# Separation Logic 4/4

Jean-Marie Madiot

Inria Paris

February 10, 2021

slides (mostly) from Arthur Charguéraud





**Before we start:**

- Don't check "listen only": activate your microphone!
- For the first few minutes: show me your faces (if you reasonably can)

Also, an ad for an internship:

*A formal proof of correctness of tail-recursion modulo constructor in the Iris logic*
with: Léo Stefanesco, François Pottier, Gabriel Scherer, Frédéric Bour
`https://www.stefanesco.com/documents/internship-trmc.pdf`



# Chapter 18

Characteristic Formulas with structural rules



## Integration of structural rules

$$
\begin{aligned}
[\![v]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ H \rhd Q\,v) \\
[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ [\![t_1]\!]\,H\,Q' \\
&\qquad\qquad \wedge \forall x.\ [\![t_2]\!]\,(Q'\,x)\,Q) \\
[\![\mathsf{if}\,b\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ (b = \mathsf{true} \Rightarrow [\![t_1]\!]\,H\,Q)\ \wedge\ (b = \mathsf{false} \Rightarrow [\![t_2]\!]\,H\,Q)) \\
[\![v_1\,v_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ \mathsf{App}\,v_1\,v_2\,H\,Q) \\
[\![\mathsf{let\,rec}\,f = \lambda x.\,t_1\,\mathsf{in}\,t_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ \forall f.\ Pf \Rightarrow [\![t_2]\!]\,H\,Q)
\end{aligned}
$$

where $Pf \equiv (\forall x H' Q'.\ [\![t_1]\!]\,H'\,Q' \Rightarrow \mathsf{App}\,f\,x\,H'\,Q')$

## Definition of the local predicate (1/2)

To support:

$$\frac{H = H_1 \star H_2 \qquad [\![t]\!]\, H_1\, Q_1 \qquad Q_1 \star H_2 = Q}{[\![t]\!]\, H\, Q}\ \textsc{frame'}$$

we would define:

$$\mathsf{local}\,\mathcal{F} \;\equiv\; \lambda H Q.\ \exists H_1 H_2 Q_1. \begin{cases} H = H_1 \star H_2 \\ \mathcal{F}\, H_1\, Q_1 \\ Q_1 \star H_2 = Q \end{cases}$$



## Framing using the local predicate

To prove "$\{H\}\ t\ \{Q\}$", by the rule FRAME', it suffices to show:

$$H = H_1 \star H_2 \;\wedge\; \{H_1\}\ t\ \{Q_1\} \;\wedge\; Q_1 \star H_2 = Q$$

To prove "local $[\![t]\!]\, H\, Q$", by definition of "local", it suffices to show:

$$H = H_1 \star H_2 \;\wedge\; [\![t]\!]\, H_1\, Q_1 \;\wedge\; Q_1 \star H_2 = Q$$



## Definition of the local predicate (2/2)

To support:

$$\frac{H \rhd H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \rhd Q \star \mathsf{GC}}{\{H\}\ t\ \{Q\}}\ \textsc{combined}$$

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\, H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P \Rightarrow \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textsc{prop}$$

we define:

$$\mathsf{local}\,\mathcal{F} \;\equiv\; \lambda H Q.\ \forall h.\ H\, h \Rightarrow \exists H_1 H_2 Q_1. \begin{cases} (H_1 \star H_2)\, h \\ \mathcal{F}\, H_1\, Q_1 \\ Q_1 \star H_2 \rhd Q \star \mathsf{GC} \end{cases}$$



## Iterated applications of structural rules

The local predicate may be duplicated as many times as needed:

$$\mathsf{local}\,[\![t]\!]\, H\, Q \;=\; \mathsf{local}\,(\mathsf{local}\,[\![t]\!])\, H\, Q$$

For example, to prove "local $[\![t]\!]\, H\, Q$", it suffices to show:

$$H = H_1 \star H_2 \;\wedge\; \mathsf{local}\,[\![t]\!]\, H_1\, Q_1 \;\wedge\; Q_1 \star H_2 = Q$$

When not needed, "local" may be simply erased:

$$[\![t]\!]\, H\, Q \;\Rightarrow\; \mathsf{local}\,[\![t]\!]\, H\, Q$$

## Notation for characteristic formulas

$$\llbracket \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 \rrbracket \;\equiv\; \mathsf{local}\,(\lambda H Q.\;\; \exists Q'.\;\; \llbracket t_1 \rrbracket\, H\, Q' \;\wedge\; \forall x.\;\; \llbracket t_2 \rrbracket\, (Q'\, x)\, Q)$$

