Separation Logic 4/4

Jean-Marie Madiot

Inria Paris

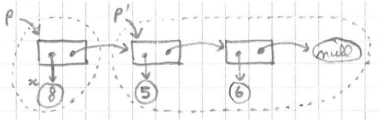
February 9, 2023

part 4/4 : some material from Arthur Charguéraud

Chapter 19

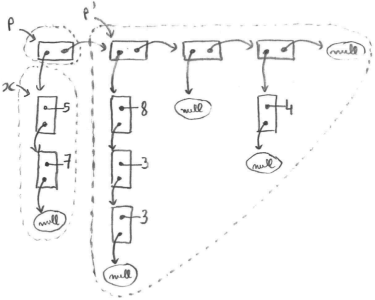
Higher-order representation predicates and the access problem

Specification of construction, for basic values



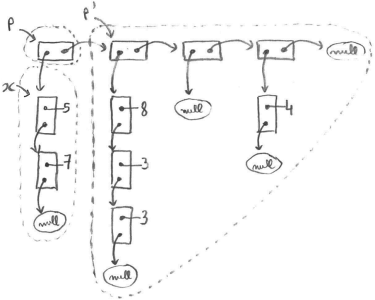
$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ (\mathsf{cons}\, x\, p')\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

Specification of construction



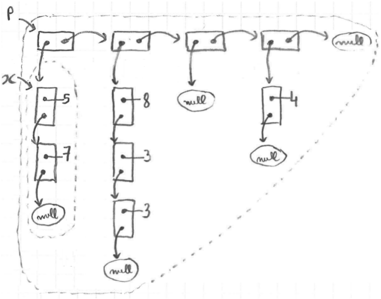
$$\{x \rightsquigarrow R\,X \;*\; p' \rightsquigarrow \mathsf{Mlistof}\,R\,L\}\ (\mathsf{cons}\,x\,p')\ \{\lambda p.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,(X :: L)\}$$

Specification of deconstruction



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

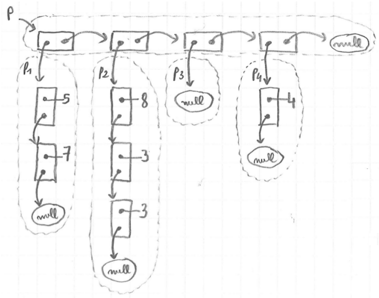
Specification of accesses: the problem



Incorrect specification for `head`:

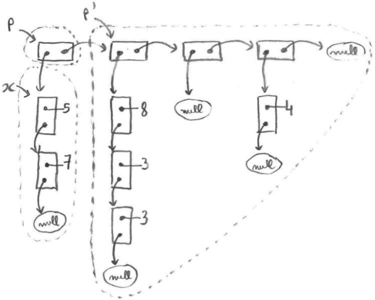
$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}\ (\mathsf{head}\, p)$$
$$\{\lambda x.\ x \rightsquigarrow R\, X\ *\ p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}$$

Specification of accesses: a brute force solution



$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \quad = \quad \exists K. \quad & p \rightsquigarrow \mathsf{MList}\, K \\
* \quad & \circledast_{i \,\in\, \{0,\ldots,|L|-1\}}\ (K[i]) \rightsquigarrow R\,(L[i]) \\
* \quad & \ulcorner |K| = |L| \urcorner
\end{aligned}
$$

Specification of accesses: focus before read

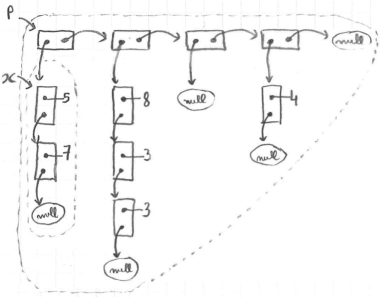


$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L) \;\; = \;\; \exists x p'. \quad & p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
* \quad & x \rightsquigarrow R\, X \\
* \quad & p' \rightsquigarrow \mathsf{Mlistof}\, R\, L
\end{aligned}
$$

Then read using:

$$\{p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}\ (p.\mathsf{hd})\ \{\lambda y.\ \ulcorner y = x \urcorner * p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}$$

