rrightarrow} \color{red}{\boxed{P(n+1)} * \boxed{B'}} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} &\Rightarrow n = 2
\end{aligned}$$

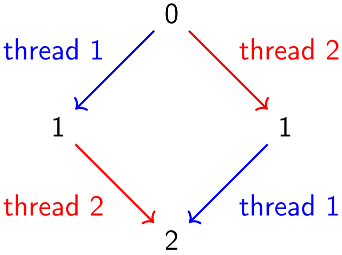


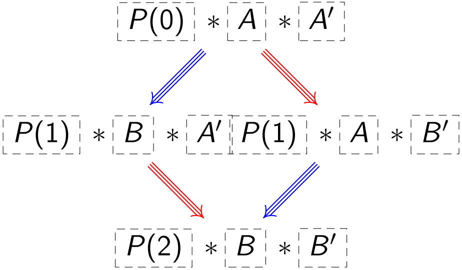


Ghost updates needed for our example

Purely in terms of ghost, our constraints are, for all n,

$$
\begin{array}{rcll}
\mathit{True} & \Rrightarrow & \boxed{P(0)} * \boxed{A} * \boxed{A'} & \textit{ghost allocation} \\
\boxed{P(n)} * \boxed{A} & \Rrightarrow & \boxed{P(n+1)} * \boxed{B} & \textit{ghost update} \\
\boxed{P(n)} * \boxed{A'} & \Rrightarrow & \boxed{P(n+1)} * \boxed{B'} & \textit{ghost update} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} & \Rrightarrow & n = 2 &
\end{array}
$$

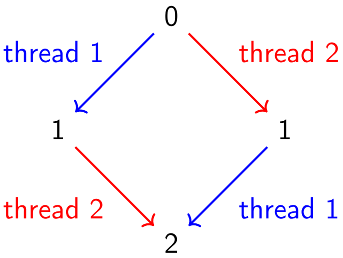


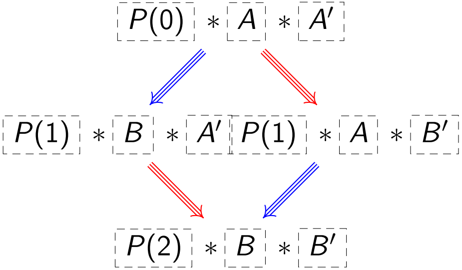


Ghost updates needed for our example

Purely in terms of ghost, our constraints are, for all n,

$$
\begin{array}{rcll}
\mathit{True} & \Rrightarrow & \boxed{P(0)} * \boxed{A} * \boxed{A'} & \textit{ghost allocation} \\
\boxed{P(n)} * \boxed{A} & \Rrightarrow & \boxed{P(n+1)} * \boxed{B} & \textit{ghost update} \\
\boxed{P(n)} * \boxed{A'} & \Rrightarrow & \boxed{P(n+1)} * \boxed{B'} & \textit{ghost update} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} & \Rightarrow & n = 2 & ???
\end{array}
$$





Step back

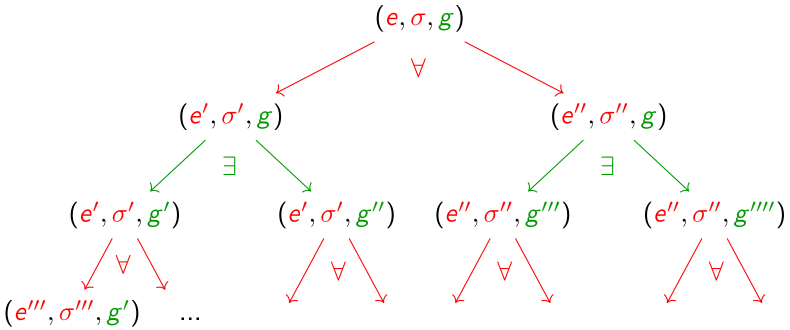
What is ghost state?

Demonic ($\forall$) and angelic ($\exists$) non-determinism

Correctness of a non-deterministic program requires correctness for all physical steps, but for each, we get to choose a ghost update:

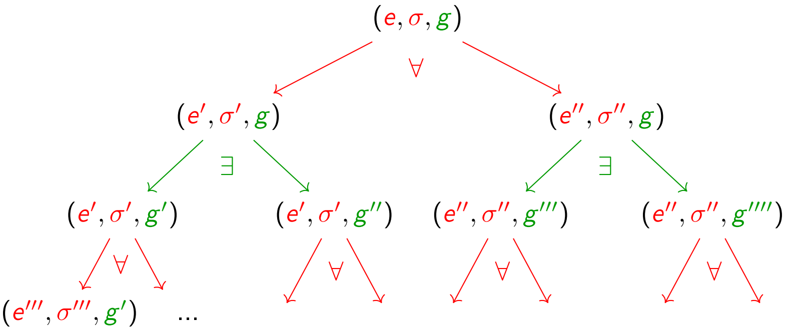
Demonic (∀) and angelic (∃) non-determinism

Correctness of a non-deterministic program requires correctness for all physical steps, but for each, we get to choose a ghost update:



Demonic (∀) and angelic (∃) non-determinism

Correctness of a non-deterministic program requires correctness for all physical steps, but for each, we get to choose a ghost update:



Visible in the definition of *wp*, e.g. when e is not a value $wp\ e\ \phi \triangleq$

$$\ldots \forall \sigma\ S(\sigma) \twoheadrightarrow \Rrightarrow \ldots \forall e' \forall \sigma'\ (e, \sigma) \rightarrow (e', \sigma') \twoheadrightarrow \Rrightarrow\ S(\sigma') * wp\ e'\ \phi$$

where $\Rrightarrow$ contains an existential: $[\![\Rrightarrow P]\!](a) \triangleq \ldots \exists b \ldots [\![P]\!](b)$

Composition, monoids, updates

Associative symmetric composition: our ghost state is a *monoid* $(\mathcal{M}, \cdot)$ – in fact a semigroup

At any given time:

- each thread t_i "owns" one element $g_i \in \mathcal{M}$, called **resource**
  ownership of g_i is written $\boxed{g_i}$.

Composition, monoids, updates

Associative symmetric composition: our ghost state is a *monoid* $(\mathcal{M}, \cdot)$ – in fact a semigroup

At any given time:

- each thread t_i "owns" one element $g_i \in \mathcal{M}$, called **resource**
  ownership of g_i is written $\boxed{g_i}$.
- each unopened invariant I_j "owns" a resource h_j that satisfies it

Composition, monoids, updates

Associative symmetric composition: our ghost state is a *monoid* $(\mathcal{M}, \cdot)$ – in fact a semigroup

At any given time:

- each thread t_i "owns" one element $g_i \in \mathcal{M}$, called **resource**
  ownership of g_i is written $\boxed{g_i}$.
- each unopened invariant I_j "owns" a resource h_j that satisfies it
- the combination $g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k$ is the global resource

Composition, monoids, updates

Associative symmetric composition: our ghost state is a *monoid* $(\mathcal{M}, \cdot)$ – in fact a semigroup

At any given time:

- each thread t_i "owns" one element $g_i \in \mathcal{M}$, called **resource** ownership of g_i is written $\boxed{g_i}$.
- each unopened invariant I_j "owns" a resource h_j that satisfies it
- the combination $g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k$ is the global resource

The prover performs updates both: $\boxed{g_i} \Rrightarrow \boxed{g_i'}$ and $\boxed{h_j} \Rrightarrow \boxed{h_j'}$.

Composition maps to separation $\boxed{g \cdot h} = \boxed{g} * \boxed{h}$

Composition, monoids, updates

Associative symmetric composition: our ghost state is a *monoid* $(\mathcal{M}, \cdot)$ – in fact a semigroup

At any given time:

- each thread t_i "owns" one element $g_i \in \mathcal{M}$, called **resource**
  ownership of g_i is written $\boxed{g_i}$.
- each unopened invariant I_j "owns" a resource h_j that satisfies it
- the combination $g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k$ is the global resource

The prover performs updates both: $\boxed{g_i} \Rrightarrow \boxed{g'_i}$ and $\boxed{h_j} \Rrightarrow \boxed{h'_j}$.

Composition maps to separation $\boxed{g \cdot h} = \boxed{g} * \boxed{h}$

Updating is all well and good but what can we conclude from $\boxed{g}$?

How to escape the ghost box?