Definition of Coq notation:

$$(\mathsf{Let}\, x = \mathcal{F}_1 \,\mathsf{in}\, \mathcal{F}_2) \;\equiv\; \mathsf{local}\,(\lambda H Q.\;\; \exists Q'.\;\; \mathcal{F}_1\, H\, Q' \;\wedge\; \forall x.\;\; \mathcal{F}_2\, (Q'\, x)\, Q)$$

With this notation:

$$\llbracket \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 \rrbracket \;\equiv\; (\mathsf{Let}\, x = \llbracket t_1 \rrbracket \,\mathsf{in}\, \llbracket t_2 \rrbracket)$$

Technically:

$$\llbracket \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 \rrbracket \;\equiv\; (\mathsf{Let}\, X = \llbracket t_1 \rrbracket \,\mathsf{in}\, \llbracket ([x \to X]\, t_2) \rrbracket)$$



## Characteristic formulas generation, with notation

$$\begin{array}{lcl}
\llbracket v \rrbracket & \equiv & \mathsf{Ret}\; v \\
\llbracket \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 \rrbracket & \equiv & \mathsf{Let}\; x = \llbracket t_1 \rrbracket \;\mathsf{in}\; \llbracket t_2 \rrbracket \\
\llbracket \mathsf{if}\, b \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2 \rrbracket & \equiv & \mathsf{If}\; b \;\mathsf{then}\; \llbracket t_1 \rrbracket \;\mathsf{else}\; \llbracket t_2 \rrbracket \\
\llbracket v_1\, v_2 \rrbracket & \equiv & \mathsf{App}\; v_1\; v_2 \\
\llbracket \mathsf{let\,rec}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2 \rrbracket & \equiv & \mathsf{Let\,Rec}\; f\; x \;=\; \llbracket t_1 \rrbracket \;\mathsf{in}\; \llbracket t_2 \rrbracket
\end{array}$$



## Tactics for characteristic formulas

What the user sees:

$$\mathsf{Let}\; x = \mathcal{F}_1 \;\mathsf{in}\; \mathcal{F}_2$$

What is hidden behind the notation:

$$\mathsf{local}\,(\lambda H Q.\;\; \exists Q'.\;\; \mathcal{F}_1\, H\, Q' \;\wedge\; \forall x.\;\; \mathcal{F}_2\, (Q'\, x)\, Q)$$

What the user would need to execute:

```
apply local_erase; esplit; split.
```

What the user writes:

```
xlet.
```



# Chapter 19

Higher-order representation predicates

## Overview

1. Higher-order predicate:

   $p \rightsquigarrow \mathsf{MList}\, L$ is generalized into $p \rightsquigarrow \mathsf{MlistofRL}$

2. Identity representation predicate:

   $p \rightsquigarrow \mathsf{MlistofId}\, L$ is the same as $p \rightsquigarrow \mathsf{MList}\, L$

3. Control accesses:

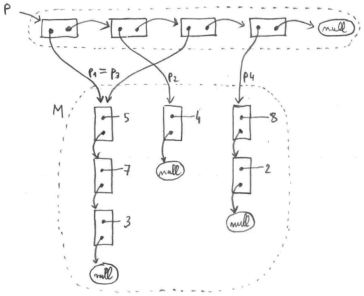
   $\{p \rightsquigarrow \mathsf{MCellofId}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd})\ \{\lambda x.\, [x = v_1] \star ...\}$

4. Compose recursively:

   $p \rightsquigarrow \mathsf{NodeofRXMlistof(Narytreeof}\, R)\, L$



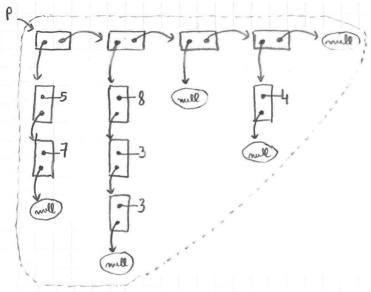
## Mutable list of possibly-aliased lists




$$p \rightsquigarrow \mathsf{MList}\, K \ \star \left( \circledast_{(p_i, L_i) \in M} \ p_i \rightsquigarrow \mathsf{MList}\, L_i \right) \star [\forall p_i \in K.\ p_i \in \mathsf{dom}\, M]$$



