Separation Logic 3/4

Jean-Marie Madiot

Inria Paris

February 17, 2026

Specifications of `iter`

Invariant on **past** elements:

$$\forall f l I \quad \frac{\textsc{Iter-past}\quad \forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}$$

Invariant on **future** elements:

$$\forall f l I \quad \frac{\textsc{Iter-Future}\quad \forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}$$

Application of `iter`

specs:

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past} \qquad \frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

```
let length l =
  let r = ref 0 in
  iter (fun x -> incr r) l;
  !r
```

Exercise: Use rule Iter-past to prove `length`

Exercise: Use rule Iter-future to prove `length`

Proof of length with ITER-PAST

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past}$$

Choose $I\ k \equiv r \mapsto |k|$

$\{\ulcorner\urcorner\}$

```
let r = ref 0
```

$\{r \mapsto 0\}$

$\{I\ []\}$

```
iter (fun x -> incr r) l;
```

$\{I\ l\}$

$\{r \mapsto |l|\}$

```
!r
```

$\{\lambda v.\ \ulcorner v = |l|\urcorner\}$

Proof of length with ITER-PAST

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past}$$

Choose $I\ k \equiv r \mapsto |k|$

$\{\ulcorner\urcorner\}$
```
let r = ref 0
```
$\{r \mapsto 0\}$
$\{I\ []\}$
```
iter (fun x -> incr r) l;
```
$\{I\ l\}$
$\{r \mapsto |l|\}$
```
!r
```
$\{\lambda v.\ \ulcorner v = |l|\urcorner\}$

$\{I\ k\}$
$\{r \mapsto |k|\}$
```
(fun x -> incr r) x
```
$\approx$
```
incr r
```
$\{r \mapsto |k| + 1\}$
$\{r \mapsto |k \& x|\}$
$\{I\ (k \& x)\}$

Proof of length with ITER-FUTURE

$$\frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

Choose $I'\ k \equiv r \mapsto |l| - |k|$

$\{\ulcorner\urcorner\}$

```
let r = ref 0
```

$\{r \mapsto 0\}$

$\{r \mapsto |l| - |l|\}$

$\{I\ l\}$

```
iter (fun x -> incr r) l;
```

$\{I\ []\}$

$\{r \mapsto |l| - |[]|\}$

$\{r \mapsto |l|\}$

```
!r
```

$\{\lambda v.\ \ulcorner v = |l|\urcorner\}$

Proof of length with Iter-future

$$\frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

Choose $I'\ k \equiv r \mapsto |l| - |k|$

$\{\ulcorner\urcorner\}$
`let r = ref 0`
$\{r \mapsto 0\}$
$\{r \mapsto |l| - |l|\}$
$\{I\ l\}$
`iter (fun x -> incr r) l;`
$\{I\ []\}$
$\{r \mapsto |l| - |[]|\}$
$\{r \mapsto |l|\}$
`!r`
$\{\lambda v.\ \ulcorner v = |l|\urcorner\}$

$\{I\ (x :: k)\}$
$\{r \mapsto |l| - |x :: k|\}$
`(fun x -> incr r)` x
$\approx$
`incr r`
$\{r \mapsto |l| - |x :: k| + 1\}$
$\{r \mapsto |l| - |k|\}$
$\{I\ k\}$

Application of `iter`

specs:

$$\textsc{Iter-past}\ \frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}} \qquad \textsc{Iter-Future}\ \frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}$$

Exercise: Give an example where Iter-Past is insufficient. Adapt the premise of the rule to allow your example.

Application of `iter`

specs:

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past} \qquad \frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

Exercise: Give an example where Iter-Past is insufficient. Adapt the premise of the rule to allow your example.

```
iter (fun x -> assert (x < 7)) [1; 2; 3; 4]
```

Application of `iter`

specs:

$$\textsc{Iter-past}\ \frac{\forall x k\ \{I\,k\}\ f\,x\ \{\lambda\_.\ I\,(k\&x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\,l\ \{\lambda\_.\ I\,l\}} \qquad \textsc{Iter-Future}\ \frac{\forall x k\ \{I'\,(x :: k)\}\ f\,x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\,l\ \{\lambda\_.\ I'\,\mathsf{nil}\}}$$

Exercise: Give an example where Iter-Past is insufficient. Adapt the premise of the rule to allow your example.

```
iter (fun x -> assert (x < 7)) [1; 2; 3; 4]
```

Adaptation of premise: $\forall x k.\ x \in l \Rightarrow \{I\,k\}\ f\,x\ \{\lambda\_.\ I\,(k\&x)\}$

Application of `iter`

specs:

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past} \qquad \frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

Exercise: Give an example where Iter-Past is insufficient. Adapt the premise of the rule to allow your example.

```
iter (fun x -> assert (x < 7)) [1; 2; 3; 4]
```

Adaptation of premise: $\forall x k.\ x \in l \Rightarrow \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}$

```
let r = ref 54321 in
iter (fun x -> assert (!r mod 10 = x); r := !r / 10)
```

....what are the possible arguments?

Application of `iter`

specs:

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past} \qquad \frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

Exercise: Give an example where Iter-Past is insufficient. Adapt the premise of the rule to allow your example.

```
iter (fun x -> assert (x < 7)) [1; 2; 3; 4]
```

Adaptation of premise: $\forall x k.\ x \in l \Rightarrow \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}$

```
let r = ref 54321 in
iter (fun x -> assert (!r mod 10 = x); r := !r / 10)
    [1; 2; 3; 4]
```

Application of `iter`

specs:

$$\frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}\ \textsc{Iter-past} \qquad \frac{\forall x k\ \{I'\ (x :: k)\}\ f\ x\ \{\lambda\_.\ I'\ k\}}{\{I'\ l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\ \mathsf{nil}\}}\ \textsc{Iter-Future}$$

Exercise: Give an example where Iter-Past is insufficient. Adapt the premise of the rule to allow your example.

```
iter (fun x -> assert (x < 7)) [1; 2; 3; 4]
```

Adaptation of premise: $\forall x k.\ x \in l \Rightarrow \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}$

```
let r = ref 54321 in
iter (fun x -> assert (!r mod 10 = x); r := !r / 10)
    [1; 2; 3; 4]
```

Adaptation: $\forall x k.\ \mathsf{prefix}(k \& x, l) \Rightarrow \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}$

Proof of ITER-FUTURE

Suppose I', f such that $\forall xk\ \{I'\,(x :: k)\}\ f\,x\ \{\lambda\_.\ I'\,k\}$ $(*)$.

```
let rec iter (f : 'a -> unit) (l : 'a list) =
  match l with
  | [] -> ()
  | x::t -> f x; iter f t
```

We prove $\{I'\,l\}$ `iter` $f\,l\ \{\lambda\_.\ I'\,\mathsf{nil}\}$ by induction on l.

$\{I'\,(x :: t)\}$
$f\,x$ by $(*)$
$\{I'\,t\}$
`iter` $f\,t$ by I.H.,
$\{I'\,\mathsf{nil}\}$

Proof of Iter-future ⇒ Iter-past

$$\frac{\textsc{Iter-Future}\quad \forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}} \Rightarrow \frac{\textsc{Iter-past}\quad \forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $(*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s\ \equiv$

Proof of Iter-future ⇒ Iter-past

$$\frac{\textsc{Iter-Future}\quad \forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}} \Rightarrow \frac{\textsc{Iter-past}\quad \forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $(*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s \equiv \exists k.\ \ulcorner l = k +\!\!+ s \urcorner * I\,k$.