Validity

Idea: pick an invariant on the global resource, *validity*: $\mathit{valid} : \mathcal{M} \rightarrow \texttt{Prop}$

$$\mathit{valid}(g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k) \;\; \text{at all times}$$

Validity

Idea: pick an invariant on the global resource, *validity*: $\mathit{valid} : \mathcal{M} \to \texttt{Prop}$

$$\mathit{valid}(g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k) \;\; \text{at all times}$$

Follows rely-guarantee-style protocol:

Validity

Idea: pick an invariant on the global resource, *validity*: $\mathit{valid} : \mathcal{M} \rightarrow \texttt{Prop}$

$$\mathit{valid}(g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k) \quad \text{at all times}$$

Follows rely-guarantee-style protocol:

- ownership $\boxed{g_i}$ *provides* validity of some global resource:

$$\overline{\boxed{g_i} \Rightarrow \exists g_{others} \in \mathcal{M}\ \mathit{valid}(g_i \cdot g_{others})}$$

Validity

Idea: pick an invariant on the global resource, *validity*: $\mathit{valid} : \mathcal{M} \to \texttt{Prop}$

$$\mathit{valid}(g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k) \text{ at all times}$$

Follows rely-guarantee-style protocol:

- ownership $\boxed{g_i}$ *provides* validity of some global resource:

$$\frac{}{\boxed{g_i} \Rightarrow \exists g_{others} \in \mathcal{M}\ \mathit{valid}(g_i \cdot g_{others})} \quad \text{or just} \quad \frac{}{\boxed{g_i} \Rightarrow \mathit{valid}(g_i)}$$

Validity

Idea: pick an invariant on the global resource, *validity*: $\mathit{valid} : \mathcal{M} \to \texttt{Prop}$

$$\mathit{valid}(g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k) \;\; \text{at all times}$$

Follows rely-guarantee-style protocol:

- ownership $\boxed{g_i}$ *provides* validity of some global resource:

$$\frac{}{\boxed{g_i} \Rightarrow \exists g_{others} \in \mathcal{M} \; \mathit{valid}(g_i \cdot g_{others})} \quad \text{or just} \quad \frac{}{\boxed{g_i} \Rightarrow \mathit{valid}(g_i)}$$

- an update $\boxed{g_i} \Rrightarrow \boxed{g_i'}$ *requires* preservation of global validity:

$$\frac{\forall g \in \mathcal{M} \;\; \mathit{valid}(g_i \cdot g) \Rightarrow \mathit{valid}(g_i' \cdot g) \qquad \mathit{valid}(g_i) \Rightarrow \mathit{valid}(g_i')}{\boxed{g_i} \Rrightarrow \boxed{g_i'}}$$

Validity

Idea: pick an invariant on the global resource, *validity*: $valid : \mathcal{M} \to \texttt{Prop}$

$$valid(g_1 \cdot \ldots \cdot g_n \cdot h_1 \cdot \ldots \cdot h_k) \ \text{ at all times}$$

Follows rely-guarantee-style protocol:

- ownership $\boxed{g_i}$ *provides* validity of some global resource:

$$\frac{}{\boxed{g_i} \Rightarrow \exists g_{others} \in \mathcal{M} \ valid(g_i \cdot g_{others})} \quad \text{or just} \quad \frac{}{\boxed{g_i} \Rightarrow valid(g_i)}$$

- an update $\boxed{g_i} \Rrightarrow \boxed{g_i'}$ *requires* preservation of global validity:

$$\frac{\forall g \in \mathcal{M} \ \ valid(g_i \cdot g) \Rightarrow valid(g_i' \cdot g) \qquad valid(g_i) \Rightarrow valid(g_i')}{\boxed{g_i} \Rrightarrow \boxed{g_i'}} \quad \triangleq g_i \rightsquigarrow g_i'$$

$g_i \rightsquigarrow g_i'$ is called a *frame-preserving update*

Designing the appropriate monoid

Smaller subproblems: split P as $\boxed{P(n)} = \exists xy\ \boxed{Q(x)} * \boxed{Q'(y)} * n = x + y$

$$\text{(Before splitting)}$$

$$\begin{aligned}
\mathit{True} &\Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'} \\
\boxed{P(n)} * \boxed{A} &\Rrightarrow \boxed{P(n+1)} * \boxed{B} \\
\boxed{P(n)} * \boxed{A'} &\Rrightarrow \boxed{P(n+1)} * \boxed{B'} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} &\Rrightarrow n = 2
\end{aligned}$$

Designing the appropriate monoid

Smaller subproblems: split P as $\boxed{P(n)} = \exists xy\ \boxed{Q(x)} * \boxed{Q'(y)} * n = x + y$

$$(\textit{Before splitting})$$

$$\begin{aligned}
\mathit{True} &\Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'} \\
\boxed{P(n)} * \boxed{A} &\Rrightarrow \boxed{P(n+1)} * \boxed{B} \\
\boxed{P(n)} * \boxed{A'} &\Rrightarrow \boxed{P(n+1)} * \boxed{B'} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} &\Rrightarrow n = 2
\end{aligned}$$

$$(\textit{Subproblem} + \textit{more steps})$$

$$\begin{aligned}
\mathit{True} &\Rrightarrow \boxed{Q(0)} * \boxed{A} \\
\boxed{Q(0)} * \boxed{A} &\Rrightarrow \boxed{Q(1)} * \boxed{B} \\
\boxed{Q(x)} * \boxed{A} &\Rrightarrow x = 0 \\
\boxed{Q(x)} * \boxed{B} &\Rrightarrow x = 1
\end{aligned}$$

$$(\textit{same for } \boxed{A'}, \boxed{B'}, \boxed{Q'()})$$

Designing the appropriate monoid

Smaller subproblems: split P as $\boxed{P(n)} = \exists xy\ \boxed{Q(x)} * \boxed{Q'(y)} * n = x + y$

$$(\textit{Before splitting})$$

$$\begin{aligned}
\textit{True} &\Rrightarrow \boxed{P(0)} * \boxed{A} * \boxed{A'} \\
\boxed{P(n)} * \boxed{A} &\Rrightarrow \boxed{P(n+1)} * \boxed{B} \\
\boxed{P(n)} * \boxed{A'} &\Rrightarrow \boxed{P(n+1)} * \boxed{B'} \\
\boxed{P(n)} * \boxed{B} * \boxed{B'} &\Rrightarrow n = 2
\end{aligned}$$

$$(\textit{Subproblem} + \textit{more steps})$$

$$\begin{aligned}
\textit{True} &\Rrightarrow \boxed{Q(0)} * \boxed{A} \\
\boxed{Q(0)} * \boxed{A} &\Rrightarrow \boxed{Q(1)} * \boxed{B} \\
\boxed{Q(x)} * \boxed{A} &\Rrightarrow x = 0 \\
\boxed{Q(x)} * \boxed{B} &\Rrightarrow x = 1
\end{aligned}$$

$$(\textit{same for } \boxed{A'}, \boxed{B'}, \boxed{Q'()})$$

Equivalent goal: find $Q(0)$, $Q(1)$, A, B such that:

- $\boxed{Q(1)} * \boxed{A} \Rrightarrow \textit{False}$
- $\boxed{Q(0)} * \boxed{B} \Rrightarrow \textit{False}$
- $Q(0) \cdot A$ is "the whole thing", so that: $Q(0) \cdot A \rightsquigarrow Q(1) \cdot B$

Commutative-monoid-with-validity $(\mathcal{M}_{half}, \cdot)$

$$\mathcal{M}_{half} ::= full(0) \mid full(1) \mid half(0) \mid half(1) \mid \times$$

Commutative-monoid-with-validity $(\mathcal{M}_{half}, \cdot)$

$$\mathcal{M}_{half} ::= full(0) \mid full(1) \mid half(0) \mid half(1) \mid \times$$

Operations and validity:

$- \cdot -$	$full(x)$	$half(0)$	$half(1)$	$\times$
$full(y)$	$\times$	$\times$	$\times$	$\times$
$half(0)$	$\times$	$full(0)$	$\times$	$\times$
$half(1)$	$\times$	$\times$	$full(1)$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$

$valid(-)$	*True*	*True*	*True*	*False*

Commutative-monoid-with-validity $(\mathcal{M}_{half}, \cdot)$

$$\mathcal{M}_{half} ::= \mathit{full}(0) \mid \mathit{full}(1) \mid \mathit{half}(0) \mid \mathit{half}(1) \mid \times$$

Operations and validity:

$- \cdot -$	$\mathit{full}(x)$	$\mathit{half}(0)$	$\mathit{half}(1)$	$\times$
$\mathit{full}(y)$	$\times$	$\times$	$\times$	$\times$
$\mathit{half}(0)$	$\times$	$\mathit{full}(0)$	$\times$	$\times$
$\mathit{half}(1)$	$\times$	$\times$	$\mathit{full}(1)$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$
$\mathit{valid}(-)$	*True*	*True*	*True*	*False*

Properties of interest (no $\mathit{full}(-)$, no $\times$):

$$\mathit{valid}(\mathit{half}(x) \cdot \mathit{half}(y)) \;\Rightarrow\; x = y$$

$$\mathit{half}(0) \cdot \mathit{half}(0) \;\rightsquigarrow\; \mathit{half}(1) \cdot \mathit{half}(1)$$

Demo: `half_ra.v`

Exercise: `start_finish.v`

Quiz

Properties we have:

$$\boxed{\mathit{half}(x) \cdot \mathit{half}(y)} \;\Rightarrow\; x = y$$

$$\boxed{\mathit{half}(0)} * \boxed{\mathit{half}(0)} \;\Rrightarrow\; \boxed{\mathit{half}(1)} * \boxed{\mathit{half}(1)}$$

Properties we want:

$$\boxed{Q(0)} * \boxed{A} \;\Rrightarrow\; \boxed{Q(1)} * \boxed{B}$$

$$\boxed{Q'(0)} * \boxed{A'} \;\Rrightarrow\; \boxed{Q'(1)} * \boxed{B'}$$

$$\boxed{Q(x)} * \boxed{B} * \boxed{Q'(y)} * \boxed{B'} \;\Rightarrow\; x = 1 \land y = 1$$

Can we choose: $Q(x) = \mathit{half}(x)$, $A = \mathit{half}(0)$, $B = \mathit{half}(1)$?