## Mutable list of disjoint mutable lists



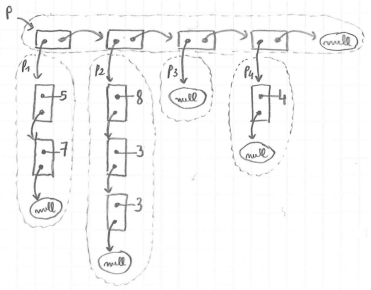

$$L \ = \ (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) :: \mathsf{nil}$$

$p \rightsquigarrow \mathsf{MlistofMlist}\, L$

(to be later generalized into: $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$)



## Representation using iterated star

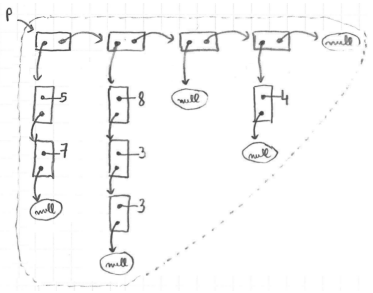



$$L \ = \ (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) :: \mathsf{nil}$$

$$K \ = \ p_1 :: p_2 :: p_3 :: p_4 :: \mathsf{nil}$$

$$p \rightsquigarrow \mathsf{MlistofMlist}\, L \quad \equiv \quad \exists K.\quad p \rightsquigarrow \mathsf{MList}\, K \ \star \ \circledast_{i \in [0, |L|)}\ (K[i]) \rightsquigarrow \mathsf{MList}\, (L[i]) \ \star \ [\, |K| = |L| \,]$$

## Representation using a recursive predicate



$$L \;=\; (5::7::\mathsf{nil})::(8::3::3::\mathsf{nil}) \\ ::(\mathsf{nil})::(4::\mathsf{nil})::\mathsf{nil}$$

$$\begin{array}{l} p \rightsquigarrow \mathsf{MlistofMlist}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\ \quad |\, \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ \quad |\, X :: L' \;\Rightarrow\; \exists x p'.\;\; p \rightsquigarrow \{|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\} \\ \qquad\qquad \star\; p' \rightsquigarrow \mathsf{MlistofMlist}\, L' \\ \qquad\qquad \star\; x \rightsquigarrow \mathsf{MList}\, X \end{array}$$



## Generalization to a higher-order predicate

$$\begin{array}{l} p \rightsquigarrow \mathsf{MlistofMlist}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\ \quad |\, \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ \quad |\, X :: L' \;\Rightarrow\; \exists x p'.\;\; p \rightsquigarrow \{|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\} \\ \qquad\qquad \star\; p' \rightsquigarrow \mathsf{MlistofMlist}\, L' \\ \qquad\qquad \star\; x \rightsquigarrow \mathsf{MList}\, X \end{array}$$

Generalization:

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

In particular:

$$p \rightsquigarrow \mathsf{MlistofMlist}\, L \;=\; p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{MList}\, L$$



## Type-checking

$p \rightsquigarrow \mathsf{Mlistof}\, R\, L$ *is a notation for* $\mathsf{Mlistof}\, R\, L\, p$ (of type Hprop)
$x \rightsquigarrow R\, X$ *is a notation for* $R\, X\, x$ (of type Hprop)

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

**Exercise:** since $(p : \mathsf{loc})$ and $(x : \mathsf{Val})$ and $(X : A)$ for some $A$, what is the type of $R$? What is the type of Mlistof?

- $R : A \to \mathsf{Val} \to \mathsf{Hprop}$
- $\mathsf{Mlistof} : \forall A.\; (A \to \mathsf{Val} \to \mathsf{Hprop}) \to \mathsf{list} A \to \mathsf{loc} \to \mathsf{Hprop}$



## The identity representation predicate

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

$$\begin{array}{l} p \rightsquigarrow \mathsf{MList}\, L \;\equiv\; \mathsf{match}\, L\, \mathsf{with} \\ \quad |\, \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ \quad |\, x :: L' \;\Rightarrow\; \exists p'.\;\; p \rightsquigarrow \{|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\} \\ \qquad\qquad \star\; p' \rightsquigarrow \mathsf{MList}\, L' \end{array}$$

