Temporary Read-Only Permissions for Separation Logic

Arthur Charguéraud   François Pottier

(talk delivered by **Armaël Guéneau**)

ESOP 2017
Uppsala, April 28, 2017

Separation Logic: to own, or not to own

The setting is basic (sequential) Separation Logic.

Separation Logic is about **disjointness**, therefore about **unique ownership**.

A dichotomy arises:

> *To every memory cell, array, or user-defined data structure,*
> *either we have **no access at all**, or we have **full read-write access**.*

Separation Logic: to own, or not to own

The setting is basic (sequential) Separation Logic.

Separation Logic is about **disjointness**, therefore about **unique ownership**.

A dichotomy arises:

> *To every memory cell, array, or user-defined data structure,*
> *either we have **no access at all**, or we have **full read-write access**.*

This is visible in the read and write axioms, which both need a **full permission**:

SET
$\{l \hookrightarrow v'\}\ (\mathsf{set}\ l\ v)\ \{\lambda y.\ l \hookrightarrow v\}$

TRADITIONAL READ AXIOM
$\{l \hookrightarrow v\}\ (\mathsf{get}\ l)\ \{\lambda y.\ [y = v] \star l \hookrightarrow v\}$

The problem

Suppose we are implementing an abstract data type of mutable sequences.

An abstract predicate $s \leadsto \mathsf{Seq}\, L$ represents the **unique ownership** of a sequence.

Here is a typical specification of sequence concatenation:

$$\begin{array}{l} \{s_1 \leadsto \mathsf{Seq}\, L_1 \star s_2 \leadsto \mathsf{Seq}\, L_2\} \\ (\mathit{append}\, s_1\, s_2) \\ \{\lambda s_3.\ s_3 \leadsto \mathsf{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \leadsto \mathsf{Seq}\, L_1 \star s_2 \leadsto \mathsf{Seq}\, L_2\} \end{array}$$

The problem

Suppose we are implementing an abstract data type of mutable sequences.

An abstract predicate $s \rightsquigarrow \mathrm{Seq}\, L$ represents the **unique ownership** of a sequence.

Here is a typical specification of sequence concatenation:

$$\begin{array}{l}\{s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\} \\ (\mathit{append}\, s_1\, s_2) \\ \{\lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\}\end{array}$$

Although correct, this style of specification can be criticized on several grounds:

- It is a bit **noisy**.
- It requires the permissions $s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2$ to be **threaded through the proof** of *append*.
- It actually **does not guarantee that s_1 and s_2 are unmodified** in memory.
- It requires s_1 and s_2 to be **distinct** data structures. — (next slide)

The problem, focus: sharing is not permitted

This specification requires s_1 and s_2 to be **distinct** (disjoint) data structures:

$$
\begin{array}{l}
\{s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \;\star\; s_2 \rightsquigarrow \mathrm{Seq}\, L_2\} \\
(\mathit{append}\, s_1\, s_2) \\
\{\lambda s_3.\; s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2) \;\star\; s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \;\star\; s_2 \rightsquigarrow \mathrm{Seq}\, L_2\}
\end{array}
$$

$(\mathit{append}\, s\, s)$ requires $s \rightsquigarrow \mathrm{Seq}\, L \;\star\; s \rightsquigarrow \mathrm{Seq}\, L$, which the client cannot produce.

The problem, focus: sharing is not permitted

This specification requires s_1 and s_2 to be **distinct** (disjoint) data structures:

$$\begin{array}{l}\{s_1 \leadsto \mathrm{Seq}\, L_1 \;\star\; s_2 \leadsto \mathrm{Seq}\, L_2\}\\ (\mathit{append}\, s_1\; s_2)\\ \{\lambda s_3.\; s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2) \star s_1 \leadsto \mathrm{Seq}\, L_1 \;\star\; s_2 \leadsto \mathrm{Seq}\, L_2\}\end{array}$$

$(\mathit{append}\, s\; s)$ requires $s \leadsto \mathrm{Seq}\, L \;\star\; s \leadsto \mathrm{Seq}\, L$, which the client cannot produce.

To allow $(\mathit{append}\, s\; s)$, we must establish **another specification**:

$$\begin{array}{l}\{s \leadsto \mathrm{Seq}\, L\}\\ (\mathit{append}\, s\; s)\\ \{\lambda s_3.\; s_3 \leadsto \mathrm{Seq}\,(L \mathbin{+\!\!+} L) \star s \leadsto \mathrm{Seq}\, L\}\end{array}$$

Duplicate work and **increased complication** for us and for our clients.

Fractional permissions to the rescue...?

In (some) Concurrent SLs, sequence concatenation can be specified as follows:

$$\begin{array}{l}\forall \pi_1, \pi_2.\ \{\pi_1 \cdot (s_1 \leadsto \mathsf{Seq}\, L_1) \star \pi_2 \cdot (s_2 \leadsto \mathsf{Seq}\, L_2)\} \\ \qquad (\mathit{append}\, s_1\, s_2) \\ \qquad \{\lambda s_3.\ s_3 \leadsto \mathsf{Seq}\,(L_1 \mathbin{+\!\!+} L_2) \star \pi_1 \cdot (s_1 \leadsto \mathsf{Seq}\, L_1) \star \pi_2 \cdot (s_2 \leadsto \mathsf{Seq}\, L_2)\}\end{array}$$

We **scale** an assertion by a fraction: $\pi \cdot H$.