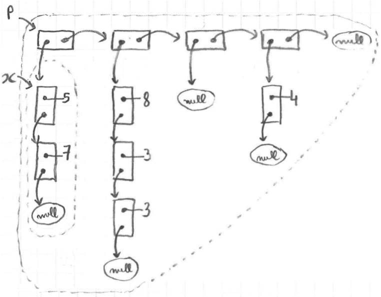
Specification of accesses with separating implication



Correct specification with $-\!\!*$:

$$\{p \leadsto \mathsf{Mlistof}\, R\,(X :: L)\}\ (\texttt{head}\, p)$$
$$\{\lambda x.\ x \leadsto R\,X\ *\ (x \leadsto R\,X\ -\!\!*\ \ p \leadsto \mathsf{Mlistof}\, R\,(X :: L))\}$$

Specification of accesses with separating implication

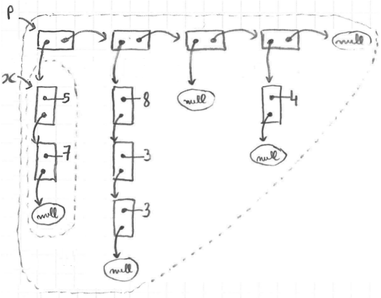


Correct specification with $-\!\!*$:

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

Exercise: What problem is there, e.g. if $x \rightsquigarrow RX$ is $x \mapsto X$ (i.e. $R = \mathsf{Ref}$)?
Exercise: How to generalize the specification to solve this problem?

Specification of accesses with separating implication

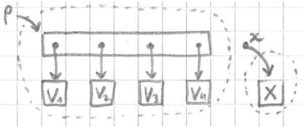


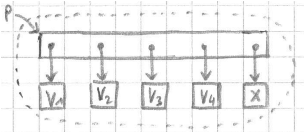
Correct specification with $-\!\!*$, generalized:

$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}\ (\mathtt{head}\, p)$$
$$\{\lambda x.\ x \rightsquigarrow R\, X\ *\ (\forall X', x \rightsquigarrow R\, X'\ -\!\!*\ \ p \rightsquigarrow \mathsf{Mlistof}\, R\, (X' :: L))\}$$

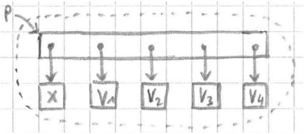
Ownership transfer with a queue of mutable items

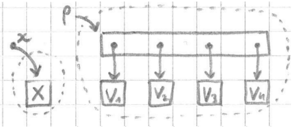
Push:





Pop:





Specification of queues of basic items

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

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

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

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

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 "$\mathsf{Q}\, R$" instead of "$\mathsf{Queueof}\, R$".

Specification of queues of mutable items

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

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

Specification of queues of mutable items

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

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

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

Specification of queues of mutable items

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

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

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

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

Specification of queues of mutable items

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

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

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

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

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

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`:

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

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

The copy problem

Incorrect specification for `copy`:

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

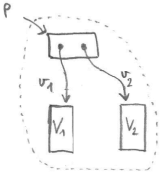
Exercise: specify a function $\mathtt{copy}\, f\, p$ that duplicates a mutable queue specified using Queueof, where f is a function to duplicate items.

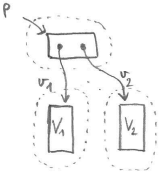
$$\begin{array}{rl} & \big(\forall x X.\ \{x \rightsquigarrow R\, X\}\ (f\, x)\ \{\lambda x'.\ x \rightsquigarrow R\, X \ *\ x' \rightsquigarrow R\, X\}\big) \\ \Rightarrow & \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\} \\ & (\mathsf{copy}\, f\, p) \\ & \{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\,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

Chapter 20

Higher-order representation predicates for records

Representation for records





$$
\begin{aligned}
p \rightsquigarrow \mathsf{MCellof}\, R_1\, V_1\, R_2\, V_2 \;\equiv\; \exists v_1 v_2. \quad & p \rightsquigarrow \{|\mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2|\} \\
* \;\; & v_1 \rightsquigarrow R_1\, V_1 \\
* \;\; & v_2 \rightsquigarrow R_2\, V_2
\end{aligned}
$$

Representation predicate for lists, revisited

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

$$
\begin{array}{rcl}
p \leadsto \mathsf{MCellof}\, R_1\, V_1\, R_2\, V_2 & \equiv & \exists v_1 v_2.\quad p \leadsto \{\!|\mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2|\!\} \\
& & \qquad\quad\ \ast\ \ v_1 \leadsto R_1\, V_1 \\
& & \qquad\quad\ \ast\ \ v_2 \leadsto R_2\, V_2
\end{array}
$$

Exercise: rewrite the specification of Mlistof using MCellof.