**Exercise:** define the identity representation predicate Id such that

$$p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L \;=\; p \rightsquigarrow \mathsf{MList}\, L$$

Definition:

$$x \rightsquigarrow \mathsf{Id}\, X \;\equiv\; [x = X]$$

## Summary

1. Higher-order predicate:

   $p \rightsquigarrow \mathsf{MList}\, L$ is generalized into $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$

2. Identity representation predicate:

   $p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L$ is the same as $p \rightsquigarrow \mathsf{MList}\, L$

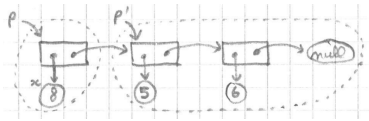


## Chapter 20

Higher-order representation predicates and the access problem



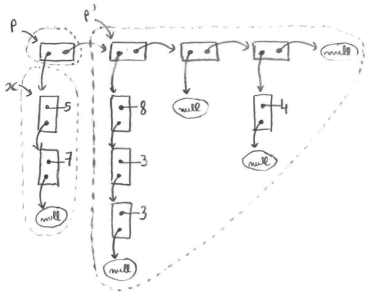
## Specification of construction, for basic values




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

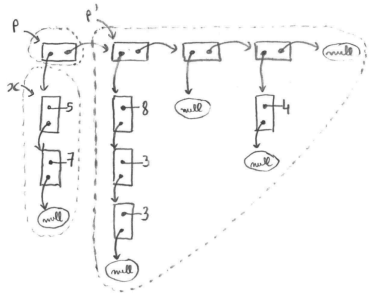


## Specification of construction




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

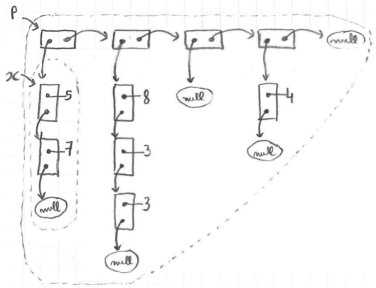
## Specification of deconstruction



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



## Specification of accesses: the problem

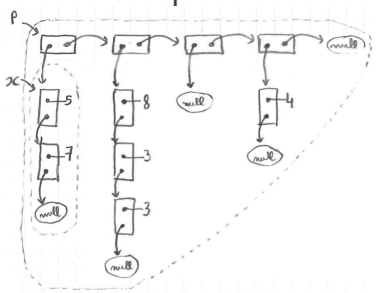


Incorrect specification for `head`:

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



## Specification of accesses: a partial solution



Correct yet limited specification:

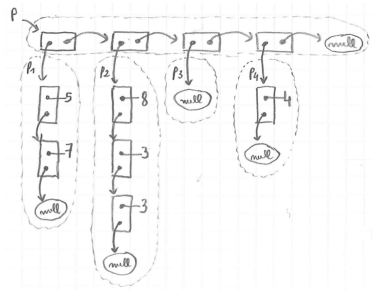
$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}$ $(\mathtt{head}\, p)$
$\{\lambda x.\ x \rightsquigarrow R\, X \ \star\ (x \rightsquigarrow R\, X \ \mathrel{-\!\!\star}\ p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L))\}$

An intuition for $(\mathrel{-\!\!\star})$ is: $(H \ \star\ (H \mathrel{-\!\!\star} H')) \rhd H'$. More precisely:

$$\frac{H_1 \star H_2 \vdash H_3}{H_1 \vdash (H_2 \mathrel{-\!\!\star} H_3)} \qquad \frac{H_1 \vdash (H_2 \mathrel{-\!\!\star} H_3) \qquad H_4 \vdash H_2}{H_1 \star H_4 \vdash H_3}$$

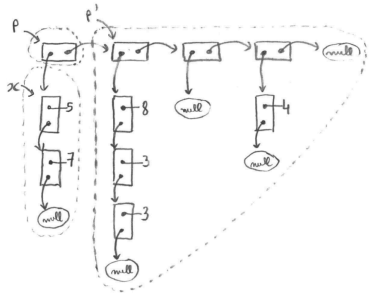


## Specification of accesses: a brute force solution



$p \rightsquigarrow \mathsf{Mlistof}\, R\, L \quad = \quad \exists K.\quad p \rightsquigarrow \mathsf{MList}\, K$
$\star\ \circledast_{i \in [0, |L|)}\ (K[i]) \rightsquigarrow R\, (L[i])$
$\star\ [\, |K| = |L|\, ]$