Fractional permissions to the rescue...?

In (some) Concurrent SLs, sequence concatenation can be specified as follows:

$$\begin{array}{l} \forall \pi_1, \pi_2.\ \{\pi_1 \cdot (s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \pi_2 \cdot (s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\} \\ \qquad (\mathit{append}\, s_1\, s_2) \\ \qquad \{\lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star \pi_1 \cdot (s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \pi_2 \cdot (s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\} \end{array}$$

We **scale** an assertion by a fraction: $\pi \cdot H$.

This addresses the main aspects of the problem,

- but is still **noisy**,
- might seem a bit frightening to non-experts,
- and still requires careful **splitting, threading, and joining** of permissions.
- “Hiding” fractions (Heule et al, 2013) adds another layer of sophistication.

In this paper

We propose a solution that is:

- **not as powerful** as fractional permissions (or other share algebras),
- but significantly **simpler**. (A design “sweet spot”?)

Our contributions:

- introducing a **read-only modality**, *const* (in the paper: “RO”).
  $\mathit{const}(H)$ gives **temporary** read-only access to the memory governed by H.
- finding **simple and sound reasoning rules** for *const*.
- proposing **a model** that justifies these rules.

Some Intuition

Reasoning Rules

Model

Conclusion

Our solution

We would like the specification of *append* to look like this:

$$\{const(s_1 \rightsquigarrow \mathsf{Seq}\, L_1) \star const(s_2 \rightsquigarrow \mathsf{Seq}\, L_2)\}$$
$$(append\, s_1\, s_2)$$
$$\{\lambda s_3.\ s_3 \rightsquigarrow \mathsf{Seq}\, (L_1 \mathbin{+\!\!+} L_2)\}$$

Our solution

We would like the specification of *append* to look like this:

$$
\begin{array}{l}
\{\mathit{const}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \mathit{const}(s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\} \\
(\mathit{append}\, s_1\; s_2) \\
\{\lambda s_3.\; s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)\}
\end{array}
$$

Compared with the earlier specification based on unique read-write permissions,

- this specification is **more concise**,
- imposes **fewer proof obligations**,
- makes it clear that **the data structures cannot be modified** by *append*,
- and **does not require s_1 and s_2 to be distinct**. — (next slide)
- Furthermore, this spec **implies** both earlier specs. — (next slide)

The new spec subsumes both earlier specs

$$\{\mathbf{const}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \mathbf{const}(s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\}$$
$$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$$
$$\{\lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\}$$

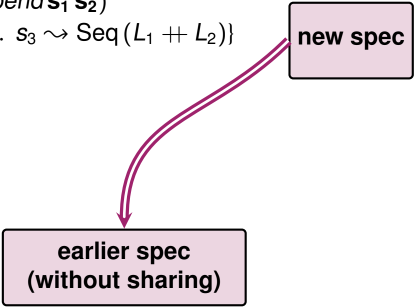
new spec

The new spec subsumes both earlier specs

$\{\mathbf{const}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \mathbf{const}(s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\}$

$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$

$\{\lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)\}$



$\{s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\}$

$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$

$\{\lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)$

$\star\, s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\}$

The new spec subsumes both earlier specs

$\{\mathbf{const}(s_1 \leadsto \mathrm{Seq}\, L_1) \star \mathbf{const}(s_2 \leadsto \mathrm{Seq}\, L_2)\}$
$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\, (L_1 +\!\!+\, L_2)\}$

new spec

new spec (with sharing)

$\{\mathbf{const}(s \leadsto \mathrm{Seq}\, L)\}$
$(\mathit{append}\, \mathbf{s}\, \mathbf{s})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\, (L +\!\!+\, L)\}$

earlier spec (without sharing)

$\{s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\}$
$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\, (L_1 +\!\!+\, L_2)$
$\star\, s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\}$

The new spec subsumes both earlier specs

$\{\mathbf{const}(s_1 \leadsto \mathrm{Seq}\, L_1) \star \mathbf{const}(s_2 \leadsto \mathrm{Seq}\, L_2)\}$
$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\}$

new spec

new spec (with sharing)

$\{\mathbf{const}(s \leadsto \mathrm{Seq}\, L)\}$
$(\mathit{append}\, \mathbf{s}\, \mathbf{s})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L +\!\!+ L)\}$

earlier spec (without sharing)

$\{s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\}$
$(\mathit{append}\, \mathbf{s_1}\, \mathbf{s_2})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)$
$\star\, s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\}$

earlier spec (with sharing)

$\{s \leadsto \mathrm{Seq}\, L\}$
$(\mathit{append}\, \mathbf{s}\, \mathbf{s})$
$\{\lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L +\!\!+ L)$
$\star\, s \leadsto \mathrm{Seq}\, L\}$

Some Intuition

Reasoning Rules

Model

Conclusion

Assertions

The syntax of assertions is extended:

$$H \quad := \quad [P] \mid l \hookrightarrow v \mid H_1 \star H_2 \mid H_1 \mathbin{\forall\!\!\!\forall} H_2 \mid \exists\!\!\exists x.\, H \mid \mathit{const}(H)$$

Read-only access to a data structure entails **read-only access to its parts**:

$$\mathit{const}(H_1 \star H_2) \quad \Vdash \quad \mathit{const}(H_1) \star \mathit{const}(H_2) \qquad \text{(the reverse is false)}$$

Read-only access can be **shared**:

$$\mathit{const}(H) \quad = \quad \mathit{const}(H) \star \mathit{const}(H)$$

(A few more axioms, not shown).