Products

Let us call commutative-monoid-with-validity *resource algebra*[0] (RA)

Products

Let us call commutative-monoid-with-validity *resource algebra*[0] (RA)

The **product** of two RA $(A, \cdot_A, \mathit{valid}_A)$ and $(B, \cdot_B, \mathit{valid}_B)$ is defined as $(A \times B, \cdot_{A\times B}, \mathit{valid}_{A\times B})$ where

$$(a, b) \cdot_{A\times B} (a', b') \triangleq (a \cdot_A a', b \cdot_B b')$$
$$\mathit{valid}_{A\times B}((a, b)) \triangleq \mathit{valid}_A(a) \wedge \mathit{valid}_B(b)$$

Products

Let us call commutative-monoid-with-validity *resource algebra*^{0} (RA)

The **product** of two RA A and B is defined as $A \times B$ with

$$(a, b) \cdot (a', b') \triangleq (a \cdot a', b \cdot b')$$
$$valid((a, b)) \triangleq valid(a) \land valid(b)$$

Products

Let us call commutative-monoid-with-validity *resource algebra*[0] (RA)

The **product** of two RA A and B is defined as $A \times B$ with

$$
\begin{aligned}
(a, b) \cdot (a', b') &\triangleq (a \cdot a', b \cdot b') \\
\mathit{valid}((a, b)) &\triangleq \mathit{valid}(a) \wedge \mathit{valid}(b)
\end{aligned}
$$

Property: frame-preserving update is pointwise:

$$
\frac{a \rightsquigarrow a' \qquad b \rightsquigarrow b'}{(a, b) \rightsquigarrow (a', b')}
$$

Option

Option of an RA A, with carrier:

```
option A := None | Some of A
```

and operations:

$- \cdot -$	None	Some(a)
None	None	Some(a)
Some(b)	Some(b)	Some($a \cdot b$)

$\mathit{valid}(-)$	True	$\mathit{valid}(a)$

Option

Option of an RA A, with carrier:

```
option A := None | Some of A
```

and operations:

$- \cdot -$	None	Some(a)
None	None	Some(a)
Some(b)	Some(b)	Some($a \cdot b$)

$\mathit{valid}(-)$	True	$\mathit{valid}(a)$

Properties of frame-preserving update:

$$\frac{a \rightsquigarrow b}{\texttt{Some}(a) \rightsquigarrow \texttt{Some}(b)} \qquad \frac{}{\texttt{Some}(a) \rightsquigarrow \texttt{None}}$$

Option

Option of an RA A, with carrier:

```
option A := None | Some of A
```

and operations:

$- \cdot -$	None	Some(a)
None	None	Some(a)
Some(b)	Some(b)	Some($a \cdot b$)

$\mathit{valid}(-)$	True	$\mathit{valid}(a)$

Properties of frame-preserving update:

$$\frac{a \rightsquigarrow b}{\texttt{Some}(a) \rightsquigarrow \texttt{Some}(b)} \qquad \frac{}{\texttt{Some}(a) \rightsquigarrow \texttt{None}}$$

In the following we write ϵ for the unit None and a for Some(a)

The algebra needed for the example

Let's use the RA : $\texttt{option}\ \mathcal{M}_{half} \times \texttt{option}\ \mathcal{M}_{half}$

$$\begin{aligned} \boxed{A} &= \boxed{half(0), \epsilon} \\ \boxed{B} &= \boxed{half(1), \epsilon} \\ \boxed{Q(x)} &= \boxed{half(x), \epsilon} \end{aligned} \qquad \begin{aligned} \boxed{A'} &= \boxed{\epsilon, half(0)} \\ \boxed{B'} &= \boxed{\epsilon, half(1)} \\ \boxed{Q'(x)} &= \boxed{\epsilon, half(x)} \end{aligned}$$

$\{ \boxed{\mathit{full}(0), \mathit{full}(0)} \}$

$$\{\boxed{\mathit{full}(0), \mathit{full}(0)}\}$$

$$\{\boxed{\mathit{half}(0), \epsilon} * \boxed{\epsilon, \mathit{half}(0)} * \boxed{\mathit{half}(0), \mathit{half}(0)}\}$$

$\{\boxed{\mathit{full}(0), \mathit{full}(0)}\}$

$\{\boxed{\mathit{half}(0), \epsilon} * \boxed{\epsilon, \mathit{half}(0)} * \boxed{\mathit{half}(0), \mathit{half}(0)}\}$

```
let r = ref 0
```

$\{\boxed{\mathit{half}(0), \epsilon} * \boxed{\epsilon, \mathit{half}(0)} * \boxed{\mathit{half}(0), \mathit{half}(0)} * r \mapsto 0\}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{half(0), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(0)}\}$
...	‖	...

$\{ \boxed{full(0), full(0)} \}$

$\{ \boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} \}$

```
let r = ref 0
```

$\{ \boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0 \}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{ \boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} \}$

$\{ \boxed{half(0), \epsilon} \}$	$\{ \boxed{\epsilon, half(0)} \}$
...	...
$\{ \boxed{half(1), \epsilon} \}$	$\{ \boxed{\epsilon, half(1)} \}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{half(0), \epsilon}\}$	$\Vert$	$\{\boxed{\epsilon, half(0)}\}$
...		...
$\{\boxed{half(1), \epsilon}\}$		$\{\boxed{\epsilon, half(1)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)}\}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

	‖	
$\{\boxed{half(0), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(0)}\}$
...	‖	...
$\{\boxed{half(1), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(1)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)}\}$

```
acquire l
```

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * R\}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{half(0), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(0)}\}$
...	‖	...
$\{\boxed{half(1), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(1)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)}\}$

```
acquire l
```

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * R\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)}\}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{half(0), \epsilon}\}$	$\Vert$	$\{\boxed{\epsilon, half(0)}\}$
...		...
$\{\boxed{half(1), \epsilon}\}$		$\{\boxed{\epsilon, half(1)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)}\}$

```
acquire l
```

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * R\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)} * x = 1 * y = 1\}$

$\{ \boxed{full(0), full(0)} \}$

$\{ \boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} \}$

```
let r = ref 0
```

$\{ \boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0 \}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{ \boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} \}$

$\{ \boxed{half(0), \epsilon} \}$	‖	$\{ \boxed{\epsilon, half(0)} \}$
...	‖	...
$\{ \boxed{half(1), \epsilon} \}$	‖	$\{ \boxed{\epsilon, half(1)} \}$

$\{ \boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} \}$

```
acquire l
```

$\{ \boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * R \}$

$\{ \boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)} \}$

$\{ \boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)} * x = 1 * y = 1 \}$

$\{ \boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto 2 * \boxed{half(x), half(y)} * x = 1 * y = 1 \}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{half(0), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(0)}\}$
...	‖	...
$\{\boxed{half(1), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(1)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)}\}$

```
acquire l
```

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * R\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)} * x = 1 * y = 1\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto 2 * \boxed{half(x), half(y)} * x = 1 * y = 1\}$

$\{r \mapsto 2\}$

$\{\boxed{full(0), full(0)}\}$

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)}\}$

```
let r = ref 0
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)} * \boxed{half(0), half(0)} * r \mapsto 0\}$

we choose $R = \exists xy\ r \mapsto x + y * \boxed{half(x), half(y)}$

```
let l = newlock ()
```

$\{\boxed{half(0), \epsilon} * \boxed{\epsilon, half(0)}\}$

$\{\boxed{half(0), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(0)}\}$
...	‖	...
$\{\boxed{half(1), \epsilon}\}$	‖	$\{\boxed{\epsilon, half(1)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)}\}$

```
acquire l
```

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * R\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)}\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto x + y * \boxed{half(x), half(y)} * x = 1 * y = 1\}$

$\{\boxed{half(1), \epsilon} * \boxed{\epsilon, half(1)} * r \mapsto 2 * \boxed{half(x), half(y)} * x = 1 * y = 1\}$

$\{r \mapsto 2\}$

```
assert (!r = 2)
```

Zoom on one thread

$$\{ \boxed{\mathit{half}(0), \epsilon} \}$$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$

Zoom on one thread

$\{ \boxed{half(0), \epsilon} \}$

```
acquire l
```

$\{ R * \boxed{half(0), \epsilon} \}$ introduce x, y

$\{ r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon} \}$ hence $x = 0$ and

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

```
incr r
```

$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

```
incr r
```

$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$ we $\Rrightarrow$ to

$\{r \mapsto 1 + y * \boxed{full(1), half(y)}\}$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

```
incr r
```

$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$ we $\Rrightarrow$ to

$\{r \mapsto 1 + y * \boxed{full(1), half(y)}\}$ split

$\{r \mapsto 1 + y * \boxed{half(1), half(y)} * \boxed{half(1), \epsilon}\}$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$
```
acquire l
```
$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$
```
incr r
```
$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$ we $\Rrightarrow$ to

$\{r \mapsto 1 + y * \boxed{full(1), half(y)}\}$ split

$\{r \mapsto 1 + y * \boxed{half(1), half(y)} * \boxed{half(1), \epsilon}\}$ $\exists$-intro

$\{\exists xy\ r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(1), \epsilon}\}$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

```
incr r
```

$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$ we $\Rrightarrow$ to

$\{r \mapsto 1 + y * \boxed{full(1), half(y)}\}$ split

$\{r \mapsto 1 + y * \boxed{half(1), half(y)} * \boxed{half(1), \epsilon}\}$ $\exists$-intro

$\{\exists xy\ r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(1), \epsilon}\}$

$\{R * \boxed{half(1), \epsilon}\}$

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

```
incr r
```

$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$ we $\Rrightarrow$ to

$\{r \mapsto 1 + y * \boxed{full(1), half(y)}\}$ split

$\{r \mapsto 1 + y * \boxed{half(1), half(y)} * \boxed{half(1), \epsilon}\}$ $\exists$-intro

$\{\exists xy\ r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(1), \epsilon}\}$

$\{R * \boxed{half(1), \epsilon}\}$