Proof of Iter-future ⇒ Iter-past

$$\frac{\textsc{Iter-Future}\quad \forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}} \Rightarrow \frac{\textsc{Iter-past}\quad \forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $(*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s \equiv \exists k.\ \ulcorner l = k +\!\!+ s \urcorner * I\,k$.

$$\frac{}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}\ \textsc{conseq}$$

Proof of Iter-future ⇒ Iter-past

$$\frac{\textsc{Iter-Future}\quad \forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}} \Rightarrow \frac{\textsc{Iter-past}\quad \forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $\quad (*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s \ \equiv\ \exists k.\ \ulcorner l = k +\!\!+ s \urcorner * I\,k$.

$$\frac{I\,\mathsf{nil} \rhd I'\,l \qquad\qquad I'\,\mathsf{nil} \rhd I\,l}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}\ \textsc{conseq}$$

Proof of Iter-future ⇒ Iter-past

$$\frac{\textsc{Iter-Future}\quad \forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}} \Rightarrow \frac{\textsc{Iter-past}\quad \forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $\quad (*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s \ \equiv\ \exists k.\ \ulcorner l = k +\!\!+ s \urcorner * I\,k$.

$$\frac{I\,\mathsf{nil} \rhd I'\,l \qquad \dfrac{}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}}\ \textsc{It.f.}\ I'\,\mathsf{nil} \rhd I\,l}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}\ \textsc{conseq}$$

Proof of Iter-future ⇒ Iter-past

$$\frac{\text{Iter-Future}\quad \forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}} \Rightarrow \frac{\text{Iter-past}\quad \forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $\quad (*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s \ \equiv\ \exists k.\ \ulcorner l = k +\!\!+ s \urcorner * I\,k$.

$$\frac{I\,\mathsf{nil} \rhd I'\,l \qquad \dfrac{\dfrac{l = k +\!\!+ s \Rightarrow}{\forall x s\ \{I'\,(x :: s)\}\ f\ x\ \{\lambda\_.\ I'\,s\}}\ \forall\text{I};\forall\text{I};\exists\text{L};\text{PROP-L}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}}\ \text{It.f.} \qquad I'\,\mathsf{nil} \rhd I\,l}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}\ \text{CONSEQ}$$

Proof of Iter-future ⇒ Iter-past

$$\frac{\forall x k\ \{I'\,(x :: k)\}\ f\ x\ \{\lambda\_.\ I'\,k\}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}}\ \textsc{Iter-Future} \quad \Rightarrow \quad \frac{\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}\ \textsc{Iter-past}$$

Suppose: f, I, l such that $\forall x k\ \{I\,k\}\ f\ x\ \{\lambda\_.\ I\,(k \& x)\}$ $\quad (*)$.

Prove: $\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}$.

Choose $I'\,s \ \equiv\ \exists k.\ \ulcorner l = k +\!\!+ s \urcorner * I\,k$.

$$\dfrac{I\,\mathsf{nil} \rhd I'\,l \qquad \dfrac{\dfrac{\dfrac{(*) \qquad I\,(k \& x) \rhd I'\,s}{l = k +\!\!+ s \Rightarrow \{I\,k\}\ f\ x\ \{\lambda\_.\ I'\,s\}}\ \textsc{conseq}}{\forall x s\ \{I'\,(x :: s)\}\ f\ x\ \{\lambda\_.\ I'\,s\}}\ \textsc{∀i;∀i;∃l;prop-l}}{\{I'\,l\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I'\,\mathsf{nil}\}}\ \textsc{It.f.} \qquad I'\,\mathsf{nil} \rhd I\,l}{\{I\,\mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\,l\}}\ \textsc{conseq}$$

Other higher-order functions

$$\textsc{Iter-past}\quad \frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil}\}\ \texttt{iter}\ f\ l\ \{\lambda\_.\ I\ l\}}$$

$$\textsc{MIter}\quad \frac{\forall x k\ \{I\ k\}\ f\ x\ \{\lambda\_.\ I\ (k \& x)\}}{\{I\ \mathsf{nil} * p \rightsquigarrow \mathsf{MList}\ l\}\ \texttt{miter}\ f\ p\ \{\lambda\_.\ I\ l * p \rightsquigarrow \mathsf{MList}\ l\}}$$

$$\textsc{Fold}\quad \frac{\forall x i k\ \{J\ i\ k\}\ f\ i\ x\ \{\lambda j.\ J\ j\ (k \& x)\}}{\{J\ a\ \mathsf{nil}\}\ \mathsf{fold}\ f\ a\ l\ \{\lambda b.\ J\ b\ l\}}$$

$$\textsc{Map}\quad \frac{\forall x k k'\ \{J\ k\ k'\}\ f\ x\ \{\lambda x'.\ J\ (k \& x)\ (k' \& x')\}}{\{J\ \mathsf{nil}\ \mathsf{nil}\}\ \mathsf{map}\ f\ l\ \{\lambda l'.\ J\ l\ l'\}}$$

Separating implication

Separating implication or "magic wand"

Recalling separating conjunction:

$$(P_1 * P_2)h \quad \equiv \quad \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge P_1\, h_1 \wedge P_2\, h_2$$

Introducing *separating implication*:

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Separating implication or "magic wand"

Recalling separating conjunction:

$$(P_1 * P_2)h \quad \equiv \quad \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge P_1\, h_1 \wedge P_2\, h_2$$

Introducing *separating implication*:

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Intuitions:

$$(P \mathrel{-\!\!*} Q) * P \quad \rhd \quad Q$$

Separation Logic	Linear Logic
$P * Q$	$P \otimes Q$
$P \mathrel{-\!\!*} Q$	$P \multimap Q$

Rules:

$$\frac{R * P \vdash Q}{R \vdash (P \mathrel{-\!\!*} Q)} \qquad \frac{R_1 \vdash (P \mathrel{-\!\!*} Q) \qquad R_2 \vdash P}{R_1 * R_2 \vdash Q}$$

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$
2. $\ulcorner \mathsf{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \mathsf{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ $\qquad\qquad\qquad\qquad\qquad\qquad$ $\{(1,2)\}$
2. $\ulcorner \mathsf{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \mathsf{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ $\quad\quad$ $\{(1,2)\}$
2. $\ulcorner \mathsf{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ $\quad\quad$ all heaps
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \mathsf{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$ — $\{(2,3)\}$ and any h with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\,h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	$\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$	only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$	$\{(2,3)\}$ and any h with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$	$\varnothing$ and any h with $1 \in \mathsf{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$	
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$	
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$	

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\,h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	$\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$	only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$	$\{(2,3)\}$ and any h with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$	$\varnothing$ and any h with $1 \in \mathsf{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$	any h with $1 \in \mathsf{dom}(h)$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$	
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$	

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\,h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	$\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$	only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$	$\{(2,3)\}$ and any h with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$	$\varnothing$ and any h with $1 \in \mathsf{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$	any h with $1 \in \mathsf{dom}(h)$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$	any h with $1 \in \mathsf{dom}(h)$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$	