A new read axiom

The traditional read axiom:

TRADITIONAL READ AXIOM

$$\{l \hookrightarrow v\}\ (\text{get}\ l)\ \{\lambda y.\ [y = v] \star l \hookrightarrow v\}$$

is replaced with a “smaller” axiom:

NEW READ AXIOM

$$\{const(l \hookrightarrow v)\}\ (\text{get}\ l)\ \{\lambda y.\ [y = v]\}$$

The traditional axiom can be derived from the new axiom.

A new frame rule

The traditional frame rule is subsumed by a new "read-only frame rule":

FRAME RULE

$$\frac{\{H\}\ t\ \{Q\} \qquad normal(H')}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

READ-ONLY FRAME RULE

$$\frac{\{H \star const(H')\}\ t\ \{Q\} \qquad normal(H')}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

Upon entry into a code block, H' can be **temporarily replaced** with $const(H')$, and upon exit, H' **magically re-appears**.

A new frame rule

The traditional frame rule is subsumed by a new “read-only frame rule”:

$$\textsc{Frame Rule}\quad \frac{\{H\}\ t\ \{Q\} \qquad \mathit{normal}(H')}{\{H \star H'\}\ t\ \{Q \star H'\}} \qquad\qquad \textsc{Read-only Frame Rule}\quad \frac{\{H \star \mathit{const}(H')\}\ t\ \{Q\} \qquad \mathit{normal}(H')}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

Upon entry into a code block, H' can be **temporarily replaced** with $\mathit{const}(H')$, and upon exit, H' **magically re-appears**.

The side condition $\mathit{normal}(H')$ means roughly that H' has no *const* components.

This means that **read-only permissions cannot be framed out**.

Not a problem, as they are always **passed down**, never returned.

They never appear in postconditions.

Some Intuition

Reasoning Rules

Model

Conclusion

Interpretation of assertions

In a heap fragment, let every cell be colored **RW** or **RO**.

Let separating conjunction require:

- **disjointness** of the RW areas;
- **disjointness** of one side's RW area with the other side's RO area;
- **agreement** on the content of the heap where the RO areas overlap.

If an assertion H describes a certain set of heaps, then:

- Let $\mathit{const}(H)$ describe **the same heaps, colored entirely RO**.
- Let $\mathit{normal}(H)$ mean that every heap in H is colored entirely RW.

Interpretation of triples

The meaning of the Hoare triple $\{H\}\ t\ \{Q\}$ is a variant of the usual:

$$\forall h_1 h_2. \left\{ \begin{array}{l} h_1 + h_2 \text{ is defined} \\ H\, h_1 \end{array} \right\} \Rightarrow \exists v h_1'. \left\{ \begin{array}{l} h_1' + h_2 \text{ is defined} \\ t / \lfloor h_1 + h_2 \rfloor \Downarrow v / \lfloor h_1' + h_2 \rfloor \\ (Q \star \mathit{true})\, v\, h_1' \\ \end{array} \right\}$$

Roughly, “If part of the heap satisfies H,
then t runs safely and changes that part of the
heap to satisfy Q, leaving the rest untouched.”

We make two changes:

Interpretation of triples

The meaning of the Hoare triple $\{H\}\ t\ \{Q\}$ is a variant of the usual:

$$\forall h_1 h_2. \left\{ \begin{array}{l} h_1 + h_2 \text{ is defined} \\ H\, h_1 \end{array} \right\} \Rightarrow \exists v h_1'. \left\{ \begin{array}{l} h_1' + h_2 \text{ is defined} \\ t / \lfloor h_1 + h_2 \rfloor \Downarrow v / \lfloor h_1' + h_2 \rfloor \\ \text{on-some-rw-frag}(Q\, v)\, h_1' \\ \end{array} \right\}$$

Roughly, "If part of the heap satisfies H, then t runs safely and changes that part of the heap to satisfy Q, leaving the rest untouched."

We make two changes:

- Q describes **a purely RW part** of the final heap.

Interpretation of triples

The meaning of the Hoare triple $\{H\}\ t\ \{Q\}$ is a variant of the usual:

$$\forall h_1 h_2 . \left\{ \begin{array}{l} h_1 + h_2 \text{ is defined} \\ H\, h_1 \end{array} \right\} \Rightarrow \exists v h'_1 . \left\{ \begin{array}{l} h'_1 + h_2 \text{ is defined} \\ t / \lfloor h_1 + h_2 \rfloor \Downarrow v / \lfloor h'_1 + h_2 \rfloor \\ \text{on-some-rw-frag}(Q\, v)\, h'_1 \\ h'_1.\mathrm{r} = h_1.\mathrm{r} \end{array} \right\}$$

Roughly, “If part of the heap satisfies H, then t runs safely and changes that part of the heap to satisfy Q, leaving the rest untouched.”