## Specification of accesses: focus before read



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

Then read using:

$$\{p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}\ (p.\mathsf{hd})\ \{\lambda y.\ [y = x] \star p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}$$



## Ownership transfer with a queue of mutable items

Push:

Pop:



## Specification of queues of basic items

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

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

$\{p \rightsquigarrow \mathsf{Queue}\, (x :: L)\}\ (\texttt{pop p})\ \{\lambda r.\ [r = x] \star p \rightsquigarrow \mathsf{Queue}\, L\}$

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



## 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 $R$".

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

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

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

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

## The copy problem

Incorrect specification for copy:

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

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

$$\big(\forall x X.\ \{x \rightsquigarrow R\, X\}\ (f\, x)\ \{\lambda x'.\ x \rightsquigarrow R\, X \star x' \rightsquigarrow R\, X\}\big)$$
$$\Rightarrow \{p \rightsquigarrow \mathsf{Queueof}\, R\, L\}$$
$$(\mathsf{copy}\, f\, p)$$
$$\{\lambda p'.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \star p' \rightsquigarrow \mathsf{Queueof}\, R\, L\}$$



# Chapter 21

Higher-order representation predicates for records



## Representation for records


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



## Representation predicate for lists, revisited

$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ \equiv\ \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \mid X :: L' \Rightarrow \exists x p'.\quad p \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} \star x \rightsquigarrow R\, X \star p' \rightsquigarrow \mathsf{Mlistof}\, R\, L'$$

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

**Exercise:** rewrite the specification of Mlistof using MCellof.

$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ \equiv\ \mathsf{match}\, L\, \mathsf{with} \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \mid X :: L' \Rightarrow p \rightsquigarrow \mathsf{MCellof}\, R\, X\, (\mathsf{Mlistof}\, R)\, L'$$

## Focus/unfocus for accessing a record field

Focus on a field:

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

Access to a focused field:

$\{p \rightsquigarrow \mathsf{MCellof}\,\mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd})\ \{\lambda x.\, [x = v_1] \star p \rightsquigarrow \mathsf{MCellof}\,\mathsf{Id}\, v_1\, R_2\, V_2\}$
$\{p \rightsquigarrow \mathsf{MCellof}\,\mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd} \mathrel{<\!\!-} w)\ \{\lambda\_.\; p \rightsquigarrow \mathsf{MCellof}\,\mathsf{Id}\, w\, R_2\, V_2\}$

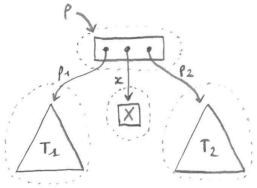


# Chapter 22

Higher-order representation predicates for trees



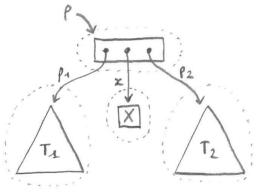
## Binary tree: representation



$p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \;\equiv\; \mathsf{match}\, T\, \mathsf{with}$
$\quad \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]$
$\quad \mid \mathsf{Node}\, X\, T_1\, T_2 \;\Rightarrow\; \exists x p_1 p_2.$
$\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\}$
$\qquad \star\; x \rightsquigarrow R\, X$
$\qquad \star\; p_1 \rightsquigarrow \mathsf{Mtreeof}\, R\, T_1$
$\qquad \star\; p_2 \rightsquigarrow \mathsf{Mtreeof}\, R\, T_2$



## Binary tree: representation, revisited

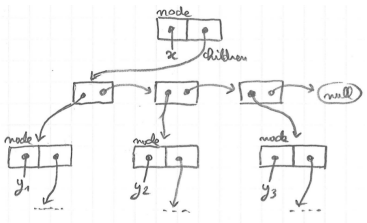


Representation predicate for tree cells:

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

$p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \;\equiv\; \mathsf{match}\, T\, \mathsf{with}$
$\quad \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]$
$\quad \mid \mathsf{Node}\, X\, T_1\, T_2 \;\Rightarrow$
$\qquad p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mtreeof}\, R)\, T_1\, (\mathsf{Mtreeof}\, R)\, T_2$