Separating implication examples

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\,h_1 \Rightarrow Q(h \uplus h_1)$$

Exercise: Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	$\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$	all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$	only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$	$\{(2,3)\}$ and any h with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$	$\varnothing$ and any h with $1 \in \mathsf{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$	any h with $1 \in \mathsf{dom}(h)$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$	any h with $1 \in \mathsf{dom}(h)$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$	any h with $1 \in \mathsf{dom}(h)$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\mathsf{False}\urcorner$
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\mathsf{False}\urcorner$
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$ ...yes: P and $\ulcorner\urcorner \mathrel{-\!\!*} P$ are equivalent
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$ ...yes: P and $\ulcorner\urcorner \mathrel{-\!\!*} P$ are equivalent
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$ yes: unfold $\rhd$ and obtain approx. (1)
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$ ...yes: P and $\ulcorner\urcorner \mathrel{-\!\!*} P$ are equivalent
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$ yes: unfold $\rhd$ and obtain approx. (1)
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$ yes
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$ ...yes: P and $\ulcorner\urcorner \mathrel{-\!\!*} P$ are equivalent
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$ yes: unfold $\rhd$ and obtain approx. (1)
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$ yes
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$ no, e.g. $P = \ulcorner\urcorner$ and $Q = 1 \mapsto 2$

Higher-order representation predicates

Mlist with a representation predicate as parameter

Mutable list representing $L : \mathsf{list}\, A$ using a parameter $R : A \to \mathsf{Val} \to \texttt{Prop}$

$$
\begin{array}{l}
p \rightsquigarrow \mathsf{listof}\, R\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\
\quad \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
\quad \mid X :: L' \Rightarrow \exists x p'. \quad p \mapsto (x, p') \\
\qquad\qquad\qquad\qquad\qquad\qquad * \; p' \rightsquigarrow \mathsf{listof}\, R\, L' \\
\qquad\qquad\qquad\qquad\qquad\qquad * \; x \rightsquigarrow R\, X
\end{array}
$$

Mlist with a representation predicate as parameter

Mutable list representing $L : \mathsf{list}\, A$ using a parameter $R : A \to \mathsf{Val} \to \texttt{Prop}$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{listof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid X :: L' \Rightarrow \exists x p'. \quad p \mapsto (x, p') \\
& & \qquad * \; p' \rightsquigarrow \mathsf{listof}\, R\, L' \\
& & \qquad * \; x \rightsquigarrow R\, X
\end{array}
$$

Remarks:

- the $x_i \rightsquigarrow R\, X_i$'s are separated by $*$ in $\mathsf{listof}\, R\, [X_1; X_2]$

Mlist with a representation predicate as parameter

Mutable list representing $L : \mathsf{list}\, A$ using a parameter $R : A \to \mathsf{Val} \to \texttt{Prop}$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{listof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid X :: L' \Rightarrow \exists x p'. \quad p \mapsto (x, p') \\
& & \qquad * \; p' \rightsquigarrow \mathsf{listof}\, R\, L' \\
& & \qquad * \; x \rightsquigarrow R\, X
\end{array}
$$

Remarks:

- the $x_i \rightsquigarrow R\, X_i$'s are separated by $*$ in $\mathsf{listof}\, R\, [X_1; X_2]$
- we can compose, e.g. $\mathsf{listof}\, (\mathsf{listof}\, \mathsf{Array})$

Mlist with a representation predicate as parameter

Mutable list representing L : list A using a parameter $R : A \to \mathsf{Val} \to \mathtt{Prop}$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{listof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid X :: L' \Rightarrow \exists x p'. \quad p \mapsto (x, p') \\
& & \qquad * \; p' \rightsquigarrow \mathsf{listof}\, R\, L' \\
& & \qquad * \; x \rightsquigarrow R\, X
\end{array}
$$

Remarks:

- the $x_i \rightsquigarrow R\, X_i$'s are separated by $*$ in $\mathsf{listof}\, R\, [X_1; X_2]$
- we can compose, e.g. $\mathsf{listof}\, (\mathsf{listof}\, \mathsf{Array})$
- $(p \rightsquigarrow \mathsf{MList}\, L) \;\Leftrightarrow\; (p \rightsquigarrow \mathsf{listof}\, \mathsf{Id}\, L)$ with $\mathsf{Id}\, X =$

Mlist with a representation predicate as parameter

Mutable list representing $L : \mathsf{list}\, A$ using a parameter $R : A \to \mathsf{Val} \to \mathtt{Prop}$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{listof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid X :: L' \Rightarrow \exists x p'. \quad p \mapsto (x, p') \\
& & \qquad * \; p' \rightsquigarrow \mathsf{listof}\, R\, L' \\
& & \qquad * \; x \rightsquigarrow R\, X
\end{array}
$$

Remarks:

- the $x_i \rightsquigarrow R\, X_i$'s are separated by $*$ in $\mathsf{listof}\, R\, [X_1; X_2]$
- we can compose, e.g. $\mathsf{listof}\, (\mathsf{listof}\, \mathsf{Array})$
- $(p \rightsquigarrow \mathsf{MList}\, L) \;\Leftrightarrow\; (p \rightsquigarrow \mathsf{listof}\, \mathsf{Id}\, L)$ with $\mathsf{Id}\, X = \lambda x. \ulcorner x = X \urcorner$

Mlist with a representation predicate as parameter

Mutable list representing $L : \mathsf{list}\, A$ using a parameter $R : A \to \mathsf{Val} \to \mathtt{Prop}$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{listof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid X :: L' \Rightarrow \exists x p'. \quad p \mapsto (x, p') \\
& & \qquad * \; p' \rightsquigarrow \mathsf{listof}\, R\, L' \\
& & \qquad * \; x \rightsquigarrow R\, X
\end{array}
$$

Remarks:

- the $x_i \rightsquigarrow R\, X_i$'s are separated by $*$ in $\mathsf{listof}\, R\, [X_1; X_2]$
- we can compose, e.g. $\mathsf{listof}\, (\mathsf{listof}\, \mathsf{Array})$
- $(p \rightsquigarrow \mathsf{MList}\, L) \;\Leftrightarrow\; (p \rightsquigarrow \mathsf{listof}\, \mathsf{Id}\, L)$ with $\mathsf{Id}\, X = \lambda x. \ulcorner x = X \urcorner$
- listof MList not appropriate for aliased sublists

Specs with H.-O. representation predicates

For base values:

$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

Specs with H.-O. representation predicates

For base values:

$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

With a higher-order representation predicate:

$$\{x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Specs with H.-O. representation predicates

For base values:

$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

With a higher-order representation predicate:

$$\{x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \mathsf{uncons}\, p\ \{\lambda (x, p').\ x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}$$

Specs with H.-O. representation predicates

For base values:

$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

With a higher-order representation predicate:

$$\{x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \mathsf{uncons}\, p\ \{\lambda (x, p').\ x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}$$

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\} \quad \texttt{p.hd} \quad \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Specs with H.-O. representation predicates

For base values:

$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

With a higher-order representation predicate:

$$\{x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}\ \mathsf{cons}\, x\, p'\ \{\lambda p.\ p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \mathsf{uncons}\, p\ \{\lambda (x, p').\ x \rightsquigarrow R\, X * p' \rightsquigarrow \mathsf{listof}\, R\, L\}$$

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\quad \texttt{p.hd}\quad \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Last postcondition cannot hold because/thanks to separation.