We make two changes:

- Q describes **a purely RW part** of the final heap.
- The RO part of the heap is **preserved**, even though Q says nothing about it.

Soundness

Theorem

With respect to this interpretation of triples, every reasoning rule is sound.

Proof.

“Straightforward”. Machine-checked. □

What about concurrency?

Our proof is carried out in a **sequential** setting.

- The proof uses big-step operational semantics.

What about **structured concurrency**, i.e., parallel composition ($e_1 \parallel e_2$)?

- We believe that *const* permissions remain sound,
- but do not have a proof – a different proof technique is required.
- They allow read-only state to be shared between threads.

What about **unstructured concurrency**, i.e., threads and channels?

- One cannot allow *const* permissions to be sent along channels.
- More complex machinery is required: fractions, lifetimes, . . .

Conclusion

We propose:

- **a simple extension** of Separation Logic with a read-only modality;
- **a simple model** that explains why this is sound.

A possible design sweet spot?

- not so easy to find, worth knowing about;
- applicable to PL design? (e.g., adding *const* to Mezzo)

Applications:

- more **concise**, more **accurate**, more **general** specifications;
- **simpler** proofs.

Pending implementation in CFML (Charguéraud).

Some Intuition

Reasoning Rules

Model

Conclusion

Temporary modifications are not forbidden

Repeating "$s \leadsto \mathrm{Seq}\, L$" in the pre- and postcondition can be deceiving.

This does **not** forbid changes to the **concrete** data structure in memory.

Here is a function that really just reads the data structure:

$$\{s \leadsto \mathrm{Seq}\, L\}\ (\mathit{length}\ s)\ \{\lambda y.\ s \leadsto \mathrm{Seq}\, L\ \star\ [\, y = |L|\,]\}$$

Temporary modifications are not forbidden

Repeating “$s \rightsquigarrow \mathrm{Seq}\, L$” in the pre- and postcondition can be deceiving.

This does **not** forbid changes to the **concrete** data structure in memory.

Here is a function that really just reads the data structure:

$$\{s \rightsquigarrow \mathrm{Seq}\, L\}\ (\mathit{length}\ s)\ \{\lambda y.\ s \rightsquigarrow \mathrm{Seq}\, L\ \star\ [\,y = |L|\,]\}$$

And a function that actually modifies the data structure:

$$\{s \rightsquigarrow \mathrm{Seq}\, L\ \star\ [\,|L| \le n\,]\}\ (\mathit{resize}\ s\ n)\ \{\lambda ().\ s \rightsquigarrow \mathrm{Seq}\, L\}$$

Amnesia (1/2)

Suppose *population* has this specification:

$$\{const(h \rightsquigarrow \mathrm{HashTable}\, M)\}\ (population\ h)\ \{\lambda y.\ [y = \mathrm{card}\, M]\}$$

Suppose a hash table is a mutable record whose *data* field points to an array:

$$\begin{array}{l} h \rightsquigarrow \mathrm{HashTable}\, M := \\ \quad \exists a. \exists L. (h \rightsquigarrow \{data = a; \ldots\} \star a \rightsquigarrow \mathrm{Array}\, L \star \ldots) \end{array}$$

Suppose there is an operation *foo* on hash tables:

$$\begin{array}{ll} \mathrm{let}\ foo\ h = & \\ \quad \mathrm{let}\ d = h.data\ \mathrm{in} & \text{– read the address of the array} \\ \quad \mathrm{let}\ p = population\ h\ \mathrm{in} & \text{– call } population \\ \quad \ldots & \end{array}$$

If “const” is sugar for repeating $h \rightsquigarrow \mathrm{HashTable}\, M$ in the pre and post, then the proof of *foo* runs into a problem...

Amnesia (2/2)

Reasoning about *foo* might go like this:

1 let *foo* $h =$
2 $\{h \rightsquigarrow \text{HashTable}\ M\}$ – *foo*'s precondition
3 $\{h \rightsquigarrow \{data = a; \ldots\} \star a \rightsquigarrow \text{Array}\ L \star \ldots\}$ – by unfolding
4 let $d = h.data$ in
5 $\{h \rightsquigarrow \{data = a; \ldots\} \star a \rightsquigarrow \text{Array}\ L \star \ldots \star [d = a]\}$ – by reading
6 $\{h \rightsquigarrow \text{HashTable}\ M \star [d = a]\}$ – by folding
7 let $p = population\ h$ in – we have to fold
8 $\{h \rightsquigarrow \text{HashTable}\ M \star [d = a] \star [p = \#M]\}$
9 $\ldots$

At line 8, the equation $d = a$ is useless.

We have **forgotten** what d represents, and **lost the benefit** of the read at line 4.

If "const" is sugar, the specification of *population* is **weaker** than it seems.

If "const" is native, there is a way around this problem. (Details omitted.)