## Trees with list of subtrees: implementation

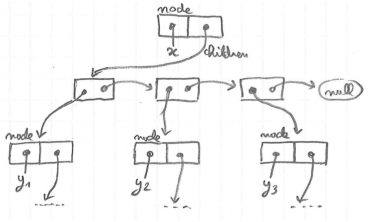



```
type 'a node = {
  mutable item : 'a;
  mutable children : ('a node) cell }

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



## Trees with list of subtrees: specification




$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$
  $\mathsf{match}\ T\ \mathsf{with}$
  $|\ \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]$
  $|\ \mathsf{Node}\ X\ L \Rightarrow \exists xc.\quad p \mapsto \{|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\}$
  $\star\ x \rightsquigarrow R\ X$
  $\star\ c \rightsquigarrow \mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R)\ L$



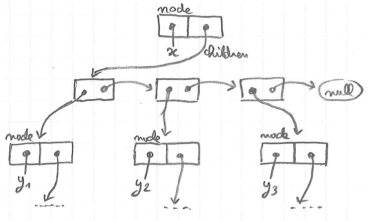
## Trees with list of subtrees: representation of nodes

$p \rightsquigarrow \mathsf{Nodeof}\ R_1\ V_1\ R_2\ V_2 \quad \equiv$
  $\exists v_1 v_2.\quad p \mapsto \{|\mathsf{item}{=}v_1;\ \mathsf{children}{=}v_2|\}$
  $\star\ v_1 \rightsquigarrow R_1\ V_1$
  $\star\ v_2 \rightsquigarrow R_2\ V_2$

$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$
  $\mathsf{match}\ T\ \mathsf{with}$
  $|\ \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]$
  $|\ \mathsf{Node}\ X\ L \Rightarrow \exists xc.\quad p \mapsto \{|\mathsf{item}{=}x;\ \mathsf{children}{=}c|\}$
  $\star\ x \rightsquigarrow R\ X$
  $\star\ c \rightsquigarrow \mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R)\ L$



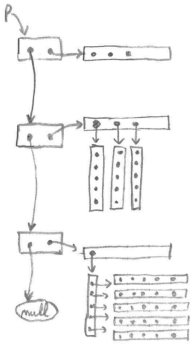
## Trees with list of subtrees, revisited




**Exercise:** rewrite the specification of Narytreeof using Nodeof.

$p \rightsquigarrow \mathsf{Narytreeof}\ R\ T \equiv$
  $\mathsf{match}\ T\ \mathsf{with}$
  $|\ \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]$
  $|\ \mathsf{Node}\ X\ L \Rightarrow p \rightsquigarrow \mathsf{Nodeof}\ R\ X\ (\mathsf{Mlistof}\ (\mathsf{Narytreeof}\ R))\ L$

## More involved application




- Exam from 2015, Exercise 3: bootstrapped chunked bags.

Available from the webpage of the course.



# Chapter 23

Iteration with higher-order representation predicates



## Iteration on lists

Recall:

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

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

Challenge:

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

**Question:** Can we use an invariant $I \equiv \lambda K.(\ldots)$?
(i.e. with a spec of the form $\{p \rightsquigarrow \ldots \star I\ \mathsf{nil}\}(\ldots)\{p \rightsquigarrow \ldots \star I\ L\}$ ?)



## 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.

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

## 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`.

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

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



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

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

Consider:

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

Derives:

$$\big(\forall x X.\ \{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{fun x -> incr x})x\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}\big) \Rightarrow$$
$$\{p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\, L\}\ (\texttt{incr\_all}\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\,(\mathsf{map}\,(+1)\,L)\}$$



# Chapter 24

Resource analysis in Separation Logic



## Controlling deallocation

(1) Remove the garbage collection rule:

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \text{GC-POST}$$

