Separation Logic 4/4

Jean-Marie Madiot

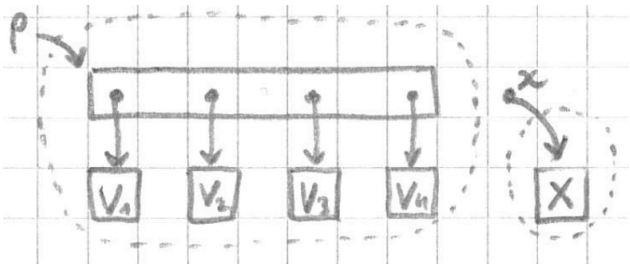
Inria Paris

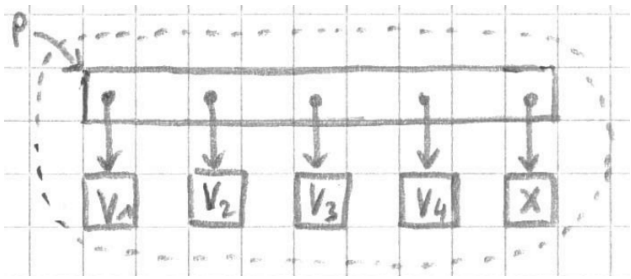
February 19, 2025

part 4/4 : some material from Arthur Charguéraud

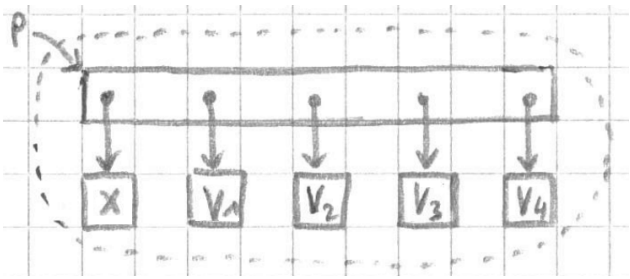
Ownership transfer with a queue of mutable items

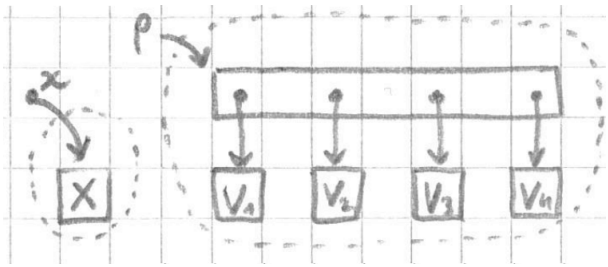
Push:





Pop:





Specification of queues of basic items

$$\{\ulcorner\urcorner\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queue}\,\mathsf{nil}\}$$

$$\forall x p L\ \{p \rightsquigarrow \mathsf{Queue}\,L\}\ (\texttt{push x p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queue}\,(L \& x)\}$$

$$\forall x p L\ \{p \rightsquigarrow \mathsf{Queue}\,(x :: L)\}\ (\texttt{pop p})\ \{\lambda r.\ \ulcorner r = x\urcorner * p \rightsquigarrow \mathsf{Queue}\,L\}$$

$$\forall p p' L L'\ \{p \rightsquigarrow \mathsf{Queue}\,L\ *\ p' \rightsquigarrow \mathsf{Queue}\,L'\}\ (\texttt{concat p p'})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queue}\,(L +\!\!+ L')\}$$

Specification of queues of mutable items

Exercise: specify functions over queues using a higher-order representation predicate written $p \rightsquigarrow \mathsf{Queueof}\, R\, L$.
Shorthand: just write "Q R" instead of "Queueof R".

Specification of queues of mutable items

Exercise: specify functions over queues using a higher-order representation predicate written $p \rightsquigarrow \mathsf{Queueof}\, R\, L$.
Shorthand: just write "Q R" instead of "Queueof R".

$$\{\ulcorner\urcorner\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\}$$

Specification of queues of mutable items

Exercise: specify functions over queues using a higher-order representation predicate written $p \rightsquigarrow \mathsf{Queueof}\, R\, L$.
Shorthand: just write "Q R" instead of "Queueof R".

$$\{\ulcorner\urcorner\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\}$$

$$\forall R p L x X \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\ *\ x \rightsquigarrow R\, X\}\ (\texttt{push}\, x\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \& X)\}$$

Specification of queues of mutable items

Exercise: specify functions over queues using a higher-order representation predicate written $p \rightsquigarrow \mathsf{Queueof}\, R\, L$.
Shorthand: just write "Q R" instead of "Queueof R".

$$\{\ulcorner\urcorner\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\}$$

$$\forall R p L x X \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\ *\ x \rightsquigarrow R\, X\}\ (\texttt{push}\, x\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \& X)\}$$

$$\forall R p L X \{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L)\}\ (\texttt{pop}\, p)\ \{\lambda x.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L\ *\ x \rightsquigarrow R\, X\}$$

Specification of queues of mutable items

Exercise: specify functions over queues using a higher-order representation predicate written $p \rightsquigarrow \mathsf{Queueof}\, R\, L$.
Shorthand: just write "Q R" instead of "Queueof R".

$$\{\ulcorner\urcorner\}\ (\texttt{create()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\}$$

$$\forall R p L x X \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\ *\ x \rightsquigarrow R\, X\}\ (\texttt{push}\, x\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \& X)\}$$

$$\forall R p L X \{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L)\}\ (\texttt{pop}\, p)\ \{\lambda x.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L\ *\ x \rightsquigarrow R\, X\}$$

$$\forall R p p' L L' \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\ *\ p' \rightsquigarrow \mathsf{Queueof}\, R\, L'\}\ (\texttt{concat}\, p\, p')\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L +\!\!+ L')\}$$

The copy problem

Incorrect specification for `copy`:

$$\{p \rightsquigarrow \mathsf{Queueof}\, R\, L\}$$
$$(\mathsf{copy}\, p)$$
$$\{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \ * \ p' \rightsquigarrow \mathsf{Queueof}\, R\, L\}$$

The copy problem

Incorrect specification for copy:

$$\{p \rightsquigarrow \mathsf{Queueof}\, R\, L\}$$
$$(\mathsf{copy}\, p)$$
$$\{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \ * \ p' \rightsquigarrow \mathsf{Queueof}\, R\, L\}$$

Exercise: specify a function copy $f\ p$ that duplicates a mutable queue specified using Queueof, where f is a function to duplicate items.