Our solution, facet 4: sharing is permitted

The top Hoare triple is the new spec of *append*, where s_1 and s_2 are instantiated with s.

$$\frac{\begin{array}{l}\{\ \ \mathit{const}(s \leadsto \mathrm{Seq}\,L) \star \mathit{const}(s \leadsto \mathrm{Seq}\,L)\ \ \} \\ (\mathit{append}\,s\,s) \\ \{\ \ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L +\!\!+ L)\ \ \}\end{array}}{\begin{array}{l}\{\ \ \mathit{const}(s \leadsto \mathrm{Seq}\,L)\ \ \} \\ (\mathit{append}\,s\,s) \\ \{\ \ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L +\!\!+ L)\ \ \}\end{array}}\ \textsf{CONSEQUENCE}$$

The bottom triple states that, with read-only access to s, *append* s s is **permitted**.

Our solution, facet 4: sharing is permitted

The top Hoare triple is the new spec of *append*,
where s_1 and s_2 are instantiated with s.

$$\dfrac{\begin{array}{l} \{\ \mathit{const}(s \rightsquigarrow \mathrm{Seq}\,L) \star \mathit{const}(s \rightsquigarrow \mathrm{Seq}\,L)\ \} \\ (\mathit{append}\, s\, s) \\ \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L +\!\!+ L)\ \} \end{array}}{\begin{array}{l} \{\ \mathit{const}(s \rightsquigarrow \mathrm{Seq}\,L)\ \} \\ (\mathit{append}\, s\, s) \\ \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L +\!\!+ L)\ \} \end{array}}\ \textsc{Consequence}$$

read-only permissions are duplicable

The bottom triple states that, with read-only access to s, *append* s s is **permitted**.

Our solution, facet 5: the earlier specification can be derived

The Hoare triple at the top is the new spec of *append*.

$$
\dfrac{\dfrac{\begin{array}{l}\{\ \mathit{const}(s_1 \leadsto \mathrm{Seq}\, L_1) \star \mathit{const}(s_2 \leadsto \mathrm{Seq}\, L_2)\ \}\\ (\mathit{append}\, s_1\, s_2)\\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\ \}\end{array}}{\begin{array}{l}\{\ \mathit{const}(s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2)\ \}\\ (\mathit{append}\, s_1\, s_2)\\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\ \}\end{array}}\ \textsc{consequence}}{\begin{array}{l}\{\ s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\ \}\\ (\mathit{append}\, s_1\, s_2)\\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2) \star s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\ \}\end{array}}\ \textsc{read-only frame}
$$

The triple at the bottom is the earlier spec of *append*.

Our solution, facet 5: the earlier specification can be derived

The Hoare triple at the top is the new spec of *append*.

permission becomes read-only

$$
\dfrac{
  \dfrac{
    \begin{array}{l}
    \{\ const(s_1 \rightsquigarrow \mathrm{Seq}\,L_1) \star const(s_2 \rightsquigarrow \mathrm{Seq}\,L_2)\ \} \\
    (append\, s_1\ s_2) \\
    \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}
    \end{array}
  }{
    \begin{array}{l}
    \{\ const(s_1 \rightsquigarrow \mathrm{Seq}\,L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\,L_2)\ \} \\
    (append\, s_1\ s_2) \\
    \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}
    \end{array}
  }\ \textsc{Consequence}
}{
  \begin{array}{l}
  \{\ s_1 \rightsquigarrow \mathrm{Seq}\,L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\,L_2\ \} \\
  (append\, s_1\ s_2) \\
  \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \rightsquigarrow \mathrm{Seq}\,L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\,L_2\ \}
  \end{array}
}\ \textsc{Read-only frame}
$$

The triple at the bottom is the earlier spec of *append*.

Our solution, facet 5: the earlier specification can be derived

The Hoare triple at the top is the new spec of *append*.

read-only permission is split

$$
\dfrac{\dfrac{\begin{array}{l}\{\ \mathit{const}(s_1 \leadsto \mathrm{Seq}\,L_1) \star \mathit{const}(s_2 \leadsto \mathrm{Seq}\,L_2)\ \} \\ (\mathit{append}\; s_1\; s_2) \\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}\end{array}}{\begin{array}{l}\{\ \mathit{const}(s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2)\ \} \\ (\mathit{append}\; s_1\; s_2) \\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}\end{array}}\ \textsc{Consequence}}{\begin{array}{l}\{\ s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2\ \} \\ (\mathit{append}\; s_1\; s_2) \\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2\ \}\end{array}}\ \textsc{Read-only frame}
$$

The triple at the bottom is the earlier spec of *append*.