Accessor spec and H.-O. representation predicate

Incorrect specification for `p.hd`:

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \texttt{p.hd}\ \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Exercise: What is a correct postcondition for `p.hd`?

Accessor spec and H.-O. representation predicate

Incorrect specification for `p.hd`:

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \texttt{p.hd}\ \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Exercise: What is a correct postcondition for `p.hd`?

$$\lambda x.\ x \rightsquigarrow R\, X * (x \rightsquigarrow R\, X \mathrel{-\!\!*} p \rightsquigarrow \mathsf{listof}\, R\, (X :: L))$$

Accessor spec and H.-O. representation predicate

Incorrect specification for `p.hd`:

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \texttt{p.hd}\ \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Exercise: What is a correct postcondition for `p.hd`?

$$\lambda x.\ x \rightsquigarrow R\, X * (x \rightsquigarrow R\, X \mathrel{-\!\!*} p \rightsquigarrow \mathsf{listof}\, R\, (X :: L))$$

Exercise: Give an example program for which this postcondition is insufficient, for example for list of mutable objects. What is a postcondition for `p.hd` that fixes this problem?

Accessor spec and H.-O. representation predicate

Incorrect specification for `p.hd`:

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \texttt{p.hd}\ \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Exercise: What is a correct postcondition for `p.hd`?

$$\lambda x.\ x \rightsquigarrow R\, X * (x \rightsquigarrow R\, X \mathrel{-\!\!*} p \rightsquigarrow \mathsf{listof}\, R\, (X :: L))$$

Exercise: Give an example program for which this postcondition is insufficient, for example for list of mutable objects. What is a postcondition for `p.hd` that fixes this problem?

```
p.hd := 1
```

Accessor spec and H.-O. representation predicate

Incorrect specification for `p.hd`:

$$\{p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}\ \texttt{p.hd}\ \{\lambda x.\ x \rightsquigarrow R\, X * p \rightsquigarrow \mathsf{listof}\, R\, (X :: L)\}$$

Exercise: What is a correct postcondition for `p.hd`?

$$\lambda x.\ x \rightsquigarrow R\, X * (x \rightsquigarrow R\, X \mathrel{-\!\!*} p \rightsquigarrow \mathsf{listof}\, R\, (X :: L))$$

Exercise: Give an example program for which this postcondition is insufficient, for example for list of mutable objects. What is a postcondition for `p.hd` that fixes this problem?

```
p.hd := 1
```

$$\lambda x.\ x \rightsquigarrow R\, X * (\forall Y.\ x \rightsquigarrow R\, Y \mathrel{-\!\!*} p \rightsquigarrow \mathsf{listof}\, R\, (Y :: L))$$

Iterator spec and H.-O. representation predicate

$$\begin{array}{rl} \forall f p l I. & (\forall x k.\ \{I\,(x :: k)\}\ f\,x\ \{\lambda\_.\ I\,k\}) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,l\}\ (\texttt{miter}\,f\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,\mathsf{nil}\} \end{array}$$

What is a problem with this specification?

$$\begin{array}{rl} \forall f p L I. & \big(\forall x X K.\ \{I\,(X :: K) * x \rightsquigarrow R\,X\}\ f\,x\ \{\lambda\_.\ I\,K * x \rightsquigarrow R\,X\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,L\}\ \texttt{miter}\,f\,p\ \{\lambda\_.\ p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,\mathsf{nil}\} \end{array}$$

Iterator spec and H.-O. representation predicate

$$\begin{array}{rl} \forall f p l I. & (\forall x k.\ \{I\,(x :: k)\}\ f\,x\ \{\lambda\_.\ I\,k\}) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,l\}\ (\texttt{miter}\,f\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,\mathsf{nil}\} \end{array}$$

What is a problem with this specification?

$$\begin{array}{rl} \forall f p L I. & \big(\forall x X K.\ \{I\,(X :: K) * x \rightsquigarrow R\,X\}\ f\,x\ \{\lambda\_.\ I\,K * x \rightsquigarrow R\,X\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,L\}\ \texttt{miter}\,f\,p\ \{\lambda\_.\ p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,\mathsf{nil}\} \end{array}$$

Hint: `let p = { hd = ref 0; tl = null } in`

Iterator spec and H.-O. representation predicate

$$\begin{array}{rl} \forall f p l I. & (\forall x k.\ \{I\,(x :: k)\}\ f\,x\ \{\lambda\_.\ I\,k\}) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,l\}\ (\texttt{miter}\,f\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,\mathsf{nil}\} \end{array}$$

What is a problem with this specification?

$$\begin{array}{rl} \forall f p L I. & \big(\forall x X K.\ \{I\,(X :: K) * x \rightsquigarrow R\,X\}\ f\,x\ \{\lambda\_.\ I\,K * x \rightsquigarrow R\,X\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,L\}\ \texttt{miter}\,f\,p\ \{\lambda\_.\ p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,\mathsf{nil}\} \end{array}$$

Hint: `let p = { hd = ref 0; tl = null } in  miter incr p`

Iterator spec and H.-O. representation predicate

$$\begin{array}{rl} \forall f p l I. & (\forall x k.\ \{I\,(x :: k)\}\ f\,x\ \{\lambda\_.\ I\,k\}) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,l\}\ (\texttt{miter}\,f\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,\mathsf{nil}\} \end{array}$$

What is a problem with this specification?

$$\begin{array}{rl} \forall f p L I. & \big(\forall x X K.\ \{I\,(X :: K) * x \rightsquigarrow R\,X\}\ f\,x\ \{\lambda\_.\ I\,K * x \rightsquigarrow R\,X\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,L\}\ \texttt{miter}\,f\,p\ \{\lambda\_.\ p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ I\,\mathsf{nil}\} \end{array}$$

Hint: `let p = { hd = ref 0; tl = null } in  miter incr p`

Exercise: specify the function `miter`, using an invariant of the form $J\,K\,K'$, where K is remaining to process and K' is a result list.

Iterator for listof

$$\forall f p R L J \;\; \frac{\begin{array}{c}\forall x X K K'. \quad \{x \rightsquigarrow R\,X * J\,(X :: K)\,K'\} \\ f\,x \\ \{\lambda\_.\ \exists Y.\ x \rightsquigarrow R\,Y\ *\ J\,K\,(K' \& Y)\}\end{array}}{\begin{array}{c}\{p \rightsquigarrow \mathsf{listof}\,R\,L\ *\ J\,L\,\mathsf{nil}\} \\ (\texttt{miter}\,f\,p) \\ \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{listof}\,R\,L'\ *\ J\,\mathsf{nil}\,L'\}\end{array}}$$

Parallelism and Concurrency

Parallel pairs

A parallel pair, written $(|t_1, t_2|)$, for evaluating two subterms in parallel.
(Note: one often sees $t_1||t_2$ for let $((), ()) = (|t_1, t_2|)$ in $()$.)

Computing: $a[i] + a[i+1] + ... + a[j-1]$.

```
let rec sum a i j =
  if j - i = 1 then a.(i) else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
    s1 + s2
 end
```

Parallel pairs

A parallel pair, written $(|t_1, t_2|)$, for evaluating two subterms in parallel.
(Note: one often sees $t_1||t_2$ for let $((), ()) = (|t_1, t_2|)$ in $()$.)

