Temporary Read-Only Permissions for Separation Logic

Making Separation Logic's
Small Axioms
Smaller

Arthur Charguéraud   François Pottier

Gallium seminar
Paris, April 10, 2017

Separation Logic: to own, or not to own

Separation Logic (Reynolds, 2002) is about disjointness of heap fragments.

- what “we” own, versus what “others” own.

Therefore, it is about unique ownership. A dichotomy arises:

- Of every memory cell, either we have ownership, or we don’t.
- If we do, then we can read and write this cell.
- If we don’t, then we can neither write nor even read this cell.

From memory cells (and arrays), this dichotomy extends to data structures:

> *To a (user-defined) data structure,*
> *either we have no access at all, or we have full read-write access.*

Separation Logic's read and write axioms

The reasoning rule for writing a cell requires (and returns) a full permission:

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

So does the reasoning rule for reading a cell:

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

They are known as "small axioms", because they require minimum permission.

Separation Logic's read and write axioms

The reasoning rule for writing a cell requires (and returns) a full permission:

SET

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

So does the reasoning rule for reading a cell:

terminology:
"permission" = "assertion"

TRADITIONAL READ AXIOM

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

They are known as "small axioms", because they require minimum permission.

Separation Logic's read and write axioms

The reasoning rule for writing a cell requires (and returns) a full permission:

SET

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

So does the reasoning rule for reading a cell:

TRADITIONAL READ AXIOM

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

They are known as "small axioms", because they require minimum permission.

But are they as small as they could be?

Isn't it excessive for reading to require a full permission?

Are there adverse consequences of working with such coarse permissions?

The problem

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

Here is a typical specification of sequence concatenation:

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

The problem

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

Here is a typical specification of sequence concatenation:

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

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 throughout the proof of *append*.
- It actually does not guarantee that s_1 and s_2 are unmodified. — (next slide)
- It requires s_1 and s_2 to be distinct data structures. — (slide after next)

The problem, facet 3: temporary modifications are not forbidden

Repeating "$s \rightsquigarrow \mathsf{Seq}\, L$" 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:

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

The problem, facet 3: 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:

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

The problem, facet 4: sharing is not permitted

The specification of *append* requires s_1 and s_2 to be distinct data structures:

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

Indeed, $s \leadsto \mathsf{Seq}\, L \,\star\, s \leadsto \mathsf{Seq}\, L$ is equivalent to *false*.

The problem, facet 4: sharing is not permitted

The specification of *append* requires s_1 and s_2 to be distinct data structures:

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

Indeed, $s \rightsquigarrow \mathrm{Seq}\,L \;\star\; s \rightsquigarrow \mathrm{Seq}\,L$ is equivalent to *false*.

As a result, to allow sharing, we must establish another specification:

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

Duplicate work for us. Increased complication and/or duplicate work for the user.

Fractional permissions to the rescue...?

Could sequence concatenation be specified as follows in Concurrent SL?

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

Yes, if the logic allows scaling, $\pi \cdot H$. This requires existential quantification to be restricted so as to be precise (Boyland, 2010).

Without scaling, one must define $s \leadsto \mathrm{Seq}\, \pi\ L$.

Fractional permissions to the rescue...?

Could sequence concatenation be specified as follows in Concurrent SL?

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

Yes, if the logic allows scaling, $\pi \cdot H$. This requires existential quantification to be restricted so as to be precise (Boyland, 2010).

Without scaling, one must define $s \leadsto \mathrm{Seq}\, \pi\, L$.

This addresses problem facets 3 and 4,

- but is still noisy,
- and still requires careful splitting, threading, and joining of permissions.

“Hiding the fractions” (Heule et al, 2013) is cool but requires yet more machinery.

In this paper

We propose a solution that is:

- not as powerful as fractional permissions (or other share algebras),
- but significantly simpler.

Our contributions:

- introducing a read-only modality, RO.
  RO(H) represents temporary read-only access to the memory governed by H.
- finding simple and sound reasoning rules for RO.
- 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:

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

Our solution

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

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

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 the earlier spec. — (slide after next)

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.

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

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.