Our solution, facet 5: the earlier specification can be derived

The Hoare triple at the top is the new spec of *append*.

$$
\cfrac{
  \cfrac{
    \begin{array}{l}
    \{\ \mathit{const}(s_1 \rightsquigarrow \mathsf{Seq}\, L_1) \star \mathit{const}(s_2 \rightsquigarrow \mathsf{Seq}\, L_2)\ \} \\
    (\mathit{append}\, s_1\, s_2) \\
    \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathsf{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\ \}
    \end{array}
  }{
    \begin{array}{l}
    \{\ \mathit{const}(s_1 \rightsquigarrow \mathsf{Seq}\, L_1 \star s_2 \rightsquigarrow \mathsf{Seq} \\
    (\mathit{append}\, s_1\, s_2) \\
    \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathsf{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\ \}
    \end{array}
  }\ \text{CONSEQUENCE}
}{
  \begin{array}{l}
  \{\ s_1 \rightsquigarrow \mathsf{Seq}\, L_1 \star s_2 \rightsquigarrow \mathsf{Seq}\, L_2\ \} \\
  (\mathit{append}\, s_1\, s_2) \\
  \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathsf{Seq}\,(L_1 \mathbin{+\!\!+} L_2) \star s_1 \rightsquigarrow \mathsf{Seq}\, L_1 \star s_2 \rightsquigarrow \mathsf{Seq}\, L_2\ \}
  \end{array}
}\ \text{READ-ONLY FRAME}
$$

***append* is called**

The triple at the bottom is the earlier spec of *append*.

Our solution, facet 5: the earlier specification can be derived

The Hoare triple at the top is the new spec of *append*.

$$\{\ const(s_1 \leadsto \mathrm{Seq}\,L_1) \star const(s_2 \leadsto \mathrm{Seq}\,L_2)\ \}$$
$$(append\, s_1\, s_2)$$
$$\{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}$$

CONSEQUENCE

$$\{\ const(s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\ \dots$$
$$(append\, s_1\, s_2)$$
$$\{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}$$

read-write permission magically re-appears

READ-ONLY FRAME

$$\{\ s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2\ \}$$
$$(append\, s_1\, s_2)$$
$$\{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2\ \}$$

The triple at the bottom is the earlier spec of *append*.

Our solution, facet 5: the earlier specification can be derived

The Hoare triple at the top is the new spec of *append*.

$$
\dfrac{
\dfrac{
\begin{array}{l}
\{\ \mathit{const}(s_1 \leadsto \mathrm{Seq}\,L_1) \star \mathit{const}(s_2 \leadsto \mathrm{Seq}\,L_2)\ \} \\
(\mathit{append}\, s_1\ s_2) \\
\{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}
\end{array}
}{
\begin{array}{l}
\{\ \mathit{const}(s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq} \\
(\mathit{append}\, s_1\ s_2) \\
\{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}
\end{array}
}\ \textsc{Consequence}
}{
\begin{array}{l}
\{\ s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2\ \} \\
(\mathit{append}\, s_1\ s_2) \\
\{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \leadsto \mathrm{Seq}\,L_1 \star s_2 \leadsto \mathrm{Seq}\,L_2\ \}
\end{array}
}\ \textsc{Read-only frame}
$$

no need to join read-only permissions

The triple at the bottom is the earlier spec of *append*.

Assertions

The syntax of assertions is extended:

$$H \quad := \quad [P] \mid l \hookrightarrow v \mid H_1 \star H_2 \mid H_1 \mathbin{\text{\textbackslash\!\!/}} H_2 \mid \exists x.\, H \mid \mathit{const}(H)$$

Read-only access to a data structure entails read-only access to its parts:

$$\mathit{const}(H_1 \star H_2) \quad \Vdash \quad \mathit{const}(H_1) \star \mathit{const}(H_2) \qquad \text{(the reverse is false)}$$

Read-only permissions are duplicable (therefore, no need to count them!):

$$\mathit{const}(H) \quad = \quad \mathit{const}(H) \star \mathit{const}(H)$$

Read-only permissions are generally well-behaved:

$$\begin{array}{rcll}
\mathit{const}([P]) & = & [P] & \\
\mathit{const}(H_1 \vee H_2) & = & \mathit{const}(H_1) \vee \mathit{const}(H_2) & \\
\mathit{const}(\exists x.\, H) & = & \exists x.\, \mathit{const}(H) & \\
\mathit{const}(\mathit{const}(H)) & = & \mathit{const}(H) & \\
\mathit{const}(H) & \Vdash & \mathit{const}(H') & \text{if } H \Vdash H'
\end{array}$$

Reasoning rules (structural)

READ-ONLY FRAME RULE

$$\frac{\{H \star \mathit{const}(H')\}\ t\ \{Q\} \qquad \mathit{normal}(H')}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

CONSEQUENCE

$$\frac{H \Vdash H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \Vdash Q}{\{H\}\ t\ \{Q\}}$$

DISCARD-PRE

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star \mathrm{GC}\}\ t\ \{Q\}}$$

DISCARD-POST

$$\frac{\{H\}\ t\ \{Q \star \mathrm{GC}\}}{\{H\}\ t\ \{Q\}}$$

EXTRACT-PROP

$$\frac{P \ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}$$

EXTRACT-OR

$$\frac{\{H_1\}\ t\ \{Q\} \qquad \{H_2\}\ t\ \{Q\}}{\{H_1 \vee\!\!\vee H_2\}\ t\ \{Q\}}$$

EXTRACT-EXISTS

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists\!\!\exists x.\, H\}\ t\ \{Q\}}$$

Reasoning rules (syntax-directed)