Computing: $a[i] + a[i+1] + ... + a[j-1]$.

```
let rec sum a i j =
  if j - i = 1 then a.(i) else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
    s1 + s2
 end
```

Generalizable to map-reduce: $f(t[0]) \oplus f(a[1]) \oplus ... \oplus f(a[n-1])$.
(on which condition on $\oplus$?)

Reasoning rule for parallel pairs

$$\frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 * H_2\}\ (|t_1, t_2|)\ \{Q_1 * Q_2\}}\ \textsc{parallel}$$

where $Q_1 * Q_2 \equiv \lambda(x_1, x_2).\ Q_1\, x_1 * Q_2\, x_2$

Reasoning rule for parallel pairs

$$\frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 * H_2\}\ (|t_1, t_2|)\ \{Q_1 * Q_2\}}\ \textsc{parallel}$$

where $Q_1 * Q_2 \equiv \lambda(x_1, x_2).\ Q_1\, x_1 * Q_2\, x_2$

This rule restricts parallel threads to act on disjoint parts of memory.
(No need for non-interference conditions.)

Concurrent locks: example

```
let r = ref 0
let s = ref n
let p = newlock()

let concurrent_step () =
  acquire p;
  incr r;
  decr s;
  release p
```

The intention is that `concurrent_step ()` can be called concurrently by different threads running in parallel.

Concurrent locks: example

```
let r = ref 0
let s = ref n
let p = newlock()

let concurrent_step () =
  acquire p;
  incr r;
  decr s;
  release p
```

The intention is that `concurrent_step ()` can be called concurrently by different threads running in parallel.

Heap predicate $p \rightsquigarrow \mathsf{Lock}\, H$ asserts that lock p protects an invariant H.
Here:

$$p \rightsquigarrow \mathsf{Lock}\, (\exists i.\ (r \mapsto i) * (s \mapsto n - i))$$

Concurrent locks: specification of operations

Representation predicate:

$$p \rightsquigarrow \mathsf{Lock}\, H$$

It is duplicable, i.e.:

$$p \rightsquigarrow \mathsf{Lock}\, H \quad \rhd \quad p \rightsquigarrow \mathsf{Lock}\, H \;*\; p \rightsquigarrow \mathsf{Lock}\, H$$

Operations:

$$\forall H. \qquad \{H\}\ (\texttt{newlock}\ ())\ \{\lambda p.\ p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$\forall p H. \qquad \{p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{acquire}\ \texttt{p})\ \{\lambda \_.\ H * p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$\forall p H.\ \{H * p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{release}\ \texttt{p})\ \{\lambda \_.\ p \rightsquigarrow \mathsf{Lock}\, H\}$$

Concurrent locks: exercise

$$\forall H. \quad \{H\}\ (\texttt{newlock}\ ()\ )\ \{\lambda p.\ p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$\forall pH. \quad \{p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{acquire p})\ \{\lambda\_.\ H * p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$\forall pH. \{H * p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{release p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$H_0 \equiv p \rightsquigarrow \mathsf{Lock}\, (\exists i.\ (r \mapsto i) * (s \mapsto n - i))$$

Exercise: States before/after declarations; specify and prove the function.

```
let r = ref 0
let s = ref n
let p = newlock ()

let concurrent_step () =
   acquire p;
   incr r; decr s;
   release p
```

Concurrent locks: exercise

Exercise: States before/after declarations; specify and prove the function.

```
let r = ref 0
let s = ref n
let p = newlock ()

let concurrent_step () =
   acquire p;
   incr r; decr s;
   release p
```

1: $\ulcorner\urcorner$. 2: $r \mapsto 0$. 3: $r \mapsto 0 * s \mapsto n$.
4: $p \rightsquigarrow \mathsf{Lock}\,(\exists i.\ (r \mapsto i) * (s \mapsto n - i))$.
7: $(r \mapsto i) * (s \mapsto n - i)$. 8: $(r \mapsto i + 1) * (s \mapsto n - i)$.
9: $(r \mapsto i + 1) * (s \mapsto n - i - 1)$. Instantiate the invariant with $i + 1$.

Fractional permissions

Fractional permissions (Boyland, 2003)

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with } 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) \;*\; (r \overset{1/2}{\mapsto} v)$$

More generally, if $0 < \alpha, \beta \leqslant 1$:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} v)$$

Fractional permissions (Boyland, 2003)

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with } 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) \;*\; (r \overset{1/2}{\mapsto} v)$$

More generally, if $0 < \alpha, \beta \leqslant 1$:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} v)$$

Values must agree: if $0 < \alpha, \beta \leqslant 1$:

$$\Big((r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} w)\Big) \;\rhd\; \Big((r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} w) * \ulcorner v = w \urcorner\Big)$$

Fractional permissions (Boyland, 2003)

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with } 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) * (r \overset{1/2}{\mapsto} v)$$

More generally, if $0 < \alpha, \beta \leqslant 1$:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) * (r \overset{\beta}{\mapsto} w)$$

Values must agree: if $0 < \alpha, \beta \leqslant 1$:

$$\Big((r \overset{\alpha}{\mapsto} v) * (r \overset{\beta}{\mapsto} w)\Big) \;\rhd\; \Big((r \overset{\alpha}{\mapsto} v) * (r \overset{\beta}{\mapsto} w) * \ulcorner v = w \urcorner\Big)$$

Operations:

$$\{\ulcorner\urcorner\}\ \texttt{(ref v)}\ \{\lambda r.\ r \overset{1}{\mapsto} v\}$$

$$\{r \overset{1}{\mapsto} v'\}\ \texttt{(r := v)}\ \{\lambda\_.\ r \overset{1}{\mapsto} v\}$$

$$\forall \alpha.\ \ \alpha > 0 \Rightarrow\ \ \{r \overset{\alpha}{\mapsto} v\} \quad \texttt{(!r)} \quad \{\lambda x.\ \ulcorner x = v \urcorner * (r \overset{\alpha}{\mapsto} v)\}$$

Concurrent locks: can we prove this program?

```
let r = ref 0
let p = newlock()

let f () =
  acquire p;
  incr r;
  release p

let () =
  let _ = (| f(), f() |) in
  acquire p;
  assert (!r == 2)
```

Attempt to prove `incr2`

```
let r = ref 0
let p = newlock()
let f () = acquire p; incr r; release p
let () = let _ = (| f(), f() |) in acquire p; assert (!r == 2)
```

rewritten graphically as:

```
let r = ref 0
let p = newlock()

acquire p;          ||    acquire p;
r := !r + 1;        ||    r := !r + 1;
release p;          ||    release p;

acquire p;
assert (!r == 2);
```

Attempt to prove `incr2`

```
let r = ref 0
let p = newlock()
let f () = acquire p; incr r; release p
let () = let _ = (| f(), f() |) in acquire p; assert (!r == 2)
```

rewritten graphically as:

```
let r = ref 0
let p = newlock()
```

```
acquire p;          ||    acquire p;
r := !r + 1;        ||    r := !r + 1;
release p;          ||    release p;
```

```
acquire p;
assert (!r == 2);
```

$$p \rightsquigarrow \mathsf{Lock}\,(???)$$

Attempt to prove `incr2`

```
let r = ref 0
let p = newlock()
let f () = acquire p; incr r; release p
let () = let _ = (| f(), f() |) in acquire p; assert (!r == 2)
```

rewritten graphically as:

```
let r = ref 0
let p = newlock()
```

```
acquire p;          ||    acquire p;
r := !r + 1;        ||    r := !r + 1;
release p;          ||    release p;
```

```
acquire p;
assert (!r == 2);
```