The copy problem

Incorrect specification for `copy`:

$$\begin{array}{l}\{p \rightsquigarrow \mathsf{Queueof}\, R\, L\} \\ (\mathsf{copy}\, p) \\ \{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \ * \ p' \rightsquigarrow \mathsf{Queueof}\, R\, L\}\end{array}$$

Exercise: specify a function `copy` $f\, p$ that duplicates a mutable queue specified using Queueof, where f is a function to duplicate items.

$$\begin{array}{ll} & \left(\forall x X.\ \{x \rightsquigarrow R\, X\}\ (f\, x)\ \{\lambda x'.\ x \rightsquigarrow R\, X \ * \ x' \rightsquigarrow R\, X\}\right) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\} \\ & (\mathsf{copy}\, f\, p) \\ & \{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \ * \ p' \rightsquigarrow \mathsf{Queueof}\, R\, L\}\end{array}$$

Chapter 20

Higher-order representation predicates for records

Representation for records

$$p \rightsquigarrow \mathsf{MCellOf}\ P_1\ P_2 \quad \equiv \quad \exists v_1 v_2.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2|\!\} \ast\ v_1 \rightsquigarrow P_1 \ast\ v_2 \rightsquigarrow P_2$$

Recall: for $P : \mathsf{val} \to \mathsf{Hprop}$, $v \rightsquigarrow P$ is just a notation for $P\ v$

P is typically of the form $R\,X$ e.g. $\mathsf{Mlist}\,L$ or $\mathsf{Mlistof}\,R\,L$

Representation predicate for lists, revisited

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid X :: L' \Rightarrow \exists x p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & \qquad\qquad * \;\; x \rightsquigarrow R\, X \\
& & \qquad\qquad * \;\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L'
\end{array}
$$

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MCellOf}\ P_1\ P_2 & \equiv & \exists v_1 v_2.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2|\!\} \\
& & \qquad\qquad * \;\; v_1 \rightsquigarrow P_1 \\
& & \qquad\qquad * \;\; v_2 \rightsquigarrow P_2
\end{array}
$$

Exercise: rewrite the definition of Mlistof using MCellof.

Representation predicate for lists, revisited

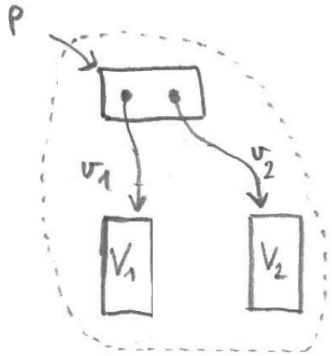
$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\
\qquad \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
\qquad \mid X :: L' \Rightarrow \exists x p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
\qquad\qquad\qquad\qquad * \;\; x \rightsquigarrow R\, X \\
\qquad\qquad\qquad\qquad * \;\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L'
\end{array}
$$

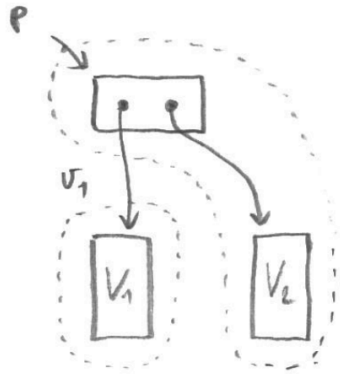
$$
\begin{array}{l}
p \rightsquigarrow \mathsf{MCellOf}\ P_1\ P_2 \;\equiv\; \exists v_1 v_2.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2|\!\} \\
\qquad\qquad\qquad\qquad * \;\; v_1 \rightsquigarrow P_1 \\
\qquad\qquad\qquad\qquad * \;\; v_2 \rightsquigarrow P_2
\end{array}
$$

Exercise: rewrite the definition of Mlistof using MCellof.

$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \equiv \mathsf{match}\, L\, \mathsf{with} \\
\qquad \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
\qquad \mid X :: L' \Rightarrow p \rightsquigarrow \mathsf{MCellof}\,(R\, X)\,(\mathsf{Mlistof}\, R\, L')
\end{array}
$$

Focus/unfocus for accessing a record field

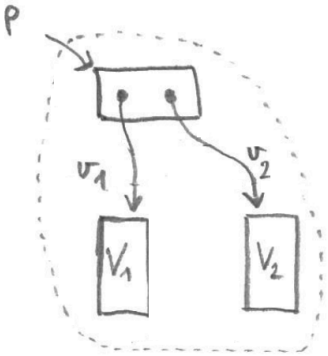


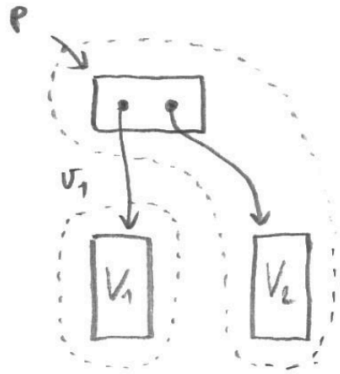


Focus on a field:

$$
\begin{aligned}
p \rightsquigarrow \mathsf{MCellof}\, P_1\, P_2 &= \exists v_1.\ p \rightsquigarrow \mathsf{MCellof}\,(\mathsf{Id}\, v_1)\, P_2 \ * \ v_1 \rightsquigarrow P_1 \\
&= \exists v_2.\ p \rightsquigarrow \mathsf{MCellof}\, P_1\,(\mathsf{Id}\, v_2) \ * \ v_2 \rightsquigarrow P_2
\end{aligned}
$$

Focus/unfocus for accessing a record field





Focus on a field:

$$
\begin{aligned}
p \rightsquigarrow \mathsf{MCellof}\, P_1\, P_2 &= \exists v_1.\ p \rightsquigarrow \mathsf{MCellof}\,(\mathsf{Id}\, v_1)\, P_2 \ * \ v_1 \rightsquigarrow P_1 \\
&= \exists v_2.\ p \rightsquigarrow \mathsf{MCellof}\, P_1\,(\mathsf{Id}\, v_2) \ * \ v_2 \rightsquigarrow P_2
\end{aligned}
$$

Access to a focused field:

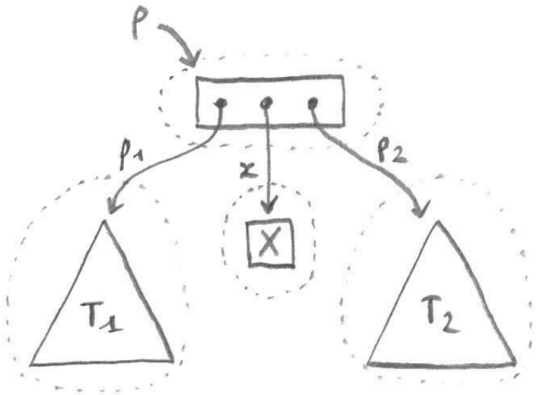
$$\{p \rightsquigarrow \mathsf{MCellof}\,(\mathsf{Id}\, v_1)\, P_2\}\ (p.\mathsf{hd})\ \{\lambda x.\ulcorner x = v_1 \urcorner * p \rightsquigarrow \mathsf{MCellof}\,(\mathsf{Id}\, v_1)\, P_2\}$$

$$\{p \rightsquigarrow \mathsf{MCellof}\,(\mathsf{Id}\, v_1)\, P_2\}\ (p.\mathsf{hd} \mathrel{<\!\!-} w)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MCellof}\,(\mathsf{Id}\, w)\, P_2\}$$

Chapter 21

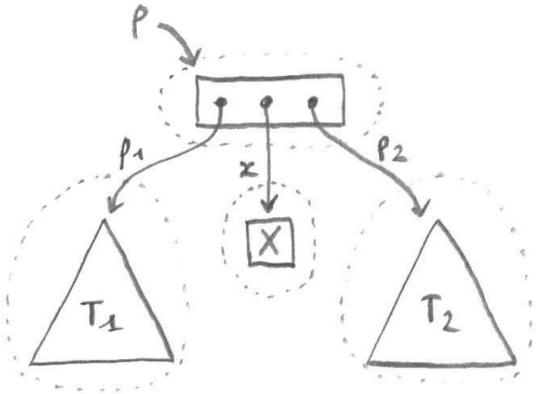
Higher-order representation predicates for trees