IF

$$\frac{n \neq 0 \ \Rightarrow\ \{H\}\ t_1\ \{Q\} \qquad n = 0 \ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\text{if } n \text{ then } t_1 \text{ else } t_2)\ \{Q\}}$$

VAL

$$\{[\,]\}\ v\ \{\lambda y.\ [y = v]\}$$

FRAMED SEQUENCING RULE (LET)

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\ x \star H'\}\ t_2\ \{Q\}}{\{H \star H'\}\ (\text{let } x = t_1 \text{ in } t_2)\ \{Q\}}$$

APP

$$\frac{v_1 = \mu f.\lambda x.t \qquad \{H\}\ ([v_1/f]\,[v_2/x]\ t)\ \{Q\}}{\{H\}\ (v_1\ v_2)\ \{Q\}}$$

REF

$$\{[\,]\}\ (\text{ref } v)\ \{\lambda y.\ \exists l.\ [y = l] \star l \hookrightarrow v\}$$

NEW READ AXIOM (GET)

$$\{const(l \hookrightarrow v)\}\ (\text{get } l)\ \{\lambda y.\ [y = v]\}$$

SET

$$\{l \hookrightarrow v'\}\ (\text{set } l\ v)\ \{\lambda y.\ l \hookrightarrow v\}$$

Memories & heaps – a simple model of access rights

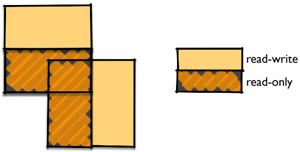
A **memory** is a finite map of locations to values.

A **heap** h is a pair of two **disjoint** memories $h.\mathrm{f}$ and $h.\mathrm{r}$.

- $h.\mathrm{f}$ represents the locations to which we have **full access**;
- $h.\mathrm{r}$ represents the locations to which we have **read-only access**.

An **assertion**, or **permission**, is a predicate over heaps (or: a set of heaps).

Heap composition & separating conjunction

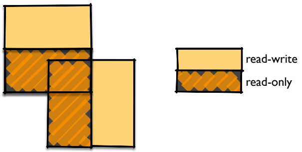


The combination of two heaps:

$$h_1 + h_2 = (h_1.\mathrm{f} \uplus h_2.\mathrm{f},\ h_1.\mathrm{r} \cup h_2.\mathrm{r})$$

is defined only if:

Heap composition & separating conjunction



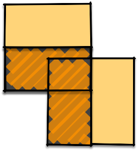
The combination of two heaps:

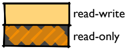
$$h_1 + h_2 = (h_1.\mathrm{f} \uplus h_2.\mathrm{f},\ h_1.\mathrm{r} \cup h_2.\mathrm{r})$$

is defined only if:

- the read-write components $h_1.\mathrm{f}$ and $h_2.\mathrm{f}$ are **disjoint**;

Heap composition & separating conjunction





The combination of two heaps:

$$h_1 + h_2 = (h_1.\mathrm{f} \uplus h_2.\mathrm{f},\ h_1.\mathrm{r} \cup h_2.\mathrm{r})$$

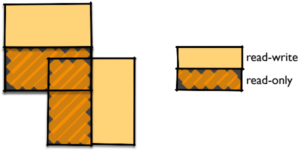
is defined only if:

- the read-write components h_1.f and h_2.f are **disjoint**;
- the read-only components h_1.r and h_2.r **agree** where they overlap;

Heap composition & separating conjunction

The combination of two heaps:

$$h_1 + h_2 = (h_1.\mathrm{f} \uplus h_2.\mathrm{f},\ h_1.\mathrm{r} \cup h_2.\mathrm{r})$$



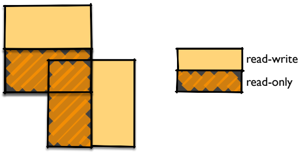
is defined only if:

- the read-write components $h_1.\mathrm{f}$ and $h_2.\mathrm{f}$ are **disjoint**;
- the read-only components $h_1.\mathrm{r}$ and $h_2.\mathrm{r}$ **agree** where they overlap;
- the read-write component $h_1.\mathrm{f}$ is **disjoint** with the read-only component $h_2.\mathrm{r}$, and vice-versa.

Heap composition & separating conjunction

The combination of two heaps:

$$h_1 + h_2 = (h_1.\mathrm{f} \uplus h_2.\mathrm{f},\ h_1.\mathrm{r} \cup h_2.\mathrm{r})$$



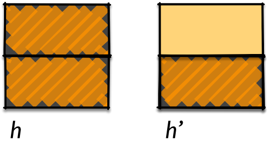
is defined only if:

- the read-write components $h_1.\mathrm{f}$ and $h_2.\mathrm{f}$ are **disjoint**;
- the read-only components $h_1.\mathrm{r}$ and $h_2.\mathrm{r}$ **agree** where they overlap;
- the read-write component $h_1.\mathrm{f}$ is **disjoint** with the read-only component $h_2.\mathrm{r}$, and vice-versa.