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n.\ r \mapsto n)$$

Attempt to prove `incr2`

```
let r = ref 0
let p = newlock()
let f () = acquire p; incr r; release p
let () = let _ = (| f(), f() |) in acquire p; assert (!r == 2)
```

rewritten graphically as:

```
let r = ref 0
let p = newlock()
```

```
acquire p;          ||    acquire p;
r := !r + 1;        ||    r := !r + 1;
release p;          ||    release p;
```

```
acquire p;
assert (!r == 2);
```

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n.\ r \mapsto n * \ulcorner \ \ldots ? \ldots \ \urcorner)$$

Attempt to prove `incr2`

```
let r = ref 0
let p = newlock()
let f () = acquire p; incr r; release p
let () = let _ = (| f(), f() |) in acquire p; assert (!r == 2)
```

rewritten graphically as:

```
let r = ref 0
let p = newlock()
```

```
acquire p;          ||    acquire p;
r := !r + 1;        ||    r := !r + 1;
release p;          ||    release p;
```

```
acquire p;
assert (!r == 2);
```

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n.\ r \mapsto n * \ulcorner \ \ldots ? \ \ldots \ \urcorner)$$

Problem: it is impossible to prove, only with invariants, that this program does not crash (i.e. to prove $\{\ulcorner\mathsf{True}\urcorner\}$ program $\{\ulcorner\mathsf{True}\urcorner\}$).

Second attempt with auxiliary variables

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = newlock()
```

```
acquire p;                 ||    acquire p;
r := !r + 1;               ||    r := !r + 1;
r1 := !r1 + 1;             ||    r2 := !r2 + 1;
release p;                 ||    release p;
```

```
acquire p;
assert (!r == 2);
```

Second attempt with auxiliary variables

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = newlock()
```

```
acquire p;          ||    acquire p;
r := !r + 1;        ||    r := !r + 1;
r1 := !r1 + 1;      ||    r2 := !r2 + 1;
release p;          ||    release p;
```

```
acquire p;
assert (!r == 2);
```

Exercise: Find a lock invariant and then prove {True} program {True}.

Hint: you can define a (temporary!) notation $\ell \hookrightarrow v \ \equiv \ \ell \overset{1/2}{\mapsto} v$.

Second attempt with auxiliary variables

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = newlock()
```

```
acquire p;          ||   acquire p;
r := !r + 1;        ||   r := !r + 1;
r1 := !r1 + 1;      ||   r2 := !r2 + 1;
release p;          ||   release p;
```

```
acquire p;
assert (!r == 2);
```

Exercise: Find a lock invariant and then prove {True} program {True}.

Hint: you can define a (temporary!) notation $\ell \hookrightarrow v \ \equiv \ \ell \overset{1/2}{\mapsto} v$.

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2)$$

Proof

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto (0+0) * r_1 \mapsto 0 * r_2 \mapsto 0\}$

Proof

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto (0+0) * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

Proof

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto (0+0) * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

```
let p = newlock()
```

Proof

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```
$\{r \mapsto (0+0) * r_1 \mapsto 0 * r_2 \mapsto 0\}$
$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$
```
let p = newlock()
```
$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

Proof

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```
$\{r \mapsto (0+0) * r_1 \mapsto 0 * r_2 \mapsto 0\}$
$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$
```
let p = newlock()
```
$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$
$\{(p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0) * (p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0)\}$

Proof

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```
$\{r \mapsto (0+0) * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

```
let p = newlock()
```
$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

$\{(p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0) * (p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0)\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$	$\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0\}$
`acquire p;`	`acquire p;`
`r := !r + 1;`	`r := !r + 1;`
`...`	`...`

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto (1 + n_2) * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto (1 + n_2) * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * H\}$ (choosing $n_1 = 1$ and $n_2 = n_2$)

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto (1 + n_2) * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * H\}$ (choosing $n_1 = 1$ and $n_2 = n_2$)

```
release p
```

Left thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some n_1, n_2:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto (n_1 + n_2 + 1) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

so by agreement on r_1 we know $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto (0 + n_2 + 1) * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto (1 + n_2) * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * H\}$ (choosing $n_1 = 1$ and $n_2 = n_2$)

```
release p
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1\}$

Right thread

$$H \quad \equiv \quad \exists n_1, n_2.\ r \mapsto (n_1 + n_2) * (r_1 \overset{1/2}{\mapsto} n_1) * (r_2 \overset{1/2}{\mapsto} n_2)$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0\}$

```
acquire p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0 * H\}$

```
r := !r + 1;
r2 := !r2 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 1 * H\}$

```
release p
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 1\}$

Finish up

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = newlock()
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$ ‖ $\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0\}$

```
acquire p;                    ‖   acquire p;
r := !r + 1;                  ‖   r := !r + 1;
r1 := !r1 + 1;                ‖   r2 := !r2 + 1;
release p;                    ‖   release p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1\}$ ‖ $\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 1\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1\}$

```
acquire p;
```

$\{r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1 * r \mapsto (n_1 + n_2) * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1 * r \mapsto 1 + 1 * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1\}$

$\{r_1 \mapsto 1 * r_2 \mapsto 1 * r \mapsto 2\}$

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

Some remarks

Problems with this proof:

- same example with an arbitrary number of threads?
- need to prove adding instructions preserves the semantics? (erasure theorem, complexify the instrumentation of the code)
- that's reasoning, it *should not* be in the program

We will see other ways, though similar in spirit and complexity.

What are invariants lacking?

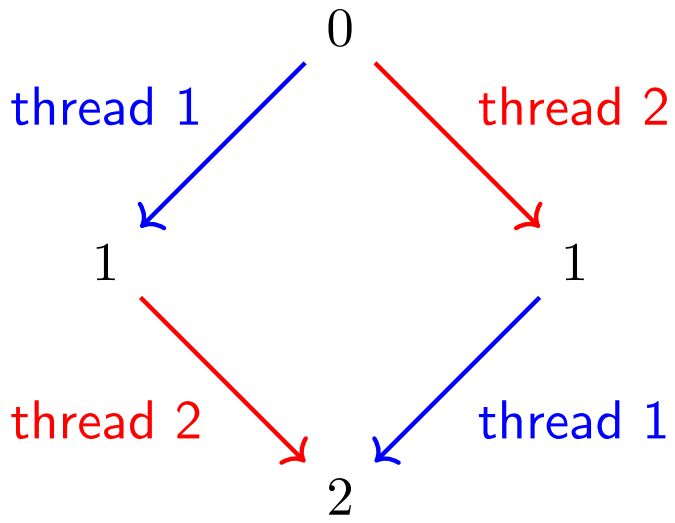
At high-level the program has two interleavings:

```
    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:

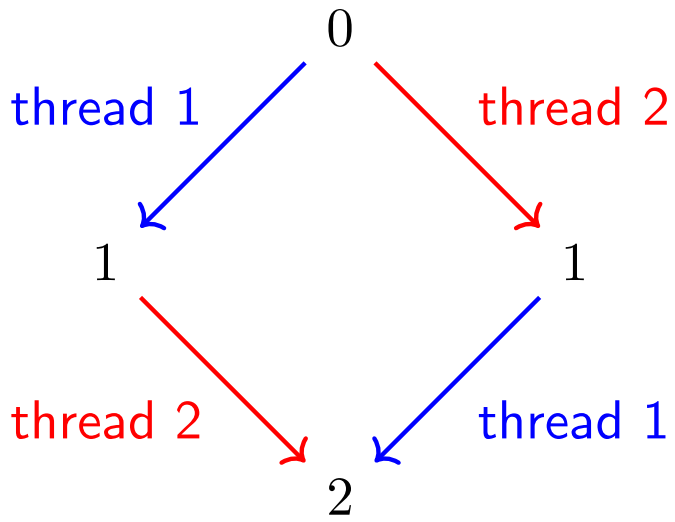
```
    let r = ref 0 in
    let l = newlock () in
acquire l;    || acquire l;
r := !r + 1; || r := !r + 1;
release  l    || release l
    acquire l;
    assert (!r = 2)
```



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)
```