Binary tree: representation



$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \;\equiv\; \mathsf{match}\, T\, \mathsf{with} \\
\qquad \mid \mathsf{Leaf} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
\qquad \mid \mathsf{Node}\, X\, T_1\, T_2 \;\Rightarrow\; \exists x p_1 p_2. \\
\qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
\qquad\qquad * \;\; x \rightsquigarrow R\, X \\
\qquad\qquad * \;\; p_1 \rightsquigarrow \mathsf{Mtreeof}\, R\, T_1 \\
\qquad\qquad * \;\; p_2 \rightsquigarrow \mathsf{Mtreeof}\, R\, T_2
\end{array}
$$

Binary tree: representation, revisited

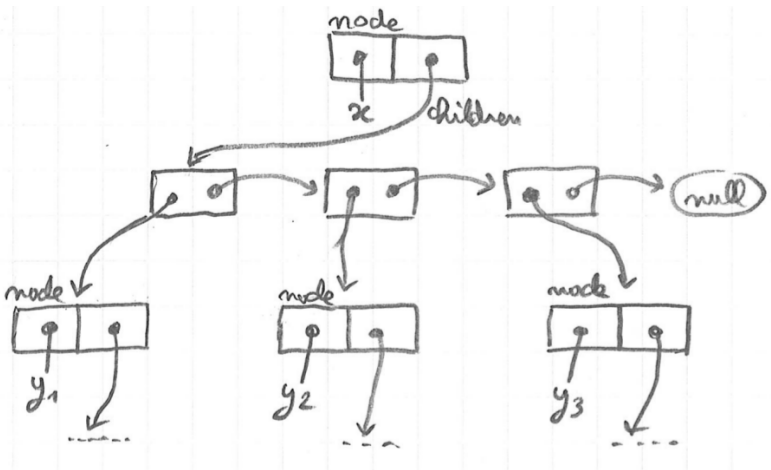


Representation predicate for tree cells:

$$p \rightsquigarrow \mathsf{Nodeof}\, P_1\, P_2\, P_3 \;\equiv$$
$$\exists v_1 v_2 v_3.\quad p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{left}{=}v_2;\ \mathsf{right}{=}v_3|\!\} \;*\; v_1 \rightsquigarrow P_1 \;*\; v_2 \rightsquigarrow P_2 \;*\; v_3 \rightsquigarrow P_3$$

$$p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \;\equiv\; \mathsf{match}\, T\, \mathsf{with}$$
$$|\, \mathsf{Leaf} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner$$
$$|\, \mathsf{Node}\, X\, T_1\, T_2 \;\Rightarrow$$
$$p \rightsquigarrow \mathsf{Nodeof}\, (R\, X)\, (\mathsf{Mtreeof}\, R\, T_1)\, (\mathsf{Mtreeof}\, R\, T_2)$$

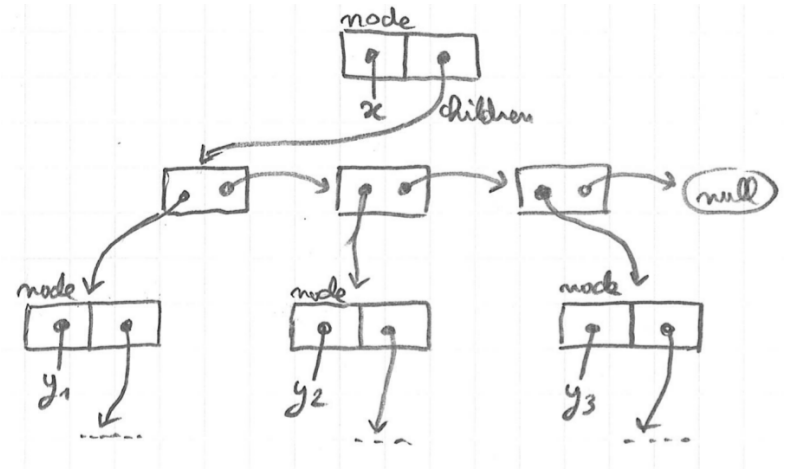
Trees with list of subtrees: implementation




```
type 'a node = {
  mutable item : 'a;
  mutable children : ('a node) cell }
```

```
Inductive tree (A:Type) : Type :=
  | Leaf : tree A
  | Node : A → list (tree A) → tree A.
```

Trees with list of subtrees: specification



$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$

$\mathsf{match}\ T\ \mathsf{with}$

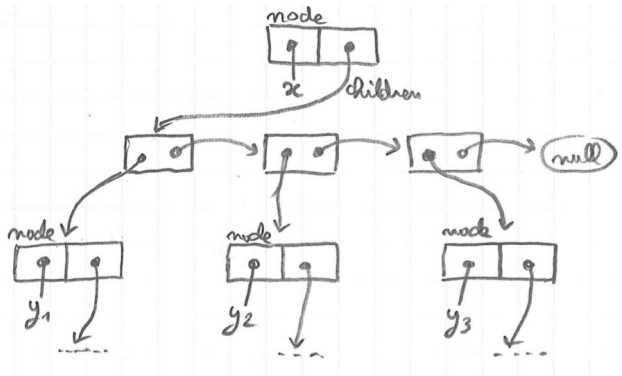
$|\ \mathsf{Leaf} \Rightarrow\ \ulcorner p = \mathsf{null} \urcorner$

$|\ \mathsf{Node}\ X\ L \Rightarrow\ \exists x c.\ p \mapsto \{|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\}$

$\quad * \ x \rightsquigarrow R\ X$

$\quad * \ c \rightsquigarrow \mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R)\ L$

Trees with list of subtrees: representation of nodes

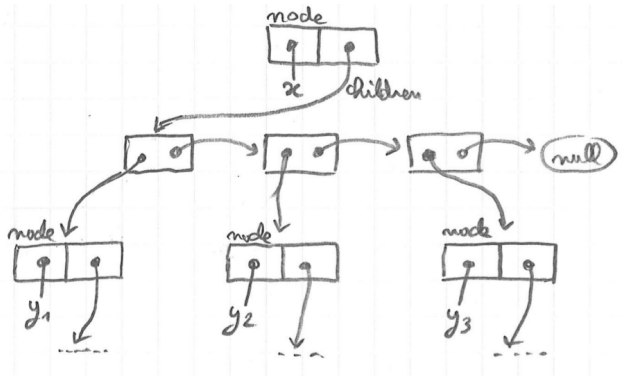


$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$
$\quad \mathsf{match}\ T\ \mathsf{with}$
$\quad | \ \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \urcorner$
$\quad | \ \mathsf{Node}\ X\ L \Rightarrow \exists x c.\ p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\!\}$
$\qquad\qquad\qquad\qquad * \ x \rightsquigarrow R\ X$
$\qquad\qquad\qquad\qquad * \ c \rightsquigarrow \mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R)\ L$

$p \rightsquigarrow \mathsf{Nodeof}'\ P_1\ P_2 \equiv \exists v_1 v_2.\, p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{children}{=}v_2|\!\} * v_1 \rightsquigarrow P_1 * v_2 \rightsquigarrow P_2$

Exercise: rewrite the definition of Narytreeof using Nodeof′.