Representation predicate for lists, revisited

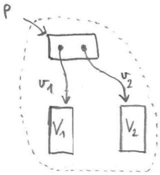
$$
\begin{aligned}
p \leadsto \mathsf{Mlistof}\, R\, L \;\equiv\; & \mathsf{match}\, L\, \mathsf{with} \\
& \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& \mid X :: L' \;\Rightarrow\; \exists x p'.\quad p \leadsto \{\!| \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' |\!\} \\
& \qquad\qquad\qquad\qquad\quad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

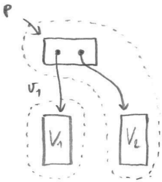
$$
\begin{aligned}
p \leadsto \mathsf{MCellof}\, R_1\, V_1\, R_2\, V_2 \;\equiv\; \exists v_1 v_2.\quad & p \leadsto \{\!| \mathsf{hd}{=}v_1;\ \mathsf{tl}{=}v_2 |\!\} \\
* \;\; & v_1 \leadsto R_1\, V_1 \\
* \;\; & v_2 \leadsto R_2\, V_2
\end{aligned}
$$

Exercise: rewrite the specification of Mlistof using MCellof.

$$
\begin{aligned}
p \leadsto \mathsf{Mlistof}\, R\, L \;\equiv\; & \mathsf{match}\, L\, \mathsf{with} \\
& \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& \mid X :: L' \;\Rightarrow\; p \leadsto \mathsf{MCellof}\, R\, X\, (\mathsf{Mlistof}\, R)\, L'
\end{aligned}
$$

Focus/unfocus for accessing a record field

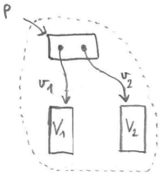


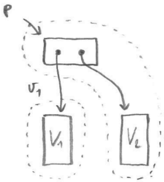


Focus on a field:

$$p \rightsquigarrow \mathsf{MCellof}\, R_1\, V_1\, R_2\, V_2 \;\;=\;\; \exists v_1.\; p \rightsquigarrow \mathsf{MCellof}\, \mathsf{Id}\, v_1\, R_2\, V_2 \;\;*\;\; v_1 \rightsquigarrow R_1\, V_1$$

Focus/unfocus for accessing a record field





Focus on a field:

$$p \rightsquigarrow \mathsf{MCellof}\, R_1\, V_1\, R_2\, V_2 \;=\; \exists v_1.\ p \rightsquigarrow \mathsf{MCellof}\, \mathsf{Id}\, v_1\, R_2\, V_2 \;*\; v_1 \rightsquigarrow R_1\, V_1$$

Access to a focused field:

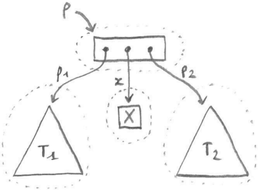
$$\{p \rightsquigarrow \mathsf{MCellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd})\ \{\lambda x.\ \ulcorner x = v_1 \urcorner * p \rightsquigarrow \mathsf{MCellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}$$

$$\{p \rightsquigarrow \mathsf{MCellof}\, \mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd} \texttt{<-}\, w)\ \{\lambda \_.\ p \rightsquigarrow \mathsf{MCellof}\, \mathsf{Id}\, w\, R_2\, V_2\}$$

Chapter 21

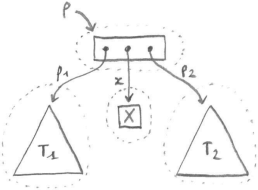
Higher-order representation predicates for trees

Binary tree: representation



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

Binary tree: representation, revisited

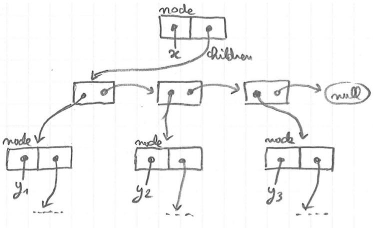


Representation predicate for tree cells:

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Nodeof}\, R_1\, V_1\, R_2\, V_2\, R_3\, V_3 \quad \equiv& \\
\exists v_1 v_2 v_3. \quad & p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{left}{=}v_2;\ \mathsf{right}{=}v_3|\!\} \\
& \ast\ v_1 \rightsquigarrow R_1\, V_1\ \ast\ v_2 \rightsquigarrow R_2\, V_2\ \ast\ v_3 \rightsquigarrow R_3\, V_3
\end{aligned}
$$