```
release R
```

Zoom on one thread

$\{\boxed{half(0), \epsilon}\}$

```
acquire l
```

$\{R * \boxed{half(0), \epsilon}\}$ introduce x, y

$\{r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(0), \epsilon}\}$ hence $x = 0$ and

$\{r \mapsto 0 + y * \boxed{full(0), half(y)}\}$

```
incr r
```

$\{r \mapsto 1 + y * \boxed{full(0), half(y)}\}$ we $\Rrightarrow$ to

$\{r \mapsto 1 + y * \boxed{full(1), half(y)}\}$ split

$\{r \mapsto 1 + y * \boxed{half(1), half(y)} * \boxed{half(1), \epsilon}\}$ $\exists$-intro

$\{\exists xy\ r \mapsto x + y * \boxed{half(x), half(y)} * \boxed{half(1), \epsilon}\}$

$\{R * \boxed{half(1), \epsilon}\}$

```
release R
```

$\{\boxed{half(1), \epsilon}\}$

Demo: `incr2.v`

Modularity?

Fixing the resource algebra once lacks modularity, making it tedious to:

- handle more threads e.g. $(\epsilon, \epsilon, \mathit{half}(0), \epsilon)$
- continue the rest of the program keeping $(\mathit{full}(0), \mathit{full}(0), \mathit{unrelated})$
- reuse a proof, allow custom resource algebras, etc

Modularity?

Fixing the resource algebra once lacks modularity, making it tedious to:

- handle more threads e.g. $(\epsilon, \epsilon, \mathit{half}(0), \epsilon)$
- continue the rest of the program keeping $(\mathit{full}(0), \mathit{full}(0), \mathit{unrelated})$
- reuse a proof, allow custom resource algebras, etc

We need several instances of a given monoid, names for those instances, to allow of several different monoids, ...

Modularity?

Fixing the resource algebra once lacks modularity, making it tedious to:

- handle more threads e.g. $(\epsilon, \epsilon, \mathit{half}(0), \epsilon)$
- continue the rest of the program keeping $(\mathit{full}(0), \mathit{full}(0), \mathit{unrelated})$
- reuse a proof, allow custom resource algebras, etc

We need several instances of a given monoid, names for those instances, to allow of several different monoids, ...

Answer: package all of that into one monoid

RA of functions

If $\mathcal{M}$ is an RA, then $X \to \mathcal{M}$ is an RA for any X:

$$(f \cdot g)(x) \triangleq \lambda x. f(x) \cdot g(x) \qquad \mathit{valid}(f) \triangleq \forall x. \mathit{valid}(f(x)) \qquad \frac{f(x) \rightsquigarrow a}{f \rightsquigarrow f[x := a]}$$

RA of functions

If $\mathcal{M}$ is an RA, then $X \to \mathcal{M}$ is an RA for any X:

$$(f \cdot g)(x) \triangleq \lambda x. f(x) \cdot g(x) \qquad valid(f) \triangleq \forall x. valid(f(x)) \qquad \frac{f(x) \rightsquigarrow a}{f \rightsquigarrow f[x := a]}$$

and so is the set of partial functions $X \rightharpoonup \mathcal{M}$. In case $\mathcal{M}$ has no unit, allows to talk about the singleton partial function:

$$\boxed{g}^{\gamma} \triangleq \boxed{[\gamma := g]}$$

Frame preservation and allocation

Problem: creating the new ghost resource $\boxed{g}^{\gamma}$ is impossible!

Frame preservation and allocation

Problem: creating the new ghost resource $\boxed{g}^{\gamma}$ is impossible!

Because $[\gamma := g]$ could be a potential frame of itself:

$$\begin{array}{r} \mathit{valid}(\emptyset \cdot [\gamma := g]) \\ \neg\mathit{valid}([\gamma := g] \cdot [\gamma := g]) \end{array} \qquad \therefore \qquad \emptyset \not\rightsquigarrow [\gamma := g]$$

Frame preservation and allocation

Problem: creating the new ghost resource $\boxed{g}^{\gamma}$ is impossible!

Because $[\gamma := g]$ could be a potential frame of itself:

$$\begin{array}{c} \mathit{valid}(\emptyset \cdot [\gamma := g]) \\ \neg\mathit{valid}([\gamma := g] \cdot [\gamma := g]) \end{array} \qquad \therefore \qquad \emptyset \not\rightsquigarrow [\gamma := g]$$

In fact $a \rightsquigarrow b$ is redefined as $a \rightsquigarrow \{b\}$. More general definition: $a \rightsquigarrow B$ where $B \subseteq \mathcal{M}$:

$$a \rightsquigarrow B \triangleq \forall c^{?} \in \mathcal{M}^{?}\ \mathit{valid}(a \cdot c^{?}) \Rightarrow \exists b \in B\ \mathit{valid}(b \cdot c^{?})$$

$$\mathcal{M}^{?} \triangleq \bot \uplus \mathcal{M} \qquad a \cdot \bot \triangleq a$$

Frame preservation and allocation

Problem: creating the new ghost resource $\boxed{g}^{\gamma}$ is impossible!

Because $[\gamma := g]$ could be a potential frame of itself:

$$\begin{array}{c} valid(\emptyset \cdot [\gamma := g]) \\ \neg valid([\gamma := g] \cdot [\gamma := g]) \end{array} \qquad \therefore \qquad \emptyset \not\rightsquigarrow [\gamma := g]$$

In fact $a \rightsquigarrow b$ is redefined as $a \rightsquigarrow \{b\}$. More general definition: $a \rightsquigarrow B$ where $B \subseteq \mathcal{M}$:

$$a \rightsquigarrow B \triangleq \forall c^{?} \in \mathcal{M}^{?}\ valid(a \cdot c^{?}) \Rightarrow \exists b \in B\ valid(b \cdot c^{?})$$

$$\mathcal{M}^{?} \triangleq \bot \uplus \mathcal{M} \qquad a \cdot \bot \triangleq a$$

We can now allocate if we have infinite possibilities and the rest of the world $c^{?}$ is finite:

$$\frac{valid(g)}{\emptyset \rightsquigarrow \{[\gamma := g] \mid \gamma \in \mathbb{N}\}} \qquad \frac{valid(g)}{True \Rrightarrow \exists \gamma\ \boxed{g}^{\gamma}}$$

true for $\mathbb{N} \xrightarrow{fin} \mathcal{M}$ but **not** for $\mathbb{N} \rightharpoonup \mathcal{M}$.

Several types of RA

The dependent product of finite partial functions to each $\mathcal{M}_i$ is an RA:

$$\prod_{i \in I} \mathbb{N} \xrightarrow{fin} \mathcal{M}_i \qquad \boxed{g : \mathcal{M}_i}^{\gamma} \triangleq \boxed{\lambda j. \begin{cases} [\gamma := g] & \text{if } i = j \\ \emptyset & \text{otherwise} \end{cases}}$$

Several types of RA

The dependent product of finite partial functions to each $\mathcal{M}_i$ is an RA:

$$\prod_{i \in I} \mathbb{N} \xrightarrow{\textit{fin}} \mathcal{M}_i \qquad\qquad \boxed{g : \mathcal{M}_i}^{\gamma} \triangleq \boxed{\lambda j. \begin{cases} [\gamma := g] & \text{if } i = j \\ \emptyset & \text{otherwise} \end{cases}}$$

The Σ of "`iProp` Σ" does the bookkeeping of saying which i correspond to which $\mathcal{M}_i$.

The command

`Context` ‘$\{$`inG` Σ $\mathcal{M}\}$

ensures that $\mathcal{M}$ is somewhere in the set and

`Context` ‘$\{$`mylibG` $\Sigma\}$

ensures that all of the "$\mathcal{M}$" of `mylib` are there too.

Persistent knowledge

How to state that a reference will not change once set?

```
let check r =
  let x = Atomic.get r in
  let y = Atomic.get r in
  assert (x = 0 || x = y)

let try_set r v =
  Atomic.compare_and_set r 0 v

let r = Atomic.make 0

try_set 3  ||  try_set 5  ||  try_set 7  ||  check ()
```

Persistent knowledge

How to state that a reference will not change once set?