Trees with list of subtrees: representation of nodes



$$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$$
$$\mathsf{match}\ T\ \mathsf{with}$$
$$|\ \mathsf{Leaf} \Rightarrow\ \ulcorner p = \mathsf{null} \urcorner$$
$$|\ \mathsf{Node}\ X\ L \Rightarrow\ \exists x c.\ p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\!\} * x \rightsquigarrow R\ X * c \rightsquigarrow \mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R)\ L$$

$$p \rightsquigarrow \mathsf{Nodeof}'\ P_1\ P_2 \equiv \exists v_1 v_2.\ p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{children}{=}v_2|\!\} * v_1 \rightsquigarrow P_1\ *\ v_2 \rightsquigarrow P_2$$

Exercise: rewrite the definition of Narytreeof using Nodeof′.

$$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$$
$$\mathsf{match}\ T\ \mathsf{with}$$
$$|\ \mathsf{Leaf} \Rightarrow\ \ulcorner p = \mathsf{null} \urcorner$$
$$|\ \mathsf{Node}\ X\ L \Rightarrow\ p \rightsquigarrow \mathsf{Nodeof}'\ (R\ X)\ (\mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R)\ L)$$

Chapter 22

Iteration with higher-order representation predicates

Iteration on lists

Recall:

$$\forall f l I.\quad \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big)$$
$$\Rightarrow\quad \{I\,\mathsf{nil}\}\ (\texttt{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$

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

Iteration on lists

Recall:

$$\begin{array}{rl} \forall f l I. & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big) \\ \Rightarrow & \{I\,\mathsf{nil}\}\ (\texttt{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

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

Challenge:

$$\begin{array}{rl} & \big(\forall x \ldots.\ \{\ldots\}\ (f\,x)\ \{\lambda\_.\ \ldots\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L\ *\ \ldots\}\ (\texttt{miter}\,f\,p)\ \{\lambda\_.\ p \rightsquigarrow \ldots\ *\ \ldots\} \end{array}$$

Question: Can we use an invariant $I \equiv \lambda K.(\ldots)$?

(i.e. with a spec of the form $\{p \rightsquigarrow \ldots * I\ \mathsf{nil}\}(\ldots)\{p \rightsquigarrow \ldots * I\ L\}$?)

Iterating over a mutable list of mutable items

Exercise: specify the function `miter`, using an invariant of the form $J\,K\,K'$, describing the state before and the state after the iteration.

Iterating over a mutable list of mutable items

Exercise: specify the function `miter`, using an invariant of the form $J\,K\,K'$, describing the state before and the state after the iteration.

$$\begin{array}{lll} \forall f\,p\,R\,L\,J. & \big(\forall x\,X\,K\,K'.\ \{x \rightsquigarrow R\,X\ *\ J\,K\,K'\} & \big) \\ & \quad (f\,x) & \\ & \quad \{\lambda\_.\ \exists X'.\ x \rightsquigarrow R\,X'\ *\ J\,(K \& X)\,(K' \& X')\} & \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L\ *\ J\,\mathsf{nil}\,\mathsf{nil}\} & \\ & (\texttt{miter}\,f\,p) & \\ & \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,L'\ *\ J\,L\,L'\} & \end{array}$$

Incrementing a mutable list of distinct references

```
let incr_all p = miter incr p
let example_p = {hd = ref 5; tl = {hd = ref 3; tl = null}}
```

$$x \rightsquigarrow \mathsf{Ref}\, X \quad \equiv \quad x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the functions `incr` and `miter incr`. What is $J\ K\ K'$?

Incrementing a mutable list of distinct references

```
let incr_all p = miter incr p
let example_p = {hd = ref 5; tl = {hd = ref 3; tl = null}}
```

$$x \rightsquigarrow \mathsf{Ref}\, X \quad \equiv \quad x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the functions `incr` and `miter incr`. What is $J\ K\ K'$?

$$\forall x X\ \{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{incr x})\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}$$

Incrementing a mutable list of distinct references

```
let incr_all p = miter incr p
let example_p = {hd = ref 5; tl = {hd = ref 3; tl = null}}
```

$$x \rightsquigarrow \mathsf{Ref}\, X \quad \equiv \quad x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the functions `incr` and `miter incr`. What is $J\ K\ K'$?

$$\forall x X\ \{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{incr x})\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\, (X+1)\}$$

$$\forall p L\ \{p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Ref}\, L\}\ (\texttt{miter incr}\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Ref}\, (\mathsf{map}\, (+1)\, L)\}$$

Incrementing a mutable list of distinct references

```
let incr_all p = miter incr p
let example_p = {hd = ref 5; tl = {hd = ref 3; tl = null}}
```

$$x \rightsquigarrow \mathsf{Ref}\, X \;\equiv\; x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the functions `incr` and `miter incr`. What is $J\ K\ K'$?

$$\forall x X\ \{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{incr x})\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}$$

$$\forall p L\ \{p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\, L\}\ (\texttt{miter incr}\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\,(\mathsf{map}\,(+1)\,L)\}$$

$$J\,K\,K' \;\equiv\; \ulcorner K' = \mathsf{map}\,(+1)\,K \urcorner$$

Incrementing a mutable list of distinct references (2/2)

$$\begin{array}{lll} \forall\, f\, p\, R\, L\, J. & \big(\forall x\, X\, K\, K'.\ \{x \rightsquigarrow R\, X\ *\ J\, K\, K'\} & \\ & \quad (f\, x) & \\ & \quad \{\lambda\_.\ \exists X'.\ x \rightsquigarrow R\, X'\ *\ J\, (K \& X)\, (K' \& X')\} & \big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\, R\, L\ *\ J\, \mathsf{nil}\, \mathsf{nil}\} & \\ & (\texttt{miter}\, f\, p) & \\ & \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\, R\, L'\ *\ J\, L\, L'\} & \end{array}$$

Consider:

$$J\, K\, K' \ \equiv\ \ulcorner K' = \mathsf{map}\, (+1)\, K \urcorner$$

Derives:

$$\big(\forall x X.\ \{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{incr}\, x)\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\, (X+1)\}\big) \Rightarrow$$

$$\{p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Ref}\, L\}\ (\texttt{miter incr}\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Ref}\, (\mathsf{map}\, (+1)\, L)\}$$

Chapter 23

Resource analysis in Separation Logic

Controlling deallocation

(1) Remove the "GC" part from the definition the triple $\{H\}\ t\ \{Q\}$:

$$\forall H' m.\ (H * H')\, m \ \Rightarrow\ \exists v m'.\ <t, m> \Downarrow <v, m'> \wedge (Q\, v * H' \cancel{* \mathsf{GC}})\, m'$$

(2) Replace the GC rule, add a "free" function for explicit deallocation:

$$\cancel{\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}} \qquad \rightsquigarrow \qquad \frac{}{\{r \mapsto \vec{v}\}\ \texttt{free}\ r\ \{\lambda\_.\ \ulcorner\urcorner\}}$$

Controlling deallocation

(1) Remove the "GC" part from the definition the triple $\{H\}\ t\ \{Q\}$:

$$\forall H' m.\ (H * H')\, m \ \Rightarrow\ \exists v m'.\ <t, m> \Downarrow <v, m'> \wedge (Q\, v * H' \cancel{* \mathsf{GC}})\, m'$$

(2) Replace the GC rule, add a "free" function for explicit deallocation:

$$\cancel{\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}} \qquad \rightsquigarrow \qquad \frac{}{\{r \mapsto \vec{v}\}\ \texttt{free}\ r\ \{\lambda\_.\ \ulcorner\urcorner\}}$$

(3) Theorem: for a full program execution starting in the empty heap, all the data still allocated at the end is described in the post-condition.