With this in mind, **separating conjunction** is interpreted as usual:

$$\begin{aligned} &H_1 \star H_2 = \\ &\quad \lambda h.\ \exists h_1 h_2.\ (h_1 + h_2 \text{ is defined}) \ \wedge\ h = h_1 + h_2 \ \wedge\ H_1\, h_1 \ \wedge\ H_2\, h_2 \end{aligned}$$

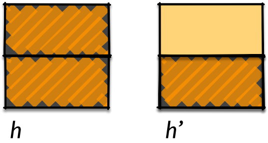
The read-only modality



const(H) is interpreted as follows:

$$
\begin{aligned}
&\mathit{const}(H) = \\
&\quad \lambda h.\ (h.\mathrm{f} = \varnothing)\ \wedge\ \exists h'.\ (h.\mathrm{r} = h'.\mathrm{f} \uplus h'.\mathrm{r}) \wedge H\, h'
\end{aligned}
$$

The read-only modality



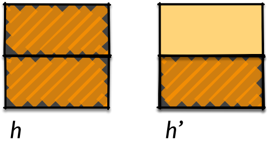
const(H) is interpreted as follows:

$$\begin{array}{l} \mathit{const}(H) = \\ \quad \lambda h.\ (h.\mathrm{f} = \varnothing)\ \wedge\ \exists h'.\ (h.\mathrm{r} = h'.\mathrm{f} \uplus h'.\mathrm{r}) \wedge H\, h' \end{array}$$

This means:

- we have write access to **nothing**.

The read-only modality



const(H) is interpreted as follows:

$$\begin{array}{l} \mathit{const}(H) = \\ \quad \lambda h.\ (h.\mathsf{f} = \varnothing)\ \wedge\ \exists h'.\ (h.\mathsf{r} = h'.\mathsf{f} \uplus h'.\mathsf{r}) \wedge H\, h' \end{array}$$

This means:

- we have write access to **nothing**.
- **if we had write access** to certain locations for which we have read access, **then** H would hold.

The rest of the connectives

$$
\begin{aligned}
[P] &= \lambda h.\ (h.\mathrm{f} = \varnothing) \wedge (h.\mathrm{r} = \varnothing) \wedge P \\
l \hookrightarrow v &= \lambda h.\ (h.\mathrm{f} = (l \mapsto v)) \wedge (h.\mathrm{r} = \varnothing) \\
H_1 \mathbin{\vee\!\!\!\vee} H_2 &= \lambda h.\ H_1\, h \vee H_2\, h \\
\exists\!\!\exists x.\, H &= \lambda h.\ \exists x.\ H\, h \\
\mathrm{normal}(H) &= \forall h.\ H\, h \Rightarrow h.\mathrm{r} = \varnothing
\end{aligned}
$$

Interpretation of triples

The meaning of the Hoare triple $\{H\}\ t\ \{Q\}$ is as follows:

$$\forall h_1 h_2. \left\{ \begin{array}{l} h_1 + h_2 \text{ is defined} \\ H\, h_1 \end{array} \right\} \Rightarrow\ \exists v h_1'. \left\{ \begin{array}{l} h_1' + h_2 \text{ is defined} \\ t / \lfloor h_1 + h_2 \rfloor \Downarrow v / \lfloor h_1' + h_2 \rfloor \\ h_1'.\mathrm{r} = h_1.\mathrm{r} \\ \text{on-some-rw-frag}(Q\, v)\, h_1' \end{array} \right\}$$

What's nonstandard?

Interpretation of triples

The meaning of the Hoare triple $\{H\}\ t\ \{Q\}$ is as follows:

$$\forall h_1\, h_2. \left\{ \begin{array}{l} h_1 + h_2 \text{ is defined} \\ H\, h_1 \end{array} \right\} \Rightarrow \exists v h_1'. \left\{ \begin{array}{l} h_1' + h_2 \text{ is defined} \\ t / \lfloor h_1 + h_2 \rfloor \Downarrow v / \lfloor h_1' + h_2 \rfloor \\ h_1'.\mathrm{r} = h_1.\mathrm{r} \\ \text{on-some-rw-frag}(Q\, v)\, h_1' \end{array} \right\}$$

What's nonstandard?

- The read-only part of the heap must be **preserved**.

Interpretation of triples

The meaning of the Hoare triple $\{H\}\ t\ \{Q\}$ is as follows:

$$\forall h_1 h_2.\left\{\begin{array}{l} h_1 + h_2 \text{ is defined} \\ H\, h_1 \end{array}\right\} \Rightarrow \exists v h_1'.\left\{\begin{array}{l} h_1' + h_2 \text{ is defined} \\ t / \lfloor h_1 + h_2 \rfloor \Downarrow v / \lfloor h_1' + h_2 \rfloor \\ h_1'.\mathrm{r} = h_1.\mathrm{r} \\ \text{on-some-rw-frag}(Q\, v)\, h_1' \end{array}\right\}$$

What's nonstandard?

- The read-only part of the heap must be **preserved**.
- The postcondition describes only **a read-write fragment of the final heap**.

$$\begin{array}{l} \text{on-some-rw-frag}(H) = \\ \qquad \lambda h.\, \exists h_1 h_2.\, (h_1 + h_2 \text{ is defined}) \wedge h = h_1 + h_2 \wedge h_1.\mathrm{r} = \varnothing \wedge H\, h_1 \end{array}$$