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \ \equiv\ & \mathsf{match}\ T\ \mathsf{with} \\
& \mid \mathsf{Leaf}\ \Rightarrow\ \ulcorner p = \mathsf{null} \urcorner \\
& \mid \mathsf{Node}\, X\, T_1\, T_2\ \Rightarrow \\
& \qquad p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mtreeof}\, R)\, T_1\, (\mathsf{Mtreeof}\, R)\, T_2
\end{aligned}
$$

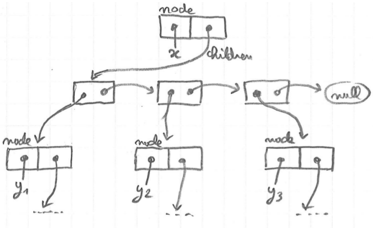
Trees with list of subtrees: implementation




```
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



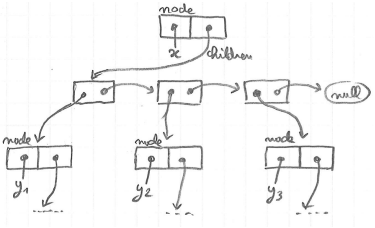
$$
\begin{aligned}
&p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv \\
&\quad \mathsf{match}\, T\, \mathsf{with} \\
&\quad \mid \mathsf{Leaf} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
&\quad \mid \mathsf{Node}\, X\, L \;\Rightarrow\; \exists xc. \quad p \mapsto \{\!|\mathsf{item}{=}x;\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

Trees with list of subtrees: representation of nodes

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Nodeof}\, R_1\, V_1\, R_2\, V_2 \quad \equiv& \\
\exists v_1 v_2.\quad & p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{children}{=}v_2|\!\} \\
& *\ v_1 \rightsquigarrow R_1\, V_1 \\
& *\ v_2 \rightsquigarrow R_2\, V_2
\end{aligned}
$$

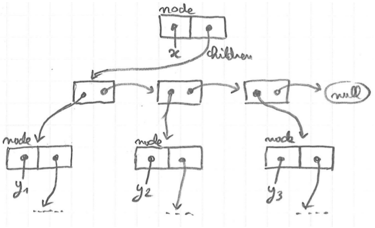
$$
\begin{aligned}
& p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv \\
& \quad \mathsf{match}\ T\ \mathsf{with} \\
& \quad |\ \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& \quad |\ \mathsf{Node}\, X\, L \Rightarrow \exists x c.\quad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\!\}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

Trees with list of subtrees, revisited



Exercise: rewrite the specification of Narytreeof using Nodeof.

Trees with list of subtrees, revisited



Exercise: rewrite the specification of Narytreeof using Nodeof.

$$
\begin{array}{l}
p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv \\
\quad \mathsf{match}\, T\, \mathsf{with} \\
\quad \mid \mathsf{Leaf} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
\quad \mid \mathsf{Node}\, X\; L \;\Rightarrow\; p \rightsquigarrow \mathsf{Nodeof}\, R\, X\; (\mathsf{Mlistof}\,(\mathsf{Narytreeof}\, R))\; L
\end{array}
$$

Chapter 22

Iteration with higher-order representation predicates

Iteration on lists

Recall:

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

$$\begin{aligned} \forall f p l I.\quad & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big) \\ \Rightarrow\quad & \{p \leadsto \mathsf{MList}\,l\ *\ I\,\mathsf{nil}\}\ (\mathtt{miter}\,f\,p)\ \{\lambda\_.\ p \leadsto \mathsf{MList}\,l\ *\ I\,l\} \end{aligned}$$

Iteration on lists

Recall:

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

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

Challenge:

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

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.

$$
\begin{array}{rl}
\forall f p R L J. & \Big(\forall x X K K'.\ \begin{array}[t]{l}\{x \rightsquigarrow R\,X \ * \ J\,K\,K'\} \\ (f\,x) \\ \{\lambda\_.\ \exists X'.\ x \rightsquigarrow R\,X' \ * \ J\,(K\&X)\,(K'\&X')\}\end{array} \Big) \\
\Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L \ * \ J\,\mathsf{nil}\,\mathsf{nil}\} \\
 & (\texttt{miter}\,f\,p) \\
 & \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,L' \ * \ J\,L\,L'\}
\end{array}
$$

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) 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 function `(fun x -> incr x)` and `incr_all`. What is $J\ K\ K'$?

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) p

let example_p =
  { hd = ref 5; tl = { hd = ref 3; tl = null } }
```

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

Exercise: using the representation predicates Ref and Mlistof, specify the function `(fun x -> incr x)` and `incr_all`. What is $J\ K\ K'$?