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

$$\{r \mapsto v\}\ (\texttt{free r})\ \{\lambda\_.\ [\,]\}$$

(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.

$$\{[\,]\}\ t\ \{\lambda n.\ [P\,n]\}$$

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

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



## Complexity analysis

Time credits:

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

Properties:

$$\$(x + y) \ =\ \$\, x \ \star\ \$\, y \quad \text{and} \quad \$\, 0 \ =\ [\,]$$

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:

$$\{t \rightsquigarrow \mathsf{Array}\, M \ \star\ \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\, [n = |M|] \star t \rightsquigarrow \mathsf{Array}\, M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\}\ (\texttt{Array.make n v})\ \{\lambda t.\ \exists L.\ t \rightsquigarrow \mathsf{Array}\, L \star [...]\}$$

Quasilinear time:

$$\{t \rightsquigarrow \mathsf{Array}\, L \star \$(c_1 |L| \log |L| + c_2)\}$$
$$(\texttt{Array.sort t})$$
$$\{\lambda t.\ \exists L'.\ t \rightsquigarrow \mathsf{Array}\, L' \star [...]\}$$



## Amortized analysis

Stack of unbounded size with amortized constant-time operations:

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

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

$\{s \rightsquigarrow \mathsf{Stack}\,(x :: L) \star \$\, c\}$ $(\texttt{Stack.pop s})$ $\{\lambda y.\ [y = x] \star s \rightsquigarrow \mathsf{Stack}\, L\}$

Representation predicate with a potential function:

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

# Chapter 25

Read-only permissions



## Motivation for read-only permissions

What we currently need to write:

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

What we wish to write:

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

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



## Fractional permissions

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

Splitting and merging:

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

More generally:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \star (r \overset{\beta}{\mapsto} v) \qquad \text{with } \ 0 < \alpha, \beta \leqslant 1$$

Operations:

$$\{[\,]\}\ (\texttt{ref v})\ \{\lambda r.\ r \overset{1}{\mapsto} v\}$$
$$\{r \overset{1}{\mapsto} v'\}\ (\texttt{r := v})\ \{\lambda \_.\ r \overset{1}{\mapsto} v\}$$
$$\forall \alpha. \qquad \{r \overset{\alpha}{\mapsto} v\} \quad (\texttt{!r}) \quad \{\lambda x.\ [x = v] \star (r \overset{\alpha}{\mapsto} v)\}$$



## Fractional permissions in practice

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

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 $H$.

## 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 duplicatable and never mentioned in post-conditions.

$$\frac{}{\mathsf{RO}(H) \;\vdash\; \mathsf{RO}(H) \star \mathsf{RO}(H)}\ \textsc{dup-ro}$$

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



## Read-only frame rule

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

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



## Read-only sequencing rule

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

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

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



## RO in practice

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

# Chapter 26

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
```