```
let check r =
  let x = Atomic.get r in
  let y = Atomic.get r in
  assert (x = 0 || x = y)

let try_set r v =
  Atomic.compare_and_set r 0 v

let r = Atomic.make 0

try_set 3  ||  try_set 5  ||  try_set 7  ||  check ()
```

Specs for `get`: once the returned value is not 0 then it will never change.

$$\{True\}\ \texttt{get ()}\ \left\{n. n = 0 \lor n \neq 0 \land \Box \boxed{shot(n)}^{\gamma}\right\}$$

$$\left\{\boxed{shot(n)}^{\gamma}\right\}\ \texttt{get ()}\ \{v. v = n\}$$

we also need some resource "*pending*" for before shooting.

Resource Algebras

A *resource algebra* is a resource algebra^0 plus a *core* (Pottier, 2013):

$$|\cdot| : \mathcal{M} \to \mathcal{M}^?$$

Intuition/axioms/properties:

- $|a|$ is a 'duplicable' part of a if it exists
- if a has no 'duplicable' part, then $|a| = \bot$

Resource Algebras

A *resource algebra* is a resource algebra[0] plus a *core* (Pottier, 2013):

$$|\cdot| : \mathcal{M} \to \mathcal{M}^?$$

Intuition/axioms/properties:

- $|a|$ is a 'duplicable' part of a if it exists
- if a has no 'duplicable' part, then $|a| = \bot$
- if $|a| \neq \bot$ then $a = a \cdot |a| = a \cdot |a| \cdot |a| = \ldots$
- if $|a| \neq \bot$ then $||a|| = |a| = |a| \cdot |a|$

Resource Algebras

A *resource algebra* is a resource algebra[0] plus a *core* (Pottier, 2013):

$$|\cdot| : \mathcal{M} \to \mathcal{M}^?$$

Intuition/axioms/properties:

- $|a|$ is a 'duplicable' part of a if it exists
- if a has no 'duplicable' part, then $|a| = \bot$
- if $|a| \neq \bot$ then $a = a \cdot |a| = a \cdot |a| \cdot |a| = \ldots$
- if $|a| \neq \bot$ then $||a|| = |a| = |a| \cdot |a|$
- if there is a unit ϵ then $|a| \neq \bot$ ($|a|$ is at least ϵ)
- no core for $\mathcal{M}_{half}$: $|half(x)| = |full(x)| = |\times| = \bot$

Resource Algebras

A *resource algebra* is a resource algebra^0 plus a *core* (Pottier, 2013):

$$|\cdot| : \mathcal{M} \to \mathcal{M}^?$$

Intuition/axioms/properties:

- $|a|$ is a 'duplicable' part of a if it exists
- if a has no 'duplicable' part, then $|a| = \bot$
- if $|a| \neq \bot$ then $a = a \cdot |a| = a \cdot |a| \cdot |a| = \ldots$
- if $|a| \neq \bot$ then $||a|| = |a| = |a| \cdot |a|$
- if there is a unit ϵ then $|a| \neq \bot$ ($|a|$ is at least ϵ)
- no core for $\mathcal{M}_{half}$: $|half(x)| = |full(x)| = |\times| = \bot$

The persistent modality $\Box$ is defined using $|-|$:

$$[\![\Box P]\!]_\rho(a) \triangleq [\![P]\!]_\rho(|a|)$$

Intuition: $\Box P$ is like $P * P * P * \ldots$ (like $!P$ in linear logic)

One shot RA

$$\mathcal{M}_{oneshot} := pending \mid shot(n) \mid \times$$

One shot RA

$$\mathcal{M}_{oneshot} := pending \mid shot(n) \mid \times$$

Composition is similar to $\mathcal{M}_{half}$. If $m \neq n$:

$\cdot$	$pending$	$shot(n)$	$shot(m)$	$\times$
$pending$	$\times$	$\times$	$\times$	$\times$
$shot(n)$	$\times$	$shot(n)$	$\times$	$\times$
$shot(m)$	$\times$	$\times$	$shot(m)$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$

One shot RA

$$\mathcal{M}_{oneshot} := pending \mid shot(n) \mid \times$$

Composition is similar to $\mathcal{M}_{half}$. If $m \neq n$:

$\cdot$	$pending$	$shot(n)$	$shot(m)$	$\times$
$pending$	$\times$	$\times$	$\times$	$\times$
$shot(n)$	$\times$	$shot(n)$	$\times$	$\times$
$shot(m)$	$\times$	$\times$	$shot(m)$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$

$valid$	$True$	$True$	$True$	$False$

One shot RA

$$\mathcal{M}_{oneshot} := pending \mid shot(n) \mid \times$$

Composition is similar to $\mathcal{M}_{half}$. If $m \neq n$:

$\cdot$	$pending$	$shot(n)$	$shot(m)$	$\times$
$pending$	$\times$	$\times$	$\times$	$\times$
$shot(n)$	$\times$	$shot(n)$	$\times$	$\times$
$shot(m)$	$\times$	$\times$	$shot(m)$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$
$valid$	$True$	$True$	$True$	$False$
$\lvert - \rvert$	$\bot$	$shot(n)$	$shot(m)$	$\bot$

One shot RA

$$\mathcal{M}_{oneshot} := pending \mid shot(n) \mid \times$$

Composition is similar to $\mathcal{M}_{half}$. If $m \neq n$:

$\cdot$	$pending$	$shot(n)$	$shot(m)$	$\times$
$pending$	$\times$	$\times$	$\times$	$\times$
$shot(n)$	$\times$	$shot(n)$	$\times$	$\times$
$shot(m)$	$\times$	$\times$	$shot(m)$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$

$valid$	$True$	$True$	$True$	$False$

$\lvert - \rvert$	$\bot$	$shot(n)$	$shot(m)$	$\bot$

Properties:

$$shot(n) \cdot shot(n) = shot(n) \qquad pending \rightsquigarrow shot(n) \qquad valid(shot(n) \cdot shot(m)) \Rightarrow n = m$$

$\mathcal{M}_{oneshot}$ is derivable from existing RAs

$\mathcal{M}_{oneshot}$ has three ingredients:

- it has two disjoint components
- $shot(-)$ has the *agreement* property

$\mathcal{M}_{oneshot}$ is derivable from existing RAs

$\mathcal{M}_{oneshot}$ has three ingredients:

- it has two disjoint components
- $shot(-)$ has the *agreement* property
- $pending$ is *exclusive* i.e. $\forall c\ \neg valid(pending \cdot c)$ — why is it useful?

$\mathcal{M}_{oneshot}$ is derivable from existing RAs

$\mathcal{M}_{oneshot}$ has three ingredients:

- it has two disjoint components
- $shot(-)$ has the *agreement* property
- *pending* is *exclusive* i.e. $\forall c\ \neg valid(pending \cdot c)$ — why is it useful?

$$anything\ exclusive \rightsquigarrow anything\ valid$$

$\mathcal{M}_{oneshot}$ is derivable from existing RAs

$\mathcal{M}_{oneshot}$ has three ingredients:

- it has two disjoint components
- $\mathit{shot}(-)$ has the *agreement* property
- $\mathit{pending}$ is *exclusive* i.e. $\forall c\ \neg \mathit{valid}(\mathit{pending} \cdot c)$ — why is it useful?

$$\mathit{anything\ exclusive} \rightsquigarrow \mathit{anything\ valid}$$

And indeed we can derive it from the corresponding RAs:

$$\mathcal{M}_{oneshot} \triangleq \mathrm{Ex}(1) +_{\lightning} \mathrm{Ag}(\mathbb{Z})$$

where:

- $\mathrm{Ex}(X)$ is the *exclusive* RA over a set X
- $\mathrm{Ag}(X)$ is the *agreement* RA over a set X
- $\mathcal{M}_1 +_{\lightning} \mathcal{M}_2$ is the *sum* of two RAs $\mathcal{M}_1$ and $\mathcal{M}_2$