(4) Corollary: terminating on the empty heap ensures no memory leaks.

$$\{\ulcorner\urcorner\}\ t\ \{\lambda n.\ \ulcorner P\, n\urcorner\}$$

Controlling deallocation

(1) Remove the "GC" part from the definition the triple $\{H\}\ t\ \{Q\}$:

$$\forall H' m.\ (H * H')\, m \ \Rightarrow\ \exists v m'.\ <t, m> \Downarrow <v, m'> \wedge (Q\, v * H' \cancel{* \mathsf{GC}})\, m'$$

(2) Replace the GC rule, add a "free" function for explicit deallocation:

$$\cancel{\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}} \qquad \rightsquigarrow \qquad \frac{}{\{r \mapsto \vec{v}\}\ \texttt{free}\ r\ \{\lambda\_.\ \ulcorner\urcorner\}}$$

(3) Theorem: for a full program execution starting in the empty heap, all the data still allocated at the end is described in the post-condition.

(4) Corollary: terminating on the empty heap ensures no memory leaks.

$$\{\ulcorner\urcorner\}\ t\ \{\lambda n.\ \ulcorner P\, n\urcorner\}$$

(separation logics with GC are sometimes called "affine" or "intuitionistic")

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

$$\{\ulcorner\urcorner\}\ (\texttt{fopen s})\ \{\lambda f.\ \exists L.\ f \rightsquigarrow \mathsf{File}\, L\}$$

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

$$\{\ulcorner\urcorner\}\ (\texttt{fopen s})\ \{\lambda f.\ \exists L.\ f \rightsquigarrow \mathsf{File}\, L\}$$
$$\{f \rightsquigarrow \mathsf{File}\,(c :: L)\}\ (\texttt{fread f})\ \{\lambda x.\ \ulcorner x = c\urcorner * f \rightsquigarrow \mathsf{File}\, L\}$$

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

$$\{\ulcorner\urcorner\}\ (\texttt{fopen s})\ \{\lambda f.\ \exists L.\ f \rightsquigarrow \mathsf{File}\, L\}$$
$$\{f \rightsquigarrow \mathsf{File}\,(c :: L)\}\ (\texttt{fread f})\ \{\lambda x.\ \ulcorner x = c\urcorner * f \rightsquigarrow \mathsf{File}\, L\}$$
$$\{f \rightsquigarrow \mathsf{File}\, L\}\ (\texttt{fclose f})\ \{\lambda\_.\ \ulcorner\urcorner\}$$

Complexity analysis

Time credits:

$$\$\,x \quad : \quad \mathsf{Hprop} \qquad\qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) \;=\; \$\,x \;\;*\;\; \$\,y \qquad \text{and} \qquad \$\,0 \;=\; \lceil\rceil$$

Complexity analysis

Time credits:

$$\$\,x \quad : \quad \mathsf{Hprop} \qquad\qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) \;=\; \$\,x \;\;*\;\; \$\,y \quad \text{and} \quad \$\,0 \;=\; \lceil\rceil$$

Principle:

The execution of every instruction costs $\$1$.

Simplification:

Entering the body of a function or a loop costs $\$1$.

Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \mathsf{Array}\, M \;*\; \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\ulcorner n = |M|\urcorner * t \rightsquigarrow \mathsf{Array}\, M\}$$

Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \mathsf{Array}\, M \;*\; \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\ulcorner n = |M|\urcorner * t \rightsquigarrow \mathsf{Array}\, M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\}\ (\texttt{Array.make n v})\ \{\lambda t.\ \exists L.\ t \rightsquigarrow \mathsf{Array}\, L \;*\; \ulcorner ... \urcorner\}$$

Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \mathsf{Array}\, M \ * \ \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\ulcorner n = |M|\urcorner * t \rightsquigarrow \mathsf{Array}\, M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\}\ (\texttt{Array.make n v})\ \{\lambda t.\ \exists L.\ t \rightsquigarrow \mathsf{Array}\, L \ * \ \ulcorner ... \urcorner\}$$

Quasilinear time:

$$\{t \rightsquigarrow \mathsf{Array}\, L \ * \ \$(c_1 |L| \log |L| + c_2)\}$$
$$(\texttt{Array.sort t})$$
$$\{\lambda t.\ \exists L'.\ t \rightsquigarrow \mathsf{Array}\, L' \ * \ \ulcorner ... \urcorner\}$$

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$\{\$\, c\}$ (`Stack.create()`) $\{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\,\mathsf{nil}\}$

$\{s \rightsquigarrow \mathsf{Stack}\, L \ast \$\, c\}$ (`Stack.push s x`) $\{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\,(x :: L)\}$

$\{s \rightsquigarrow \mathsf{Stack}\,(x :: L) \ast \$\, c\}$ (`Stack.pop s`) $\{\lambda y.\ \ulcorner y = x \urcorner \ast s \rightsquigarrow \mathsf{Stack}\, L\}$

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$$\{\$\,c\} \qquad (\texttt{Stack.create()})\ \{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\,\mathsf{nil}\}$$

$$\{s \rightsquigarrow \mathsf{Stack}\,L \ast \$\,c\} \qquad (\texttt{Stack.push s x})\ \{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\,(x :: L)\}$$

$$\{s \rightsquigarrow \mathsf{Stack}\,(x :: L) \ast \$\,c\}\ (\texttt{Stack.pop s}) \qquad \{\lambda y.\ \ulcorner y = x \urcorner \ast s \rightsquigarrow \mathsf{Stack}\,L\}$$

Representation predicate with a potential function:

$$\begin{aligned} s \rightsquigarrow \mathsf{Stack}\,L \quad \equiv \quad \exists n t M k.\quad & s \mapsto \{\!|\mathsf{size}{=}n;\ \mathsf{data}{=}t|\!\} \\ & \ast\ t \rightsquigarrow \mathsf{Array}\,M \\ & \ast\ \ulcorner n = |L| \leqslant |M| = 2^k \urcorner \\ & \ast\ \ulcorner \forall i \in [0, n).\ M[i] = L[i] \urcorner \\ & \ast\ \$(c' \cdot \mathsf{abs}(n - |M|/2)) \end{aligned}$$

Space complexity analysis

Suppose $\diamondsuit 1$ represents one *space credit* (Hoffmann, 1999)
Then allocation consumes credits; deallocation produces credits :

$$\{\diamondsuit \mathsf{size(b)}\} \quad \mathsf{alloc}(b) \quad \{\lambda x.\ x \mapsto b\}$$

$$\{x \mapsto b\} \quad \mathsf{free}(x) \quad \{\lambda \_.\ \diamondsuit \mathsf{size(b)}\}$$

The same mechanisms apply (e.g. $\diamondsuit(a+b) = \diamondsuit a * \diamondsuit b$, amortized analysis, etc.).