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

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

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

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

## Parallel rule needs read-only permissions

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

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

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

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

$$\frac{\{H_1 \star \mathsf{RO}(H_3)\}\ t_1\ \{Q_1\} \qquad \{H_2 \star \mathsf{RO}(H_3)\}\ t_2\ \{Q_2\}}{\{H_1 \star H_2 \star \mathsf{RO}(H_3)\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \text{PARALLEL-RO}$$



## Concurrent locks: example

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

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

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

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



## Concurrent locks: specification of operations

Duplicatable representation predicate:

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

Operations:

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

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

$\forall pH.$ $\{H \star p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}\ (\texttt{release\_lock p})\ \{\lambda\_.\ [\,]\}$



## Concurrent locks: exercise

**Exercise:** Describe the state at the front of each lines (except 5 and 6). 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     let () = acquire_lock p in
7     incr r;
8     decr s;
9     release_lock p
```

1: $[]$. 2: $r \mapsto 0$. 3: $r \mapsto 0 \star s \mapsto n$.
4: $p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\,(\exists i.\ (r \mapsto i) \star (s \mapsto n - i))$.
7: $(r \mapsto i) \star (s \mapsto n - i)$. 8: $(r \mapsto i + 1) \star (s \mapsto n - i)$.
9: $(r \mapsto i + 1) \star (s \mapsto n - i - 1)$. 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 27

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;

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

$$p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\,(\exists n.\ r \mapsto n \star [\ \ldots\ ?\ \ldots\ ])$$

**Problem:** it is impossible to prove, only with invariants, that this program does not crash (i.e. to prove {True} program {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;

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

**Exercise:** Give a lock invariant that allows proving {True} program {True}

$$p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\,(\exists n, n_1, n_2.\ (r \mapsto n) \star (r_1 \overset{1/2}{\mapsto} n_1) \star (r_2 \overset{1/2}{\mapsto} n_2) \star [n = n_1 + n_2])$$

## Proof

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

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

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



## Left thread

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

$\{(r_1 \overset{1/2}{\mapsto} 0) * p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}$
```
acquire_lock p;
```
$\{(r_1 \overset{1/2}{\mapsto} 0) * H)\}$ so, for some $n, n_1, n_2$ such that $n = n_1 + n_2$:
$\{(r_1 \overset{1/2}{\mapsto} 0) * (r \mapsto n) * (r_1 \overset{1/2}{\mapsto} n_1) * (r_2 \overset{1/2}{\mapsto} n_2)\}$
```
r := !r + 1;
```
$\{(r_1 \overset{1/2}{\mapsto} 0) * (r \mapsto n + 1) * (r_1 \overset{1/2}{\mapsto} n_1) * (r_2 \overset{1/2}{\mapsto} n_2)\}$
$\{(r_1 \overset{1}{\mapsto} 0) * (r \mapsto n + 1) * (r_2 \overset{1/2}{\mapsto} n_2)\}$
```
r1 := !r1 + 1;
```
$\{(r_1 \overset{1}{\mapsto} 1) * (r \mapsto n + 1) * (r_2 \overset{1/2}{\mapsto} n_2)\}$
$\{(r_1 \overset{1/2}{\mapsto} 1) * (r \mapsto n + 1) * (r_1 \overset{1/2}{\mapsto} 1) * (r_2 \overset{1/2}{\mapsto} n_2)\}$
$\{(r_1 \overset{1/2}{\mapsto} 1) * H\}$
```
release_lock p;
```
$\{(r_1 \overset{1/2}{\mapsto} 1)\}$ no $p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H$?



## Right thread

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

$\{(r_2 \overset{1/2}{\mapsto} 0) * p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}$
```
acquire_lock p;
```
$\{(r_2 \overset{1/2}{\mapsto} 0) * H)\}$
```
r := !r + 1;
r2 := !r2 + 1;
```
$\{(r_2 \overset{1/2}{\mapsto} 1) * H\}$
```
release_lock p;
```
$\{(r_2 \overset{1/2}{\mapsto} 1)\}$



## Finish up

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

| | |
|---|---|
| $\{(r_1 \overset{1/2}{\mapsto} 0) * p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}$ | $\{(r_2 \overset{1/2}{\mapsto} 0) * p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}$ |
| `acquire_lock p;` | `acquire_lock p;` |
| `r := !r + 1;` | `r := !r + 1;` |
| `r1 := !r1 + 1;` | `r2 := !r2 + 1;` |
| `release_lock p;` | `release_lock p;` |
| $\{(r_1 \overset{1/2}{\mapsto} 1) * p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}$ | $\{(r_2 \overset{1/2}{\mapsto} 1) * p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\, H\}$ |

$\{(r_1 \overset{1/2}{\mapsto} 1) * (r_2 \overset{1/2}{\mapsto} 1) * (r \mapsto n) * (r_1 \overset{1/2}{\mapsto} n_1) * (r_2 \overset{1/2}{\mapsto} n_2) * [n = n_1 + n_2]\}$
$\{(r_1 \mapsto 1) * (r_2 \mapsto 1) * (r \mapsto n) * [n = 1 + 1]\}$
```
assert (!r == 2);
```

$p \overset{\mathsf{ro}}{\rightsquigarrow} \mathsf{Lock}\,(\exists n, n_1, n_2.\ (r \mapsto n) \star (r_1 \overset{1/2}{\mapsto} n_1) \star (r_2 \overset{1/2}{\mapsto} n_2) \star [n = n_1 + n_2])$

**Question:** Would replacing "$1/2$" with "ro" work?



## 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. A summary of *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:

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

- validity in RA's e.g. $\mathsf{valid}((r_2 \mapsto 1) \cdot (r_2 \mapsto n_2)) \ \Rightarrow\ n_2 = 1$



## Conclusion

Program verification using Separation Logic gives you:

- Expressiveness: tree-shaped structures, and structures with sharing
- Expressiveness: effectful, first-class functions, with local state
- Expressiveness: concurrency, ghost state
- Modularity: most-general specifications
- Modularity: composable representation predicates
- Abstraction: existential quantification of intermediate pointers
- Abstraction: existential quantification of invariants
- Practice: formalization in Coq of all heap predicates
- Practice: characteristic formulas for reasoning rules
- Practice: more advanced separation logic
  `https://iris-project.org/#learning`