About $\square$

Demo: `one_shot.v`

Some remarks:

- you can recover the reference from the invariant — see `one_shot_cancel.v`
- for ghost ownership the $\square$ modality is not strictly necessary since we can duplicate it by hand, but it is convenient to have $\mathit{shot}(n)$ in the persistent context
- $\square$ is very convenient for consise definitions, such as

$$P \Rrightarrow Q \triangleq \square(P \twoheadrightarrow \, \models\!\!\!\Rrightarrow Q)$$

$$\{P\}\, e\, \{Q\} \triangleq \square(P \twoheadrightarrow \mathit{wp}\ e\ Q)$$

Authoritative RA

The authoritative RA over an RA $\mathcal{M}$ is, wheree $a, b \in \mathcal{M}$,

$$\mathrm{Auth}(\mathcal{M}) ::= \bullet a \mid \circ b \mid \bullet\circ(a, b) \mid \times$$

Intuition:

- $\bullet a$ is the unique global authority, or authoritative view
  you need $\bullet a$ to update

Authoritative RA

The authoritative RA over an RA $\mathcal{M}$ is, wheree $a, b \in \mathcal{M}$,

$$\mathrm{Auth}(\mathcal{M}) ::= \bullet a \mid \circ b \mid \bullet\circ(a, b) \mid \times$$

Intuition:

- $\bullet a$ is the unique global authority, or authoritative view
  you need $\bullet a$ to update
- $\circ b$ is a fragment, or fragmental view
  there can be several fragments
  use them to record independent contributions

Authoritative RA

The authoritative RA over an RA $\mathcal{M}$ is, wheree $a, b \in \mathcal{M}$,

$$\mathrm{Auth}(\mathcal{M}) ::= \bullet a \mid \circ b \mid \bullet\!\circ(a, b) \mid \times$$

Intuition:

- $\bullet a$ is the unique global authority, or authoritative view
  you need $\bullet a$ to update
- $\circ b$ is a fragment, or fragmental view
  there can be several fragments
  use them to record independent contributions

Main properties:

$$\frac{valid(a \cdot c)}{\bullet a \cdot \circ b \rightsquigarrow \bullet(a \cdot c) \cdot \circ(b \cdot c)} \qquad valid(\bullet a \cdot \circ b) \Rightarrow b \preccurlyeq a$$

Authoritative RA

Operations:

$\cdot$	$\bullet a$	$\circ b$	$\bullet\circ(a, b)$	$\times$
$\bullet a'$	$\times$	$\bullet\circ(a', b)$	$\times$	$\times$
$\circ b'$	$\bullet\circ(a, b')$	$\circ(b \cdot b')$	$\bullet\circ(a, b \cdot b')$	$\times$
$\bullet\circ(a', b')$	$\times$	$\bullet\circ(a', b \cdot b')$	$\times$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$
$\mathit{valid}(-)$	$\mathit{valid}(a)$	$\mathit{valid}(b)$	$\mathit{valid}(a) \land b \preccurlyeq a$	False
$\lvert - \rvert$	$\bot$	$\circ b$	$\circ b$	$\bot$

we could almost derive it by $\mathrm{Auth}(\mathcal{M}) = \mathrm{Excl}(\mathcal{M})^{?} \times \mathcal{M}$ but we need $\mathit{valid}(\bullet\circ(a, b))$ to also require $b \preccurlyeq a \triangleq \exists c\ a = b \cdot c$.

Example usage of $\mathrm{Auth}(\mathcal{M})$

Using $\mathrm{Auth}((\mathbb{N}, +))$ we can prove that 4 threads doing:

$$e_{incr} \triangleq \texttt{acquire l; incr r; release l}$$

will increment `r` at *least four* times. Under the lock invariant $R = \exists n\ r \mapsto n * \boxed{\bullet n}^{\gamma}$:

$$isLock\ l\ R \vdash \left\{\boxed{\circ 0}^{\gamma}\right\} e_{incr} \left\{\boxed{\circ 1}^{\gamma}\right\}$$
$$isLock\ l\ R \vdash \left\{\boxed{\circ 0}^{\gamma}\right\} (e_{incr} \mid\mid e_{incr} \mid\mid e_{incr} \mid\mid e_{incr}) \left\{\boxed{\circ 4}^{\gamma}\right\}$$
$$R \vdash \left\{\boxed{\circ 4}^{\gamma}\right\} \texttt{!r} \left\{n. n \geq 4\right\}$$

Example usage of $\mathrm{Auth}(\mathcal{M})$

Using $\mathrm{Auth}((\mathbb{N}, +))$ we can prove that 4 threads doing:

$$e_{incr} \triangleq \texttt{acquire l; incr r; release l}$$

will increment `r` at *least four* times. Under the lock invariant $R = \exists n\ r \mapsto n * \boxed{\bullet n}^{\gamma}$:

$$isLock\ l\ R \vdash \left\{ \boxed{\circ 0}^{\gamma} \right\} e_{incr} \left\{ \boxed{\circ 1}^{\gamma} \right\}$$

$$isLock\ l\ R \vdash \left\{ \boxed{\circ 0}^{\gamma} \right\} (e_{incr} \mid\mid e_{incr} \mid\mid e_{incr} \mid\mid e_{incr}) \left\{ \boxed{\circ 4}^{\gamma} \right\}$$

$$R \vdash \left\{ \boxed{\circ 4}^{\gamma} \right\} \texttt{!r} \left\{ n. n \geq 4 \right\}$$

Indeed with $\boxed{\bullet n}^{\gamma} * \boxed{\circ 4}^{\gamma}$ we can only prove $4 \preccurlyeq_{(\mathbb{N},+)} n$ which means $4 \leq n$

Intuitively $\circ 4$ does not prevent "other" $\circ 1$'s from contributing to $\bullet n$

Checking counter monotonicity using $\mathrm{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Checking counter monotonicity using $\mathrm{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Let $\mathbb{N}_{max} = (\mathbb{N}, max)$

Checking counter monotonicity using $\mathrm{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Let $\mathbb{N}_{max} = (\mathbb{N}, max)$

Invariant and specs:

Checking counter monotonicity using $\mathrm{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Let $\mathbb{N}_{max} = (\mathbb{N}, max)$

Invariant and specs:

$$\boxed{\exists n\; r \mapsto n * \left[\bullet(n : \mathbb{N}_{max})\right]^{\gamma}}$$

Checking counter monotonicity using $\text{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Let $\mathbb{N}_{max} = (\mathbb{N}, max)$

Invariant and specs:

$$\boxed{\exists n\ r \mapsto n * \left[\bullet(n : \mathbb{N}_{max})\right]^{\gamma}}$$

$$\left\{\left[\circ n\right]^{\gamma}\right\} \texttt{read ()} \left\{k. k \geq n * \left[\circ k\right]^{\gamma}\right\}$$

Checking counter monotonicity using $\text{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Let $\mathbb{N}_{max} = (\mathbb{N}, max)$

Invariant and specs:

$$\boxed{\exists n\ r \mapsto n * \left[\bullet(n : \mathbb{N}_{max})\right]^{\gamma}}$$

$$\left\{\left[\circ n\right]^{\gamma}\right\} \texttt{read ()} \left\{k.k \geq n * \left[\circ k\right]^{\gamma}\right\}$$

$$\left\{\left[\circ n\right]^{\gamma}\right\} \texttt{incr ()} \left\{\left[\circ(n+1)\right]^{\gamma}\right\}$$

Checking counter monotonicity using $\mathrm{Auth}(\mathbb{N}_{max})$