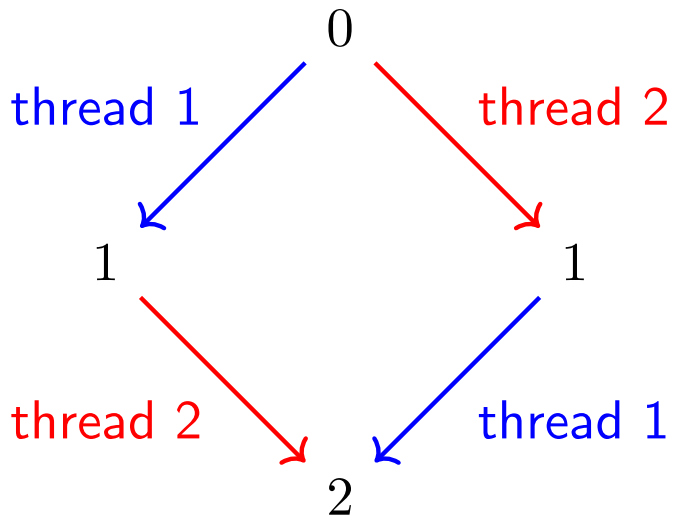


A proof would need need to reflect this somehow.

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
- *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

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = newlock()
```

```
acquire p;                  ||    acquire p;
r := !r + 1;                ||    r := !r + 1;
r1 := !r1 + 1;              ||    r2 := !r2 + 1;
release p;                  ||    release p;
```