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

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) 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 MIistof, specify the function `(fun x -> incr x)` and `incr_all`. What is $J\ K\ K'$?

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

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

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

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

Consider:

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

Derives:

$$\big(\forall x X.\ \{x \rightsquigarrow \mathsf{Ref}\,X\}\ (\mathtt{fun\ x\ \text{->}\ incr\ x})x\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}\big) \Rightarrow$$

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

Chapter 23

Resource analysis in Separation Logic

Controlling deallocation

(1) Remove the garbage collection rule:

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post}$$

(2) Add a "free" function for explicit deallocation:

$$\{r \mapsto v\}\ (\texttt{free r})\ \{\lambda\_.\ \ulcorner\urcorner\}$$

Controlling deallocation

(1) Remove the garbage collection rule:

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post}$$

(2) Add a "free" function for explicit deallocation:

$$\{r \mapsto v\}\ (\texttt{free r})\ \{\lambda\_.\ \ulcorner\urcorner\}$$

(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 garbage collection rule:

$$\frac{\{H\}\; t\; \{Q \star \mathsf{GC}\}}{\{H\}\; t\; \{Q\}}\ \textsc{gc-post}$$

(2) Add a "free" function for explicit deallocation:

$$\{r \mapsto v\}\ (\texttt{free r})\ \{\lambda\_.\ \ulcorner\urcorner\}$$

(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.

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

(note: a separation logic with GC can be 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 $(L : \mathsf{list}\, \mathsf{char})$ denotes the remaining bytes to read.

File handle protocols

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

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

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

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

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.

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

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 $(L : \mathsf{list}\,\mathsf{char})$ denotes the remaining bytes to read.

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

Complexity analysis

Time credits:

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

Properties:

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

Complexity analysis

Time credits:

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

Properties:

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

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:

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

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:

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

Time credits in pre-conditions

Constant time:

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

Linear time:

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

Quasilinear time:

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

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$$\{\textcolor{red}{\$\,c}\}\quad (\texttt{Stack.create()})\quad \{\lambda\_.\, s \leadsto \mathsf{Stack}\,\mathsf{nil}\}$$

$$\{s \leadsto \mathsf{Stack}\,L \,*\, \textcolor{red}{\$\,c}\}\quad (\texttt{Stack.push s x})\quad \{\lambda\_.\, s \leadsto \mathsf{Stack}\,(x :: L)\}$$

$$\{s \leadsto \mathsf{Stack}\,(x :: L) \,*\, \textcolor{red}{\$\,c}\}\quad (\texttt{Stack.pop s})\quad \{\lambda y.\ \ulcorner y = x \urcorner \,*\, s \leadsto \mathsf{Stack}\,L\}$$

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

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

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

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

Representation predicate with a potential function:

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

Space complexity analysis

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

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

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

The same mechanisms apply (e.g. $\Diamond(a+b) = \Diamond a * \Diamond b$, 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:

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

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

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

Fractional permissions in practice

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

Fractional permissions in practice

$$\begin{array}{l}\forall \alpha\beta.\ \{a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ *\ a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\} \\ \qquad (\mathtt{concat}\, a_1\, a_2) \\ \qquad \{\lambda a_3.\ a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ *\ a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 * a_3 \overset{1}{\rightsquigarrow} \mathsf{Array}\,(L_1 +\!\!+ L_2)\}\end{array}$$

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 $\text{let}\,((),()) = (|t_1, t_2|)\,\text{in}\,()$.)