```
let r = Atomic.make 0
let read () = Atomic.get r
let incr () =
  Atomic.fetch_and_add r 1

let check () =
  let x = read () in
  let y = read () in
  assert (y >= x)

let rec loop f () =
  f (); loop f ()

let () =
  let open Domain in
  let d1 = spawn (loop incr) in
  let d2 = spawn (loop check) in
  join d1; join d2
```

Let $\mathbb{N}_{max} = (\mathbb{N}, max)$

Invariant and specs:

$$\exists n\ r \mapsto n * \boxed{\bullet(n : \mathbb{N}_{max})}^{\gamma}$$

$$\left\{\boxed{\circ n}^{\gamma}\right\} \texttt{read ()} \left\{k. k \geq n * \boxed{\circ k}^{\gamma}\right\}$$

$$\left\{\boxed{\circ n}^{\gamma}\right\} \texttt{incr ()} \left\{\boxed{\circ(n+1)}^{\gamma}\right\}$$

Proof of `check`:

$$\left\{\boxed{\circ 0}^{\gamma}\right\} \quad \texttt{let x = read ()}$$

$$\left\{x \geq 0 * \boxed{\circ x}^{\gamma}\right\} \quad \texttt{let y = read ()}$$

$$\left\{y \geq x * \boxed{\circ y}^{\gamma}\right\} \quad \texttt{assert (y >= x)}$$

Checking counter monotonicity using $\mathrm{Auth}(\mathbb{N}_{max})$

Demo: `monotonic_counter.v`

Fractional RA

Definition:

$$\mathrm{Frac} \triangleq (0,1] \cap \mathbb{Q} \mid \times \qquad \mathit{valid}(q) \triangleq q \neq \times \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq \begin{cases} q + q' & \text{if } q + q' \leq 1 \\ \times & \text{otherwise} \end{cases}$$

Fractional RA

Definition:

$$\mathrm{Frac} \triangleq (0,1] \cap \mathbb{Q} \mid \times \qquad \mathit{valid}(q) \triangleq q \neq \times \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq \begin{cases} q + q' & \text{if } q + q' \leq 1 \\ \times & \text{otherwise} \end{cases}$$

Easier definition:

$$\mathrm{Frac} \triangleq \mathbb{Q}^{+*} \qquad \mathit{valid}(q) \triangleq q \leq 1 \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq q + q'$$

Fractional RA

Definition:

$$\text{Frac} \triangleq (0,1] \cap \mathbb{Q} \mid \times \qquad \mathit{valid}(q) \triangleq q \neq \times \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq \begin{cases} q + q' & \text{if } q + q' \leq 1 \\ \times & \text{otherwise} \end{cases}$$

Easier definition:

$$\text{Frac} \triangleq \mathbb{Q}^{+*} \qquad \mathit{valid}(q) \triangleq q \leq 1 \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq q + q'$$

You still have to be a bit careful, here is a wrong definition:

$$\text{Frac} \triangleq \mathbb{Q} \qquad \mathit{valid}(q) \triangleq 0 < q \leq 1 \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq q + q'$$

Fractional RA

Definition:

$$\mathrm{Frac} \triangleq (0,1] \cap \mathbb{Q} \mid \times \qquad valid(q) \triangleq q \neq \times \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq \begin{cases} q + q' & \text{if } q + q' \leq 1 \\ \times & \text{otherwise} \end{cases}$$

Easier definition:

$$\mathrm{Frac} \triangleq \mathbb{Q}^{+*} \qquad valid(q) \triangleq q \leq 1 \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq q + q'$$

You still have to be a bit careful, here is a wrong definition:

$$\mathrm{Frac} \triangleq \mathbb{Q} \qquad valid(q) \triangleq 0 < q \leq 1 \qquad |q| \triangleq \bot \qquad q \cdot q' \triangleq q + q'$$

For once, updates do not matter, still, you can wonder when $q \rightsquigarrow q'$ holds

Authoritative fractional RA

Derived construction: $\mathrm{FracAuth}(M) \triangleq \mathrm{Auth}((\mathrm{Frac} \times \mathcal{M})^?)$ with notations:

$$\bullet a \triangleq \bullet(1, a) \qquad\qquad \circ_q\, b \triangleq \circ(q, b)$$

Properties:

$$\circ_{q+q'}(b \cdot b') \equiv \circ_q b \cdot \circ_{q'} b' \qquad \frac{valid(a \cdot c)}{\bullet a \cdot \circ_q b \rightsquigarrow \bullet(a \cdot c) \cdot \circ_q(b \cdot c)} \qquad valid(\bullet a \cdot \circ_q b) \;\Rightarrow\; b \preccurlyeq a$$

$$valid(\bullet a \cdot \circ_1 b) \;\Rightarrow\; b = a \qquad \frac{valid(a')}{\bullet a \cdot \circ_1 b \rightsquigarrow \bullet a' \cdot \circ_1 a'}$$

Example usage of $\text{FracAuth}(\mathcal{M})$

Using $\text{FracAuth}((\mathbb{N}, +))$ we can finally prove modularly that k threads doing:

$$e_{incr} \triangleq \texttt{acquire l; incr r; release l}$$

will increment `r` at *exactly* k times. Under the lock invariant $R = \exists n\ r \mapsto n * \boxed{\bullet n}^{\gamma}$:

$$True \Rrightarrow \exists\gamma\ \boxed{\bullet 0}^{\gamma} * \boxed{\circ_{1/4} 0}^{\gamma} * \boxed{\circ_{1/4} 0}^{\gamma} * \boxed{\circ_{1/4} 0}^{\gamma} * \boxed{\circ_{1/4} 0}^{\gamma}$$

$$isLock\ l\ R \vdash \left\{\boxed{\circ_{1/4} 0}^{\gamma}\right\} e_{incr} \left\{\boxed{\circ_{1/4} 1}^{\gamma}\right\}$$

$$isLock\ l\ R \vdash \left\{\boxed{\circ_{1} 0}^{\gamma}\right\} (e_{incr} \mid\mid e_{incr} \mid\mid e_{incr} \mid\mid e_{incr}) \left\{\boxed{\circ_{1} 4}^{\gamma}\right\}$$

$$R \vdash \left\{\boxed{\circ_{1} 4}^{\gamma}\right\} \texttt{!r} \ \{n. n = 4\}$$

Other common uses of Auth

When *Loc* and *Val* are any set (not necessarily RAs), this is a useful RA:

$$\mathrm{Auth}(\mathit{Loc} \xrightharpoonup{\mathit{fin}} \mathrm{Excl}(\mathit{Val}))$$

Other common uses of Auth

When *Loc* and *Val* are any set (not necessarily RAs), this is a useful RA:

$$\mathrm{Auth}(Loc \xrightarrow{fin} \mathrm{Excl}(Val)) \qquad\qquad \ell \mapsto v \triangleq \boxed{\circ[\ell := v]}^{\gamma_{heap}}$$

Other common uses of Auth : heaps

When *Loc* and *Val* are any set (not necessarily RAs), this is a useful RA:

$$\mathrm{Auth}(\mathit{Loc} \xrightarrow{\mathit{fin}} \mathrm{Excl}(\mathit{Val})) \qquad\qquad \ell \mapsto v \triangleq \boxed{\circ[\ell := v]}^{\gamma_{\mathit{heap}}}$$

$\ell \mapsto v$ is derived; threads and invariants own the fragmental view; the wp ties the authoritative view $\boxed{\bullet\sigma}^{\gamma_{\mathit{heap}}}$ to the actual physical steps.

Other common uses of Auth : heaps

When *Loc* and *Val* are any set (not necessarily RAs), this is a useful RA:

$$\mathrm{Auth}(Loc \xrightarrow{fin} \mathrm{Excl}(Val)) \qquad\qquad \ell \mapsto v \triangleq \boxed{\circ[\ell := v]}^{\gamma_{heap}}$$

$\ell \mapsto v$ is derived; threads and invariants own the fragmental view; the wp ties the authoritative view $\boxed{\bullet\sigma}^{\gamma_{heap}}$ to the actual physical steps.

For fractional permissions, uses $\mathrm{View}(A, B)$ which generalizes $\mathrm{Auth}(A)$ to two algebras with a extra binary validity $holds : A \to B \to Prop$:

$$\mathrm{View}(Loc \to Val, Loc \to \mathrm{Frac} \times Val) \qquad\qquad \ell \mapsto_q v \triangleq \boxed{\circ[\ell := (q, v)]}^{\gamma_{heap}}$$

Other common uses of Auth : heaps

When *Loc* and *Val* are any set (not necessarily RAs), this is a useful RA:

$$\mathrm{Auth}(Loc \xrightarrow{fin} \mathrm{Excl}(Val)) \qquad\qquad \ell \mapsto v \triangleq \boxed{\circ[\ell := v]}^{\gamma_{heap}}$$

$\ell \mapsto v$ is derived; threads and invariants own the fragmental view; the wp ties the authoritative view $\boxed{\bullet\sigma}^{\gamma_{heap}}$ to the actual physical steps.

For fractional permissions, uses $\mathrm{View}(A, B)$ which generalizes $\mathrm{Auth}(A)$ to two algebras with a extra binary validity $holds : A \rightarrow B \rightarrow Prop$:

$$\mathrm{View}(Loc \rightarrow Val, Loc \rightarrow \mathrm{Frac} \times Val) \qquad\qquad \ell \mapsto_q v \triangleq \boxed{\circ[\ell := (q, v)]}^{\gamma_{heap}}$$

Singleton type class mechanism not to write γ_{heap}