Space complexity analysis

Suppose $\diamondsuit 1$ represents one *space credit* (Hoffmann, 1999)
Then allocation consumes credits; deallocation produces credits :

$$\{\diamondsuit \mathsf{size(b)}\} \quad \mathsf{alloc}(b) \quad \{\lambda x.\ x \mapsto b\}$$

$$\{x \mapsto b\} \quad \mathsf{free}(x) \quad \{\lambda \_.\ \diamondsuit \mathsf{size(b)}\}$$

The same mechanisms apply (e.g. $\diamondsuit(a+b) = \diamondsuit a * \diamondsuit b$, amortized analysis, etc.).

What if we add **garbage collection** ?

Fractional permissions

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

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

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

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

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

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 $0 < \alpha, \beta \leqslant 1$:

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

Operations:

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

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

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

Fractional permissions in practice

$$\forall \alpha\beta.\ \{a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ *\ a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ *\ a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 * a_3 \overset{1}{\rightsquigarrow} \mathsf{Array}\, (L_1 +\!\!+ L_2)\}$$

Fractional permissions in practice

$$\forall \alpha\beta.\ \{a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ *\ a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ *\ a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 * a_3 \overset{1}{\rightsquigarrow} \mathsf{Array}\, (L_1 +\!\!+ L_2)\}$$

Limitations:
– need to quantify fractions explicitly,
– need to syntactic sugar to avoid copy-pasting,
– need to re-establish post-conditions,
– a fraction $\frac{1}{2}H$ cannot be defined for arbitrary H.

Those can be alleviated with a duplicable read-only modality $\mathsf{RO}(H)$.

Chapter 24

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
```

Efficient use of parallel pairs with granularity control

```
let rec sum a i j =
  if j - i < sequential_cutoff then begin
    let r = ref 0 in
    for k = i to j-1 do
      r := !r + a.(k)
    done;
    !r
  end else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
    s1 + s2
  end
```

Efficient use of parallel pairs with granularity control

```
let rec sum a i j =
  if j - i < sequential_cutoff then begin
    let r = ref 0 in
    for k = i to j-1 do
      r := !r + a.(k)
    done;
    !r
  end 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

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

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

Reasoning rule for parallel pairs

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

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

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 = create_lock()

let concurrent_step () =
  acquire_lock p;
  incr r;
  decr s;
  release_lock 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 = create_lock()

let concurrent_step () =
  acquire_lock p;
  incr r;
  decr s;
  release_lock 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:

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

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

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

Concurrent locks: exercise

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

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

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

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

Exercise: Proof sketch of declarations; spec. and proof of function.

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

let concurrent_step () =
   acquire_lock p;
   incr r; decr s;
   release_lock p
```

Concurrent locks: exercise

Exercise: Proof sketch of declarations; spec. and proof of function.

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

let concurrent_step () =
   acquire_lock p;
   incr r; decr s;
   release_lock 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 $i + 1$.

Concurrent locks: can we prove this program?

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

let f () =
  acquire_lock p;
  incr r;
  release_lock p

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

Chapter 25

Ghost state

Same non-example

```
let r = ref 0
let p = create_lock()
let f () = acquire_lock p; incr r; release_lock p
let () = let _ = (| f(), f() |) in acquire_lock p; assert (!r == 2)
```

rewritten graphically as:

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

acquire_lock p;   ||   acquire_lock p;
r := !r + 1;      ||   r := !r + 1;
release_lock p;   ||   release_lock p;

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

Same non-example

```
let r = ref 0
let p = create_lock()
let f () = acquire_lock p; incr r; release_lock p
let () = let _ = (| f(), f() |) in acquire_lock p; assert (!r == 2)
```

rewritten graphically as:

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

```
acquire_lock p;          ||    acquire_lock p;
r := !r + 1;             ||    r := !r + 1;
release_lock p;          ||    release_lock p;
```

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

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

Same non-example

```
let r = ref 0
let p = create_lock()
let f () = acquire_lock p; incr r; release_lock p
let () = let _ = (| f(), f() |) in acquire_lock p; assert (!r == 2)
```

rewritten graphically as:

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

```
acquire_lock p;          ||    acquire_lock p;
r := !r + 1;             ||    r := !r + 1;
release_lock p;          ||    release_lock p;
```

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

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

Same non-example

```
let r = ref 0
let p = create_lock()
let f () = acquire_lock p; incr r; release_lock p
let () = let _ = (| f(), f() |) in acquire_lock p; assert (!r == 2)
```

rewritten graphically as:

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

```
acquire_lock p;          ||    acquire_lock p;
r := !r + 1;             ||    r := !r + 1;
release_lock p;          ||    release_lock p;
```

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

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

Same non-example

```
let r = ref 0
let p = create_lock()
let f () = acquire_lock p; incr r; release_lock p
let () = let _ = (| f(), f() |) in acquire_lock p; assert (!r == 2)
```

rewritten graphically as:

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

```
acquire_lock p;          ||    acquire_lock p;
r := !r + 1;             ||    r := !r + 1;
release_lock p;          ||    release_lock p;
```

```
acquire_lock 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 $\{\ulcorner\mathsf{True}\urcorner\}$).

Same non-example

```
let r = ref 0
let p = create_lock()
let f () = acquire_lock p; incr r; release_lock p
let () = let _ = (| f(), f() |) in acquire_lock p; assert (!r == 2)
```

rewritten graphically as:

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

```
acquire_lock p;          ||    acquire_lock p;
r := !r + 1;             ||    r := !r + 1;
release_lock p;          ||    release_lock p;
```

```
acquire_lock 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 $\{\ulcorner\mathsf{True}\urcorner\}$). btw: rule for assert?

More variables! Ghost variables.

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

`acquire_lock p;`	`acquire_lock p;`
`r := !r + 1;`	`r := !r + 1;`
`r1 := !r1 + 1;`	`r2 := !r2 + 1;`
`release_lock p;`	`release_lock p;`

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

More variables! Ghost variables.

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

`acquire_lock p;`	‖	`acquire_lock p;`
`r := !r + 1;`	‖	`r := !r + 1;`
`r1 := !r1 + 1;`	‖	`r2 := !r2 + 1;`
`release_lock p;`	‖	`release_lock p;`

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

Exercise: Give a lock invariant that allows proving {True} program {True} (hint: fractional permissions). Then prove the triple.

More variables! Ghost variables.

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

```
acquire_lock p;          ||    acquire_lock p;
r := !r + 1;             ||    r := !r + 1;
r1 := !r1 + 1;           ||    r2 := !r2 + 1;
release_lock p;          ||    release_lock p;
```

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

Exercise: Give a lock invariant that allows proving {True} program {True} (hint: fractional permissions). Then prove the triple.

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

Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

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

Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```
$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$
$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$
```
let p = create_lock()
```

Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

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

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

Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

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

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

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

Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

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

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

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

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

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

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

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

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

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

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

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

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

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

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

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

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

```
release_lock p;
```

Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

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

```
acquire_lock p;
```

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

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

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

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

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

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

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

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

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

```
release_lock p;
```

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

Right thread

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

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

```
acquire_lock 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_lock 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 = create_lock()
```

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