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

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

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
```

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
```

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

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

Concurrent locks: specification of operations

Duplicable representation predicate:

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

Operations:

$$\begin{aligned}
\forall H.\quad & \{H\}\ (\texttt{create_lock ()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Lock}\, H\} \\
\forall p H.\quad & \{p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{acquire_lock p})\ \{\lambda_.\ H * p \rightsquigarrow \mathsf{Lock}\, H\} \\
\forall p H.\ & \{H\ *\ p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{release_lock p})\ \{\lambda_.\ p \rightsquigarrow \mathsf{Lock}\, H\}
\end{aligned}$$

Concurrent locks: exercise

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

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

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

Exercise: Describe the state before each instruction (except line 5).
Explicit the instantiation of the existential in the invariant.

```
1    let r = ref 0
2    let s = ref n
3    let p = create_lock ()
4
5    let concurrent_step () =
6      acquire_lock p;
7      incr r;
8      decr s;
9      release_lock p
```

Concurrent locks: exercise

Exercise: Describe the state before each instruction (except line 5). Explicit the instantiation of the existential in the invariant.

```
1    let r = ref 0
2    let s = ref n
3    let p = create_lock ()
4
5    let concurrent_step () =
6       acquire_lock p;
7       incr r;
8       decr s;
9       release_lock p
```

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

Concurrent locks: 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)
```

Chapter 25

Ghost state

Same non-example

```
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()

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()

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 $\{\text{True}\}$ program $\{\text{True}\}$)

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 $\{\text{True}\}$ program $\{\text{True}\}$ (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()
```

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

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()
```

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

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

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

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

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

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

```
acquire_lock p;
```

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

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

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

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

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;
```

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

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

```
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;
```

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

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

```
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;
```

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

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

```
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;
```

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

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

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

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

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

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

```
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;
```

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

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

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

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

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

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

```
release_lock p;
```

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

Right thread

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

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

```
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()
```

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

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

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

||

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

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

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

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

```
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?
- how do we know adding variables really preserves the semantics?
- 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?
- how do we know adding variables really preserves the semantics?
- 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":

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

Some remarks

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

- same example with an arbitrary number of threads?
- how do we know adding variables really preserves the semantics?
- 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?
- how do we know adding variables really preserves the semantics?
- 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

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

Iris hypotheses go in contexts and can be named, split, rearranged...

```
                                              H1 : P1
                        H : P1 * P2           H2 : P2
P1 * P2 -* wp e Φ   →   ----------*     →     ------*
                        wp e Φ                 wp e Φ
```

Builtin consequence rule in postconditions

$\mathsf{wp}\, e\, \cdot$ 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

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

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

So instead of choosing

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

the following formulation is preferred:

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

Specifications can be applied directly since Ψ is universally quantified.

Note that separating implication $\mathrel{-\!\!*}$ has no easy "heap entailment" $\rhd$ counterpart here.

Simplified rules for load, store, alloc

Exercise:

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

Simplified rules for load, store, alloc

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

Exemple

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

Exemple

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

in other words:

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

Exemple

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

in other words:

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

after `iIntros (Φ) "Hl HΦ"` you have:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l' : loc, l ↦ #1 ∗ l' ↦ #3 -∗ Φ #l'
--------------------------------------∗
WP ref #3 {{ v, Φ v }}
```

Exemple

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

in other words:

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

after `iIntros (Φ) "Hl HΦ"` you have:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l' : loc, l ↦ #1 ∗ l' ↦ #3 -∗ Φ #l'
---------------------------------------∗
WP ref #3 {{ v, Φ v }}
```

applying the rule for allocation, we get:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l' : loc, l ↦ #1 ∗ l' ↦ #3 -∗ Φ #l'
---------------------------------------∗
∀ l0 : loc, l0 ↦ #3 -∗ Φ #l0
```

after `iIntros (l') "Hl'"` we get:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l'0 : loc, l ↦ #1 ∗ l'0 ↦ #3 -∗ Φ #l'0
"Hl'" : l' ↦ #3
--------------------------------------∗
Φ #l'
```

after `iIntros (l') "Hl'"` we get:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l'0 : loc, l ↦ #1 ∗ l'0 ↦ #3 -∗ Φ #l'0
"Hl'" : l' ↦ #3
---------------------------------------∗
Φ #l'
```

after `iApply "HΦ"` we get:

```
"Hl" : l ↦ #1
"Hl'" : l' ↦ #3
---------------------------------------∗
l ↦ #1 ∗ l' ↦ #3
```

after `iIntros (l') "Hl'"` we get:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l'0 : loc, l ↦ #1 ∗ l'0 ↦ #3 -∗ Φ #l'0
"Hl'" : l' ↦ #3
--------------------------------------∗
Φ #l'
```

after `iApply "HΦ"` we get:

```
"Hl" : l ↦ #1
"Hl'" : l' ↦ #3
--------------------------------------∗
l ↦ #1 ∗ l' ↦ #3
```