```
Class gen_heapGpreS (L V : Type) (Sigma : gFunctors) {Countable L} := {
  gen_heapGpreS_heap :: ghost_mapG Sigma L V    [...]

Class gen_heapGS (L V : Type) (Sigma : gFunctors) {Countable L} := GenHeapGS {
  gen_heap_inG :: gen_heapGpreS L V Sigma;
  gen_heap_name : gname;  [...]
```

Other common uses of Auth

Another very interesting resource algebra is:

$$\mathrm{Auth}(\mathbb{N} \xrightarrow{\textit{fin}} \mathrm{Agree}(\mathsf{iProp}))$$

Other common uses of Auth : invariants

Another very interesting resource algebra is:

$$\mathrm{Auth}(\mathbb{N} \xrightarrow{fin} \mathrm{Agree}(\mathsf{iProp})) \qquad \boxed{P}^{\iota} \triangleq \left[\circ[\iota := \mathit{agree}(P)]\right]^{\gamma_{inv}}$$

so invariants are "just" ghost state, known as *named propositions*, for example allocating a new invariant is a ghost update updating the map above.

Other common uses of Auth : invariants

Another very interesting resource algebra is:

$$\mathrm{Auth}(\mathbb{N} \xrightarrow{fin} \mathrm{Agree}(\mathsf{iProp})) \qquad \boxed{P}^{\iota} \triangleq \boxed{\circ[\iota := agree(P)]}^{\gamma_{inv}}$$

so invariants are "just" ghost state, known as *named propositions*, for example allocating a new invariant is a ghost update updating the map above... But now:

- iProp is a predicate over some F(iProp), Σ is a set of functors,
- we have a domain equation for iProp
- we need step indexing, ordered families of equivalences, RA become "camera",
- the functors in Σ are now contractive, ...

Other common uses of Auth : invariants

Another very interesting resource algebra is:

$$\mathrm{Auth}(\mathbb{N} \xrightarrow{fin} \mathrm{Agree}(\blacktriangleright \mathsf{iProp}))) \qquad \boxed{P}^{\iota} \triangleq \boxed{\circ[\iota := agree(next\ P)]}^{\gamma_{inv}}$$

so invariants are "just" ghost state, known as *named propositions*, for example allocating a new invariant is a ghost update updating the map above... But now:

- iProp is a predicate over some F(iProp), Σ is a set of functors,
- we have a domain equation for iProp
- we need step indexing, ordered families of equivalences, RA become "camera",
- the functors in Σ are now contractive, ...

Manipulating invariants — from *Iris from the ground up*

INV-ALLOC

$$\triangleright P \vdash \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

Manipulating invariants — from *Iris from the ground up*

INV-ALLOC

$$\triangleright P \vdash \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-OPEN

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P * \left[\{\iota\}\right]^{\gamma_{\mathrm{DIS}}}}$$

INV-CLOSE

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} * \triangleright P * \left[\{\iota\}\right]^{\gamma_{\mathrm{DIS}}} \;{}^{\mathcal{E}\setminus\{\iota\}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}$$

Manipulating invariants — from *Iris from the ground up*

INV-ALLOC

$$\triangleright P \vdash \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-OPEN

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \;{}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P * \left[\{\iota\}\right]^{\gamma_{\mathrm{DIS}}}}$$

INV-CLOSE

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} * \triangleright P * \left[\{\iota\}\right]^{\gamma_{\mathrm{DIS}}} \;{}^{\mathcal{E}\setminus\{\iota\}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}$$

INV-ACCESS

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True})\right)}$$

Manipulating invariants — from *Iris from the ground up*

INV-ALLOC
$$\triangleright P \vdash \,\dot{\Rrightarrow}_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-OPEN
$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \;\; {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P * \left[\{\iota\}\right]^{\gamma_{\mathrm{DIS}}}}$$

INV-CLOSE
$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} * \triangleright P * \left[\{\iota\}\right]^{\gamma_{\mathrm{DIS}}} \;\; {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}}$$

INV-ACCESS
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\dot{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\dot{\Rrightarrow}^{\mathcal{E}} \mathsf{True})\right)}$$

WP-VUP
$$\dot{\Rrightarrow}_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\; e \left\{v.\, \dot{\Rrightarrow}_{\mathcal{E}} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}}\; e \{\Phi\}$$

WP-ATOMIC
$$\frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}\dot{\Rrightarrow}^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\; e \left\{v.\; {}^{\mathcal{E}_2}\dot{\Rrightarrow}^{\mathcal{E}_1} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\; e \{\Phi\}}$$

Brace yourself

Full definition of world satisfaction, invariants, view shifts, wp

Excerpt from *Iris from the ground up*

$$W \triangleq \exists I : \mathbb{N} \xrightarrow{\text{fin}} \mathsf{iProp}.\ \boxed{\bullet\, \mathsf{ag}(\mathsf{next}(I))}^{\gamma_{\mathrm{INV}}} * \mathop{\Large *}_{\iota \in \mathrm{dom}(I)} \left((\triangleright I(\iota) * \boxed{\{\iota\}}^{\gamma_{\mathrm{DIS}}}) \vee \boxed{\{\iota\}}^{\gamma_{\mathrm{EN}}} \right)$$

(Above, ag and next are implicitly mapped pointwise over I).

$$\boxed{P}^{\iota} \triangleq \boxed{\circ\, [\iota \leftarrow \mathsf{ag}(\mathsf{next}(P))]}^{\gamma_{\mathrm{INV}}}$$

$$\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}^{\uparrow}.\ \boxed{P}^{\iota}$$

$$^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P \triangleq W * \boxed{\mathcal{E}_1}^{\gamma_{\mathrm{EN}}} \mathrel{-\!\!*} \dot{\Rrightarrow} \diamond (W * \boxed{\mathcal{E}_2}^{\gamma_{\mathrm{EN}}} * P)$$

$$P \ {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} Q \triangleq \Box (P \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} Q)$$

$$S(\sigma) \triangleq \boxed{\bullet \left(\sigma : Loc \xrightarrow{\text{fin}} \mathrm{Ex}(Val) \right)}^{\gamma_{\mathrm{HEAP}}}$$

$$\begin{aligned}
\mathsf{wp}^{S}_{\mathcal{E}}\ e\ \{\Phi\} \triangleq\ & (e \in Val \wedge \Rrightarrow_{\mathcal{E}} \Phi(e)) \\
& \vee \Big(e \notin Val \wedge \forall \sigma.\ S(\sigma) \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\emptyset} \ \big(\mathrm{red}(e, \sigma) \\
& \quad \wedge \triangleright \forall e_2, \sigma_2, \vec{e}_f.\ ((e, \sigma) \to_{\mathsf{t}} (e_2, \sigma_2, \vec{e}_f)) \mathrel{-\!\!*} {}^{\emptyset}{\Rrightarrow}^{\mathcal{E}} \\
& \qquad (S(\sigma_2) * \mathsf{wp}^{S}_{\mathcal{E}}\ e_2\ \{\Phi\} * \mathop{\Large *}_{e' \in \vec{e}_f} \mathsf{wp}^{S}_{\top}\ e'\ \{v.\mathsf{True}\})\big)\Big)
\end{aligned}$$

$$\{P\}\ e\ \{\Phi\}^{S}_{\mathcal{E}} \triangleq \Box (P \mathrel{-\!\!*} \mathsf{wp}^{S}_{\mathcal{E}}\ e\ \{\Phi\})$$

Excerpt from *Iris from the ground up*

FUP-MONO

$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q}$$

FUP-INTRO-MASK

$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{\mathsf{True} \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} \mathsf{True}}$$

FUP-TRANS

$${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} P$$

FUP-FRAME

$$Q * {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_{\mathrm{f}}}\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_{\mathrm{f}}} (Q * P)$$

FUP-UPD

$$\dot{\Rrightarrow} P \vdash \Rrightarrow_{\mathcal{E}} P$$

FUP-TIMELESS

$$\frac{\mathsf{timeless}(P)}{\triangleright P \vdash \Rrightarrow_{\mathcal{E}} P}$$

INV-PERSIST

$$\boxed{P}^{\mathcal{N}} \vdash \square\, \boxed{P}^{\mathcal{N}}$$

INV-ALLOC

$$\triangleright P \vdash \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-ACCESS

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E} \setminus \mathcal{N}} \big(\triangleright P * (\triangleright P \twoheadrightarrow {}^{\mathcal{E} \setminus \mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})\big)}$$

Fig. 15. Rules for the fancy update modality and invariants.

A *resource algebra* (RA) is a tuple $(M, \overline{\mathcal{V}} : M \to Prop, |-| : M \to M^?, (\cdot) : M \times M \to M)$ satisfying:

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$

$$\forall a, b.\, a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$

$$\forall a.\, |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$

$$\forall a.\, |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-CORE-IDEM)}$$

$$\forall a, b.\, |a| \in M \wedge a \preccurlyeq b \Rightarrow |b| \in M \wedge |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$

$$\forall a, b.\, \overline{\mathcal{V}}(a \cdot b) \Rightarrow \overline{\mathcal{V}}(a) \qquad \text{(RA-VALID-OP)}$$

$$\text{where} \quad M^? \triangleq M \uplus \{\bot\} \qquad \text{with} \qquad a^? \cdot \bot \triangleq \bot \cdot a^? \triangleq a^?$$

$$a \preccurlyeq b \triangleq \exists c \in M.\, b = a \cdot c \qquad \text{(RA-INCL)}$$

$$a \rightsquigarrow B \triangleq \forall c^? \in M^?.\, \overline{\mathcal{V}}(a \cdot c^?) \Rightarrow \exists b \in B.\, \overline{\mathcal{V}}(b \cdot c^?)$$

$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

A *unital resource algebra* (uRA) is a resource algebra M with an element ε satisfying:

$$\overline{\mathcal{V}}(\varepsilon) \qquad \forall a \in M.\, \varepsilon \cdot a = a \qquad |\varepsilon| = \varepsilon$$

Fig. 3. Resource algebras.

Variants/instances of Iris

Relaxed memory

- Invariants such as $\boxed{lock \mapsto 0 \vee lock \mapsto 1 * \exists n\ r \mapsto n}^{\iota}$ only make sense if there is an instantaneous view of the memory, which is not true in relaxed memory
- for now, axiomatic memory models do not fit Iris, but view-based operational memory models (for e.g. for release-acquire synchronisation) can be made to fit
- single-location invariants $\boxed{\ell \mid I}$ which can provide knowledge + special mechanisms (escrows) to transmit non-persistent resources

Linearizability

Under sequential consistency linearizability can be reasoned about using logically atomic triples:

$$\langle P \rangle \; e \; \langle Q \rangle$$

means: "at the linearization point in the execution of e, the resources in P are atomically consumed to produce the resources in Q"

Liveness?

- Transfinite Iris: ordinal step indices for the *existential property* and termination

	Standard Iris	*Transfinite Iris*
if $\vDash \exists x\ P$ then for some x $\vDash P$	✗	✓
$\triangleright(\exists x\ P) \Leftrightarrow \exists x \triangleright P$	✓	✗
$\triangleright(P * Q) \Leftrightarrow \triangleright P * \triangleright Q$	✓	✗

- Nola: "no later" at invariant opening, replaced with restricted formulas

Iris

$$\frac{\{P * \triangleright R\}\ e\ \{Q * \triangleright R\}}{\{P * \boxed{R}^{\iota}\}\ e\ \{Q\}}$$

Nola

$$\frac{[P * [\![F]\!]]\ e\ [Q * [\![F]\!]] \qquad F \in Fml}{[P * \boxed{F}]\ e\ [Q]}$$

Variants of Iris

- complexity analysis: resources can be time/space credits/receipts,
- type soundness, e.g. rustbelt
- relational separation logics
- session types, channels, distributed systems, cryptographic reasoning
- probabilities, non-determinism
- relaxed memory

Exercise

(1) design a resource algebra such that:

$$\mathit{valid}(\mathit{Start}) \qquad \mathit{Start} \rightsquigarrow \mathit{Finish} \qquad \mathit{Persistent}(\boxed{\mathit{Finish}}^{\gamma})$$

Exercise

(1) design a resource algebra such that:

$$valid(Start) \qquad Start \rightsquigarrow Finish \qquad Persistent(\boxed{Finish}^{\gamma})$$

(2) design a resource algebra such that:

$$valid(r(0)) \qquad \forall n \in \mathbb{N}\ r(n) \equiv t(n) \cdot r(n+1) \qquad \neg valid(t(n) \cdot t(n))$$

motivation: allocate once $\Rrightarrow \exists\gamma\ \boxed{r(0)}^{\gamma}$ to generate an infinitely many tokens $\boxed{t(i)}^{\gamma}$, each will be used to transfer resources through single-location invariants at iteration i of a loop.

Exercise

(1) design a resource algebra such that:

$$valid(Start) \qquad Start \rightsquigarrow Finish \qquad Persistent(\boxed{Finish}^{\gamma})$$

(2) design a resource algebra such that:

$$valid(r(0)) \qquad \forall n \in \mathbb{N}\ r(n) \equiv t(n) \cdot r(n+1) \qquad \neg valid(t(n) \cdot t(n))$$

motivation: allocate once $\Rrightarrow \exists \gamma\ \boxed{r(0)}^{\gamma}$ to generate an infinitely many tokens $\boxed{t(i)}^{\gamma}$, each will be used to transfer resources through single-location invariants at iteration i of a loop.

(3) Steal a reference back from an invariant? See `one_shot_cancel.v` — in general how to make *cancellable invariants*?

Exercise

(1) design a resource algebra such that:

$$valid(Start) \qquad Start \rightsquigarrow Finish \qquad Persistent(\boxed{Finish}^{\gamma})$$

(2) design a resource algebra such that:

$$valid(r(0)) \qquad \forall n \in \mathbb{N}\ r(n) \equiv t(n) \cdot r(n+1) \qquad \neg valid(t(n) \cdot t(n))$$

motivation: allocate once $\Rrightarrow \exists \gamma\ \boxed{r(0)}^{\gamma}$ to generate an infinitely many tokens $\boxed{t(i)}^{\gamma}$, each will be used to transfer resources through single-location invariants at iteration i of a loop.

(3) Steal a reference back from an invariant? See `one_shot_cancel.v` — in general how to make *cancellable invariants*?

(4) For using Iris, five exercises here: https://gitlab.mpi-sws.org/iris/tutorial-popl21