```
acquire p;
assert (!r == 2);
```

Ghost state

What is ghost state?

In parallel programs $p = e_1||\ldots||e_n$, reductions $(p, h) \rightarrow (p', h')$ can be from any of the e_i, so $\rightarrow$ is non-deterministic.

What is ghost state?

In parallel programs $p = e_1||\ldots||e_n$, reductions $(p, h) \rightarrow (p', h')$ can be from any of the e_i, so $\rightarrow$ is non-deterministic.

Correctness of non-deterministic programs must consider **all** possible physical steps $(p, h) \rightarrow (p', h')$.

What is ghost state?

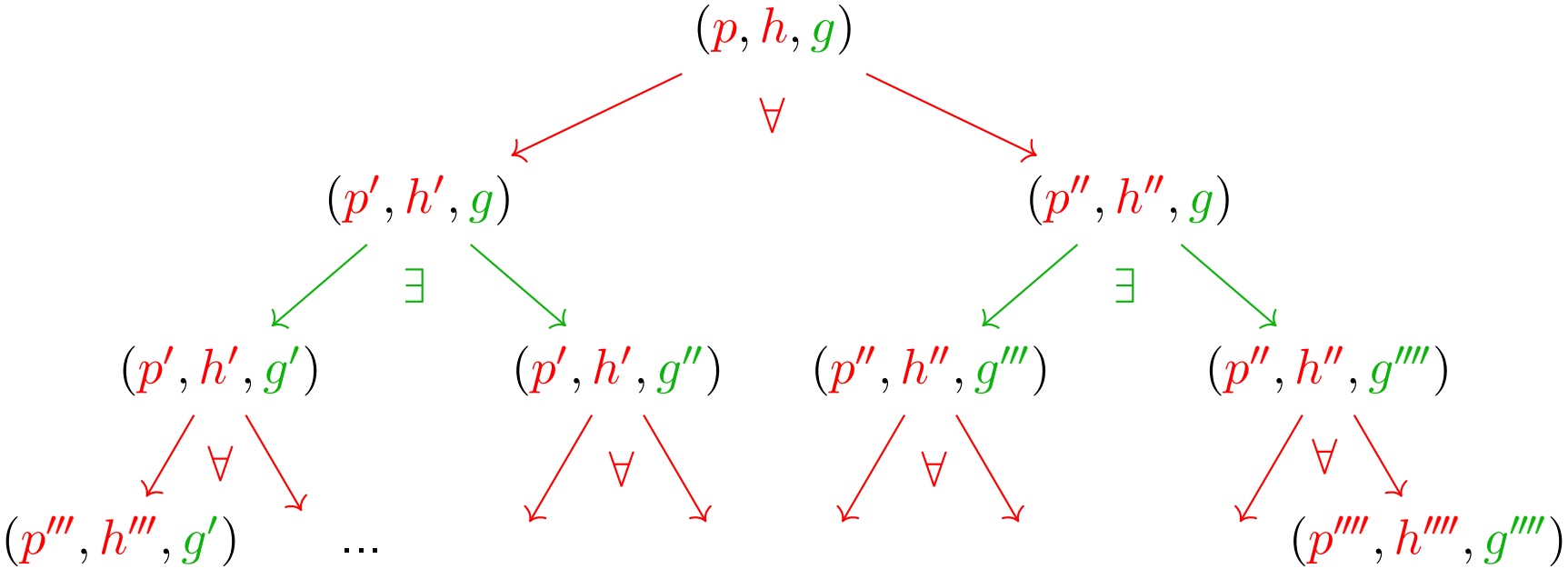
In parallel programs $p = e_1||\ldots||e_n$, reductions $(p, h) \to (p', h')$ can be from any of the e_i, so $\to$ is non-deterministic.

Correctness of non-deterministic programs must consider **all** possible physical steps $(p, h) \to (p', h')$. For each, we get to **choose** a ghost update $g \to g'$:

What is ghost state?

In parallel programs $p = e_1||\ldots||e_n$, reductions $(p, h) \to (p', h')$ can be from any of the e_i, so $\to$ is non-deterministic.

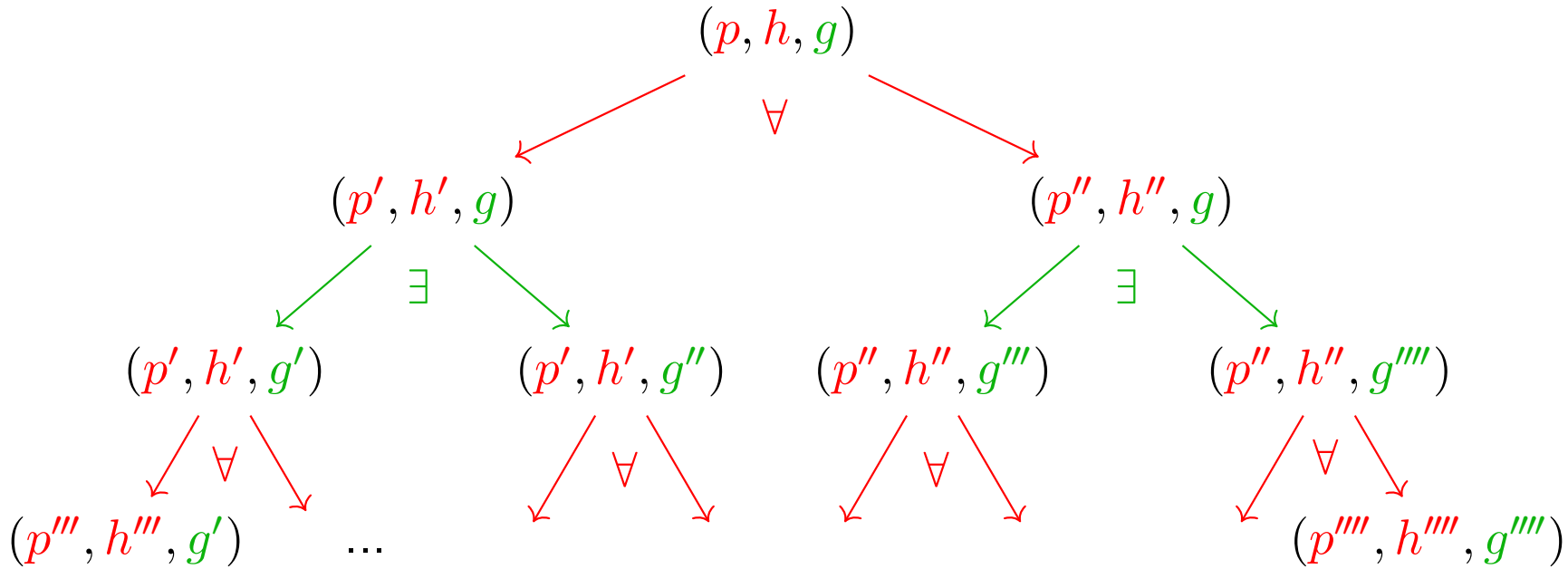
Correctness of non-deterministic programs must consider **all** possible physical steps $(p, h) \to (p', h')$. For each, we get to **choose** a ghost update $g \to g'$:



What is ghost state?

In parallel programs $p = e_1 || \ldots || e_n$, reductions $(p, h) \to (p', h')$ can be from any of the e_i, so $\to$ is non-deterministic.

Correctness of non-deterministic programs must consider **all** possible physical steps $(p, h) \to (p', h')$. For each, we get to **choose** a ghost update $g \to g'$:



Ghost state also needs to be split $g = g_1 \cdot \ldots \cdot g_n$, one part for each e_i.

Ghost state, ghost assertions

Let $(\mathcal{M}, \cdot)$ be a commutative *semigroup* (or "monoid"), i.e. $\forall abc \in \mathcal{M}$:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad\qquad a \cdot b = b \cdot a$$

Ghost state, ghost assertions

Let $(\mathcal{M}, \cdot)$ be a commutative *semigroup* (or "monoid"), i.e. $\forall abc \in \mathcal{M}$:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad\qquad a \cdot b = b \cdot a$$

$a \in \mathcal{M}$ is called a *resource*. At any given time:

- each thread t_i "owns" a resource $a_i \in \mathcal{M}$,

Ghost state, ghost assertions

Let $(\mathcal{M}, \cdot)$ be a commutative *semigroup* (or "monoid"), i.e. $\forall abc \in \mathcal{M}$:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad\qquad a \cdot b = b \cdot a$$

$a \in \mathcal{M}$ is called a *resource*. At any given time:

- each thread t_i "owns" a resource $a_i \in \mathcal{M}$,
- each unopened lock "owns" a resource $r_j \in \mathcal{M}$ that satisfies its R_j

Ghost state, ghost assertions

Let $(\mathcal{M}, \cdot)$ be a commutative *semigroup* (or "monoid"), i.e. $\forall abc \in \mathcal{M}$:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad\qquad a \cdot b = b \cdot a$$

$a \in \mathcal{M}$ is called a *resource*. At any given time:

- each thread t_i "owns" a resource $a_i \in \mathcal{M}$,
- each unopened lock "owns" a resource $r_j \in \mathcal{M}$ that satisfies its R_j
- the global resource $a_1 \cdot \ldots \cdot a_n \cdot r_1 \cdot \ldots \cdot r_k$ is the global ghost state

Ghost state, ghost assertions

Let $(\mathcal{M}, \cdot)$ be a commutative *semigroup* (or "monoid"), i.e. $\forall abc \in \mathcal{M}$:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad\qquad a \cdot b = b \cdot a$$

$a \in \mathcal{M}$ is called a *resource*. At any given time:

- each thread t_i "owns" a resource $a_i \in \mathcal{M}$,
- each unopened lock "owns" a resource $r_j \in \mathcal{M}$ that satisfies its R_j
- the global resource $a_1 \cdot \ldots \cdot a_n \cdot r_1 \cdot \ldots \cdot r_k$ is the global ghost state

Ownership of a is written $\boxed{a}$. Composition maps to separation:

$$\boxed{a \cdot b} = \boxed{a} * \boxed{b}$$

Ghost updates

The prover can perform some "ghost updates" inside each thread:

$$\boxed{a} \Rrightarrow \boxed{a'} \qquad \approx \text{changing } a \cdot a_2 \cdot a_3 \cdot \ldots \text{ to } a' \cdot a_2 \cdot a_3 \cdot \ldots$$

Ghost updates

The prover can perform some "ghost updates" inside each thread:

$$\boxed{a} \Rrightarrow \boxed{a'} \qquad \approx \text{changing } a \cdot a_2 \cdot a_3 \cdot \ldots \text{ to } a' \cdot a_2 \cdot a_3 \cdot \ldots$$

The ghost state can be updated before or after physical steps:

$$\textsc{Ghost-update}\quad \frac{P \Rrightarrow P' \qquad \{P'\}\ c\ \{\lambda x.\ Q'\} \qquad \forall x.\ Q'\,x \Rrightarrow Q\,x}{\{P\}\ c\ \{\lambda x.\ Q\}}$$

but $P \Rrightarrow P'$ does **not** imply $P \rhd P'$

Ghost updates

The prover can perform some "ghost updates" inside each thread:

$$\boxed{a} \Rrightarrow \boxed{a'} \qquad \approx \text{changing } a \cdot a_2 \cdot a_3 \cdot \ldots \text{ to } a' \cdot a_2 \cdot a_3 \cdot \ldots$$

The ghost state can be updated before or after physical steps:

$$\textsc{Ghost-update}\quad \frac{P \Rrightarrow P' \qquad \{P'\}\, c\, \{\lambda x.\, Q'\} \qquad \forall x.\, Q'\, x \Rrightarrow Q\, x}{\{P\}\, c\, \{\lambda x.\, Q\}}$$

but $P \Rrightarrow P'$ does **not** imply $P \rhd P'$

But to be useful, we must deduce *something* from $\boxed{a}$, knowing that other threads can update their own resources.

any idea how?

Ghost state to be continued