`iFrame` solves the goal.

after `iIntros (l') "Hl'"` we get:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l'0 : loc, l ↦ #1 ∗ l'0 ↦ #3 -∗ Φ #l'0
"Hl'" : l' ↦ #3
--------------------------------------∗
Φ #l'
```

after `iApply "HΦ"` we get:

```
"Hl" : l ↦ #1
"Hl'" : l' ↦ #3
--------------------------------------∗
l ↦ #1 ∗ l' ↦ #3
```

`iFrame` solves the goal.

⚠ Iris is an affine logic, it can throw away hypotheses. ⚠

Chapter 27: modalities

Persistently modality $\Box$

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

$\Box P \vartriangleright P$

Persistently modality $\Box$

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

$\Box P \vartriangleright P \qquad \Box P \vartriangleright \Box P * P$

Persistently modality $\Box$

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

$\Box P \vdash P \qquad \Box P \vdash \Box P * P \qquad \Box P \vdash \Box P * \Box P$

Persistently modality $\Box$

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

$\Box P \vdash P \qquad \Box P \vdash \Box P * P \qquad \Box P \vdash \Box P * \Box P \qquad \Box P \vdash \Box P * P * P$

Persistently modality $\Box$

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

$\Box P \vdash P \qquad \Box P \vdash \Box P * P \qquad \Box P \vdash \Box P * \Box P \qquad \Box P \vdash \Box P * P * P$

$\ulcorner P \urcorner \vdash \Box \ulcorner P \urcorner$

Persistently modality $\square$

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

$\square P \vdash P \qquad \square P \vdash \square P * P \qquad \square P \vdash \square P * \square P \qquad \square P \vdash \square P * P * P$

$\ulcorner P \urcorner \vdash \square \ulcorner P \urcorner \qquad \square(\ell \mapsto 1) \vdash \ulcorner \text{False} \urcorner$

Persistently modality $\square$

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

$\square P \vdash P \qquad \square P \vdash \square P * P \qquad \square P \vdash \square P * \square P \qquad \square P \vdash \square P * P * P$

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

Persistently modality $\square$

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

$\square P \vartriangleright P \qquad \square P \vartriangleright \square P * P \qquad \square P \vartriangleright \square P * \square P \qquad \square P \vartriangleright \square P * P * P$

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

$$\{P\}e\{\Phi\} \ \equiv\ \square(\forall \Psi\ P \twoheadrightarrow (\forall v\ \Phi v \twoheadrightarrow \Psi v) \twoheadrightarrow \mathsf{wp}\, e\, \Psi)$$

Persistently modality $\Box$

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

$$\Box P \vdash P \qquad \Box P \vdash \Box P * P \qquad \Box P \vdash \Box P * \Box P \qquad \Box P \vdash \Box P * P * P$$

$\ulcorner P \urcorner \vdash \Box \ulcorner P \urcorner$ $\qquad$ $\Box(\ell \mapsto 1) \vdash \ulcorner \mathsf{False} \urcorner$ $\qquad$ $\Box(\ell \overset{q}{\mapsto} 1) \vdash \ulcorner \mathsf{q{=}0} \urcorner$ (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:

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

Persistently modality $\square$

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

$$\square P \vartriangleright P \qquad \square P \vartriangleright \square P * P \qquad \square P \vartriangleright \square P * \square P \qquad \square P \vartriangleright \square P * P * P$$

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

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

Persistent resources can be shared between threads:

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

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

$\triangleright^n \text{False} \vdash P$ 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 $\Box$ are modalities in Iris, where "`iProp`" are not "`heap -> Prop`" and have features such as step indexing, and resources are not only heaps.

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

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:

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

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:

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

Step reductions "consume" laters:

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

Final definition of triples:

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

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:

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

Invariant resources can be accessed but must be preserved:

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

(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:

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

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:

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

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

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

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

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

Conclusion

Some separation logic features:

- tree-like structures, some sharing,
- abstracting intermediate pointers, internal structure
- higher-order representation predicates
- first-class functions, local state
- ghost state, invariants
- concurrency (rich literature, including weak memory models)
- effects (rich literature here too, e.g. effect handlers)

Bibliography: comprehensive lecture notes on Iris

More slides if more time

Chapter 27

Read-only permissions

Motivation for read-only permissions

What we currently need to write:

$$\{a_1 \rightsquigarrow \mathsf{Array}\, L_1 \,\ast\, a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$
$$(\mathtt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\,(L_1 \mathbin{+\!\!+} L_2) \,\ast\, a_1 \rightsquigarrow \mathsf{Array}\, L_1 \,\ast\, a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$

Motivation for read-only permissions

What we currently need to write:

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

What we wish to write:

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

Motivation for read-only permissions

What we currently need to write:

$$\{a_1 \rightsquigarrow \mathsf{Array}\, L_1 \;*\; a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\; a_3 \rightsquigarrow \mathsf{Array}\, (L_1 \mathbin{+\!\!+} L_2) \;*\; a_1 \rightsquigarrow \mathsf{Array}\, L_1 \;*\; a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$

What we wish to write:

$$\{a_1 \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Array}\, L_1 \;*\; a_2 \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\; a_3 \rightsquigarrow \mathsf{Array}\, (L_1 \mathbin{+\!\!+} L_2)\}$$

More than syntactic sugar:
– we wish "ro" to enforce no write operations,
– we wish to allow aliasing of read-only arguments.

Generic read-only modifier

Extension of the logic with a modifier $\mathsf{RO}(H)$ that applies to any H.

$$a \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Array}\, L \;\equiv\; \mathsf{RO}(a \rightsquigarrow \mathsf{Array}\, L)$$

$\mathsf{RO}(H)$ is duplicable and never mentioned in post-conditions.

$$\frac{}{\mathsf{RO}(H) \;\;\rhd\;\; \mathsf{RO}(H) \;*\; \mathsf{RO}(H)}\ \textsc{dup-ro}$$

$$\frac{}{\{\mathsf{RO}(l \mapsto v)\}\ (\mathsf{get}\, l)\ \{\lambda x.\ \ulcorner x = v \urcorner\}}\ \textsc{get-ro}$$

Read-only frame rule

$\mathsf{RO}(H)$ is introduced on frame:

$$\frac{\{H * \mathsf{RO}(H')\}\ t\ \{Q\} \qquad \mathsf{no\text{-}ro\text{-}in}\ H'}{\{H * H'\}\ t\ \{Q \star H'\}}\ \textsc{frame-ro}$$

Read-only sequencing rule

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'()\}\ t_2\ \{Q\}}{\{H\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \textsc{seq}$$

$$\frac{\{H \ast \mathsf{RO}(H')\}\ t_1\ \{Q'\} \qquad \{Q'() \ast \mathsf{RO}(H')\}\ t_2\ \{Q\}}{\{H \ast \mathsf{RO}(H')\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \textsc{seq-ro}$$

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \{Q'() \ast H'\}\ t_2\ \{Q\}}{\{H \ast H'\}\ (t_1\ ;\ t_2)\ \{Q\}}\ \textsc{seq-frame}$$

RO in practice

$$
\begin{array}{l}
\{\mathsf{RO}(a_1 \rightsquigarrow \mathsf{Array}\, L_1) \;*\; \mathsf{RO}(a_2 \rightsquigarrow \mathsf{Array}\, L_2)\} \\
(\texttt{concat}\, a_1\, a_2) \\
\{\lambda a_3.\;\, a_3 \rightsquigarrow \mathsf{Array}\,(L_1 \mathbin{+\!\!+} L_2)\}
\end{array}
$$

Parallel rule needs read-only permissions

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

Compute: $u[a[0]] + u[a[1]] + ... + u[a[n-1]]$.

```
map_reduce (fun x -> u.(a.(x))) 0 (+) 0 n
```

The ownership of the array `u` is needed in both branches.

Parallel rule needs read-only permissions

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

Compute: $u[a[0]] + u[a[1]] + ... + u[a[n-1]]$.

```
map_reduce (fun x -> u.(a.(x))) 0 (+) 0 n
```

The ownership of the array `u` is needed in both branches.

$$\frac{\{H_1 * \mathsf{RO}(H_3)\}\ t_1\ \{Q_1\} \qquad \{H_2 * \mathsf{RO}(H_3)\}\ t_2\ \{Q_2\}}{\{H_1 * H_2 * \mathsf{RO}(H_3)\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \textsc{parallel-ro}$$