```
acquire_lock p;          ‖   acquire_lock p;
r := !r + 1;             ‖   r := !r + 1;
r1 := !r1 + 1;           ‖   r2 := !r2 + 1;
release_lock p;          ‖   release_lock 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_lock p;
```

$\{r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner\}$

$\{r_1 \mapsto 1 * r_2 \mapsto 1 * r \mapsto n * \ulcorner n = 1 + 1 \urcorner\}$

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

Some remarks

Ghost variables are the basic way of solving this problem

Some remarks

Ghost variables are the basic way of solving this problem, but:

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

Some remarks

Ghost variables are the basic way of solving this problem, but:

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

Ghost state is a more robust approach. How they work in *Iris*:

- allocation of ghost state: for some a any "Resource Algebra":

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \gamma \rightsquigarrow \mathsf{Ghost}(a)$$

Some remarks

Ghost variables are the basic way of solving this problem, but:

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

Ghost state is a more robust approach. How they work in *Iris*:

- allocation of ghost state: for some a any "Resource Algebra":

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \gamma \rightsquigarrow \mathsf{Ghost}(a)$$

- splitting of ghost state: for any a, b in an RA:

$$\gamma \rightsquigarrow \mathsf{Ghost}(a \cdot b) \Leftrightarrow \gamma \rightsquigarrow \mathsf{Ghost}(a) * \gamma \rightsquigarrow \mathsf{Ghost}(b)$$

Some remarks

Ghost variables are the basic way of solving this problem, but:

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

Ghost state is a more robust approach. How they work in *Iris*:

- allocation of ghost state: for some a any "Resource Algebra":

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \gamma \rightsquigarrow \mathsf{Ghost}(a)$$

- splitting of ghost state: for any a, b in an RA:

$$\gamma \rightsquigarrow \mathsf{Ghost}(a \cdot b) \Leftrightarrow \gamma \rightsquigarrow \mathsf{Ghost}(a) * \gamma \rightsquigarrow \mathsf{Ghost}(b)$$

- validity in RA's e.g. $\mathsf{valid}((r_2 \mapsto 1) \cdot (r_2 \mapsto n_2)) \;\Rightarrow\; n_2 = 1$
- heaps are a RA (composition of fractional RA and agreement RA)

Chapter 26: weakest preconditions

Presentation with weakest preconditions

WP are generally preferred to triples, in practice:

- more primitive:

$$\{P\}e\{\Phi\} \;\equiv\; P \mathrel{-\!\!*} \mathsf{wp}\, e\, \Phi$$

- preconditions become hypotheses, more easily managed

Builtin consequence rule in postconditions

$(\mathsf{wp}\, e\, -)$ is a monotone predicate transformer, so $\mathsf{wp}\, e\, \Phi$ is equivalent to

$$\forall \Psi\ (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi$$

Builtin consequence rule in postconditions

$(\mathsf{wp}\, e\, -)$ is a monotone predicate transformer, so $\mathsf{wp}\, e\, \Phi$ is equivalent to

$$\forall \Psi\ (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi$$

We could choose:

$$\{P\}e\{\Phi\} \ \equiv\ P \mathrel{-\!\!*} \mathsf{wp}\, e\, \Phi$$

the following is equivalent:

$$\{P\}e\{\Phi\} \ \equiv\ \forall \Psi\ P \mathrel{-\!\!*} (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi$$

which is easier to apply since Ψ is universally quantified.

(Note that separating implication $\mathrel{-\!\!*}$ has no easy "heap entailment" $\rhd$ counterpart here, $\mathrel{-\!\!*}$ is more expressive.)

Simplified rules for load, store, alloc

Exercise:

$$\forall \ell v \Phi \quad \ldots\ldots\ldots \quad -\!\!* \left(\ldots\ldots\ldots\ldots\ldots\ldots\right) \quad -\!\!* \mathsf{wp}\,(!\ell)\,\Phi$$
$$\forall \ell v v' \Phi \quad \ldots\ldots\ldots \quad -\!\!* \left(\ldots\ldots\ldots\ldots\ldots\ldots\right) \quad -\!\!* \mathsf{wp}\,(\ell \leftarrow v)\,\Phi$$
$$\forall v \Phi \quad \ldots\ldots\ldots \quad -\!\!* \left(\ldots\ldots\ldots\ldots\ldots\ldots\right) \quad -\!\!* \mathsf{wp}\,(\texttt{ref}\, v)\,\Phi$$

Simplified rules for load, store, alloc

$$
\begin{array}{llll}
\forall \ell v \Phi & \ell \mapsto v & -\!\!* \, (\ell \mapsto v -\!\!* \, \Phi v) & -\!\!* \, \mathsf{wp}\,(!\ell)\,\Phi \\
\forall \ell v v' \Phi & \ell \mapsto v' & -\!\!* \, (\ell \mapsto v -\!\!* \, \Phi()) & -\!\!* \, \mathsf{wp}\,(\ell \leftarrow v)\,\Phi \\
\forall v \Phi & \top & -\!\!* \, (\forall \ell \;\; \ell \mapsto v -\!\!* \, \Phi \ell) & -\!\!* \, \mathsf{wp}\,(\texttt{ref}\, v)\,\Phi
\end{array}
$$

Exemple

$$\{\ell \mapsto 1\}\ \texttt{ref}\ 3\ \{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

Exemple

$$\{\ell \mapsto 1\}\ \texttt{ref}\ 3\ \{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi\ \ (\ell \mapsto 1) \mathrel{-\!\!*} (\forall \ell'.\ell \mapsto 1 * \ell' \mapsto 3 \mathrel{-\!\!*} \Phi \ell') \mathrel{-\!\!*} \mathsf{wp}\,(\texttt{ref}\ 3)\,\Phi$$

Exemple

$$\{\ell \mapsto 1\}\ \texttt{ref}\ 3\ \{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi\ \ (\ell \mapsto 1) \mathrel{-\!\!*} (\forall \ell'.\ell \mapsto 1 * \ell' \mapsto 3 \mathrel{-\!\!*} \Phi \ell') \mathrel{-\!\!*} \mathsf{wp}\,(\texttt{ref}\ 3)\,\Phi$$

show in Rocq if there's time.

Chapter 27: modalities

Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$\square P \rhd P$ $\quad$ $\square P \rhd \square P * P$ $\quad$ $\square P \rhd \square P * \square P$ $\quad$ $\square P \rhd \square P * P * P$

$\ulcorner P \urcorner \rhd \square \ulcorner P \urcorner$ $\quad$ $\square(\ell \mapsto 1) \rhd \ulcorner$False$\urcorner$ $\quad$ $\square(\ell \overset{q}{\mapsto} 1) \rhd \ulcorner$q=0$\urcorner$ (if even allowed)

Persistently modality $\Box$

$\Box P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\Box P \rhd P \qquad \Box P \rhd \Box P * P \qquad \Box P \rhd \Box P * \Box P \qquad \Box P \rhd \Box P * P * P$$

$$\ulcorner P \urcorner \rhd \Box \ulcorner P \urcorner \qquad \Box(\ell \mapsto 1) \rhd \ulcorner \mathsf{False} \urcorner \qquad \Box(\ell \overset{q}{\mapsto} 1) \rhd \ulcorner \mathsf{q{=}0} \urcorner \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \;\equiv\; \Box(\forall \Psi \;\; P \mathrel{-\!\!*} (\forall v \;\; \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\square P \rhd P \qquad \square P \rhd \square P * P \qquad \square P \rhd \square P * \square P \qquad \square P \rhd \square P * P * P$$

$$\ulcorner P \urcorner \rhd \square \ulcorner P \urcorner \qquad \square(\ell \mapsto 1) \rhd \ulcorner \mathsf{False} \urcorner \qquad \square(\ell \overset{q}{\mapsto} 1) \rhd \ulcorner \mathsf{q{=}0} \urcorner \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \;\equiv\; \square(\forall \Psi \; P \mathrel{-\!\!*} (\forall v \; \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

Persistent resources can be shared between threads:

$$\frac{\{P_1 * \square P\}\; e_1 \;\{Q_1\} \qquad \{P_2 * \square P\}\; e_2 \;\{Q_2\}}{\{P_1 * P_2 * \square P\}\; (|e_1, e_2|)\; \{Q_1 \star Q_2\}}$$

Persistently modality $\Box$

$\Box P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\Box P \rhd P \qquad \Box P \rhd \Box P * P \qquad \Box P \rhd \Box P * \Box P \qquad \Box P \rhd \Box P * P * P$$

$$\ulcorner P \urcorner \rhd \Box \ulcorner P \urcorner \qquad \Box(\ell \mapsto 1) \rhd \ulcorner \mathsf{False} \urcorner \qquad \Box(\ell \overset{q}{\mapsto} 1) \rhd \ulcorner \mathsf{q{=}0} \urcorner \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \;\equiv\; \Box(\forall \Psi \; P \mathrel{-\!\!*} (\forall v \; \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

Persistent resources can be shared between threads:

$$\frac{\{P_1 * \Box P\}\; e_1 \;\{Q_1\} \qquad \{P_2 * \Box P\}\; e_2 \;\{Q_2\}}{\{P_1 * P_2 * \Box P\}\; (|e_1, e_2|)\; \{Q_1 \star Q_2\}}$$

Some ghost resources are persistents (e.g. to indicate a task is done), some are not (e.g. to provide a thread with an information it can consume).

Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

$\triangleright$ and $\square$ are modalities in Iris, where "`iProp`" are not "`heap -> Prop`" and have features such as step indexing, and resources are not only heaps.

Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

$\triangleright$ and $\square$ are modalities in Iris, where "`iProp`" are not "`heap -> Prop`" and have features such as step indexing, and resources are not only heaps.

$\triangleright^n \mathsf{False} \vdash P$ iff P holds for n steps

Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

$\triangleright$ and $\square$ are modalities in Iris, where "`iProp`" are not "`heap -> Prop`" and have features such as step indexing, and resources are not only heaps.

$\triangleright^n \mathsf{False} \vdash P$ iff P holds for n steps $\qquad \vdash \exists k.\ \triangleright^k \mathsf{False}$

More later

The **Löb rule** is very convenient for partial correctness:

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

More later

The **Löb rule** is very convenient for partial correctness:

$$\frac{Q \land \triangleright P \vdash P}{Q \vdash P}$$

Step reductions "consume" laters:

$$\frac{\{P\}e'\{Q\} \qquad e \to e'}{\{\triangleright P\}e\{Q\}}$$

More later

The **Löb rule** is very convenient for partial correctness:

$$\frac{Q \land \triangleright P \vdash P}{Q \vdash P}$$

Step reductions "consume" laters:

$$\frac{\{P\}e'\{Q\} \qquad e \to e'}{\{\triangleright P\}e\{Q\}}$$

Final definition of triples:

$$\{P\}e\{\Phi\} \;\equiv\; \square(\forall \Psi\ P \mathrel{-\!\!*} \triangleright(\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

Complete set of rules: Lecture Notes on Iris

Invariants

New construct $\boxed{R}^{\iota}$: duplicable, but the resource R is lost at allocation:

$$\boxed{R}^{\iota} \vdash \Box \boxed{R}^{\iota}$$

$$\frac{S, \boxed{R}^{\iota} \vdash \{P\}e\{Q\}}{S \vdash \{\triangleright R * P\}e\{Q\}} \text{ INV-ALLOC}$$

Invariants

New construct $\boxed{R}^{\iota}$: duplicable, but the resource R is lost at allocation:

$$\boxed{R}^{\iota} \vdash \Box \boxed{R}^{\iota} \qquad \frac{S, \boxed{R}^{\iota} \vdash \{P\}e\{Q\}}{S \vdash \{\triangleright R * P\}e\{Q\}}\ \textsc{inv-alloc}$$

Invariant resources can be accessed but must be preserved:

$$\frac{e \text{ is atomic} \qquad S, \boxed{R}^{\iota} \vdash \{\triangleright R * P\}e\{\triangleright R * Q\}}{S, \boxed{R}^{\iota} \vdash \{P\}e\{Q\}}\ \textsc{inv-open}$$

(note: easy to have inconsistent invariants rules, one needs to add stratification not to nest openings)

Locks

Locks can be derived from Compare-And-Swap:

```
let create_lock () = ref false
let acquire p = if CAS p false true then () else acquire p
let release p = p := false
```

Locks

Locks can be derived from Compare-And-Swap:

```
let create_lock () = ref false
let acquire p = if CAS p false true then () else acquire p
let release p = p := false
```

The logic rules can be derived using invariants:

$$p \rightsquigarrow \mathsf{Lock}\,(R, \gamma) \equiv \exists \iota.\, \boxed{p \mapsto \mathsf{true} \;\; \curlyvee \;\; p \mapsto \texttt{false} * R * \gamma \rightsquigarrow \mathsf{Ghost}(K)}^{\iota}$$

with resource algebra $(\varepsilon, K, \bot)$ s.t. $x \cdot \varepsilon = \varepsilon \cdot x = x$ otherwise $x \cdot y = \bot$.

Locks

Locks can be derived from Compare-And-Swap:

```
let create_lock () = ref false
let acquire p = if CAS p false true then () else acquire p
let release p = p := false
```

The logic rules can be derived using invariants:

$$p \rightsquigarrow \mathsf{Lock}\,(R, \gamma) \equiv \exists \iota.\, \boxed{p \mapsto \mathsf{true} \;\; \curlyvee \;\; p \mapsto \texttt{false} * R * \gamma \rightsquigarrow \mathsf{Ghost}(K)}^{\iota}$$

with resource algebra $(\varepsilon, K, \bot)$ s.t. $x \cdot \varepsilon = \varepsilon \cdot x = x$ otherwise $x \cdot y = \bot$.

$$\forall R. \qquad \{R\}\ (\texttt{create\_lock ()})\ \{\lambda p.\ \exists \gamma.\, p \rightsquigarrow \mathsf{Lock}\,(R, \gamma)\}$$

$$\forall pR. \qquad \{p \rightsquigarrow \mathsf{Lock}\,(R, \gamma)\}\ (\texttt{acquire\_lock p})\ \{\lambda\_.\ R * p \rightsquigarrow \mathsf{Lock}\,(R, \gamma)\}$$

$$\forall pR.\ \{R * p \rightsquigarrow \mathsf{Lock}\,(R, \gamma)\}\ (\texttt{release\_lock p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Lock}\,(R, \gamma)\}$$

Conclusion

Some separation logic features:

- tree-like structures, some sharing, abstracting internal structure,
- higher-order representation predicates, first-class functions, local state
- resource management, ghost state, invariants
- concurrency, weak memory models, effect handlers

Going further: lecture notes on Iris