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

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*.

$$
\dfrac{\dfrac{\begin{array}{l}\{\ \mathrm{RO}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \mathrm{RO}(s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\ \}\\ (\mathit{append}\, s_1\, s_2)\\ \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}\end{array}}{\begin{array}{l}\{\ \mathrm{RO}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\ \}\\ (\mathit{append}\, s_1\, s_2)\\ \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2)\ \}\end{array}}\ \textsf{CONSEQUENCE}}{\begin{array}{l}\{\ s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\ \}\\ (\mathit{append}\, s_1\, s_2)\\ \{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+ L_2) \star s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\ \}\end{array}}\ \textsf{READ-ONLY FRAME}
$$

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

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

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 weakened

$$\dfrac{\dfrac{\begin{array}{l}\{\ \mathrm{RO}(s_1 \leadsto \mathrm{Seq}\, L_1) \star \mathrm{RO}(s_2 \leadsto \mathrm{Seq}\, L_2)\ \}\\ (\mathit{append}\, s_1\ s_2)\\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\ \}\end{array}}{\begin{array}{l}\{\ \mathrm{RO}(s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2)\ \}\\ (\mathit{append}\, s_1\ s_2)\\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2)\ \}\end{array}}\ \text{CONSEQUENCE}}{\begin{array}{l}\{\ s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\ \}\\ (\mathit{append}\, s_1\ s_2)\\ \{\ \lambda s_3.\ s_3 \leadsto \mathrm{Seq}\,(L_1 \mathbin{+\!\!+} L_2) \star s_1 \leadsto \mathrm{Seq}\, L_1 \star s_2 \leadsto \mathrm{Seq}\, L_2\ \}\end{array}}\ \text{READ-ONLY FRAME}$$

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*.

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

***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*.

$$
\dfrac{
\dfrac{
\begin{array}{l}
\{\ \mathrm{RO}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1) \star \mathrm{RO}(s_2 \rightsquigarrow \mathrm{Seq}\, L_2)\ \} \\
(\mathit{append}\, s_1\, s_2) \\
\{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+\, L_2)\ \}
\end{array}
}{
\begin{array}{l}
\{\ \mathrm{RO}(s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L \\
(\mathit{append}\, s_1\, s_2) \\
\{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+\, L_2)\ \}
\end{array}
}\ \text{CONSEQUENCE}
}{
\begin{array}{l}
\{\ s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\ \} \\
(\mathit{append}\, s_1\, s_2) \\
\{\ \lambda s_3.\ s_3 \rightsquigarrow \mathrm{Seq}\,(L_1 +\!\!+\, L_2) \star s_1 \rightsquigarrow \mathrm{Seq}\, L_1 \star s_2 \rightsquigarrow \mathrm{Seq}\, L_2\ \}
\end{array}
}\ \text{READ-ONLY FRAME}
$$

read-write permission magically re-appears

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

Some Intuition

Reasoning Rules

Model

Conclusion

Permissions

The syntax of permissions is as follows:

$$H \quad := \quad [P] \mid l \hookrightarrow v \mid H_1 \star H_2 \mid H_1 \Cup H_2 \mid \exists x.\, H \mid \mathrm{RO}(H)$$

Every permission H has a read-only form $\mathrm{RO}(H)$.

Properties of RO

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

$$\mathrm{RO}(H_1 \star H_2) \quad \triangleright \quad \mathrm{RO}(H_1) \star \mathrm{RO}(H_2) \qquad \text{(the reverse is false)}$$

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

$$\mathrm{RO}(H) \quad = \quad \mathrm{RO}(H) \star \mathrm{RO}(H)$$

Read-only permissions are generally well-behaved:

$$\begin{array}{rcll}
\mathrm{RO}([P]) & = & [P] & \\
\mathrm{RO}(H_1 \mathbin{\text{\textbackslash\!\!/}} H_2) & = & \mathrm{RO}(H_1) \mathbin{\text{\textbackslash\!\!/}} \mathrm{RO}(H_2) & \\
\mathrm{RO}(\exists\!\exists x.\, H) & = & \exists\!\exists x.\, \mathrm{RO}(H) & \\
\mathrm{RO}(\mathrm{RO}(H)) & = & \mathrm{RO}(H) & \\
\mathrm{RO}(H) & \triangleright & \mathrm{RO}(H') & \qquad\qquad \text{if } H \triangleright H'
\end{array}$$

A new read axiom

The traditional read axiom:

TRADITIONAL READ AXIOM

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

is replaced with a “smaller” axiom:

NEW READ AXIOM

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

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":

$$\frac{\{H\}\ t\ \{Q\} \qquad \text{normal}\ H'}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{Frame rule} \qquad \frac{\{H \star \mathrm{RO}(H')\}\ t\ \{Q\} \qquad \text{normal}\ H'}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{Read-only frame rule}$$

This says: upon entry into a block, H' is temporarily replaced with $\mathrm{RO}(H')$, and upon exit, magically re-appears.

The side condition normal H' means roughly that H' has no RO components.

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

If they could, the read-only frame rule would clearly be unsound. (Exercise!)

How read-only permissions are used

No reasoning rule involves a triple whose postcondition contains RO.

Read-only permissions always appear in preconditions, never in postconditions.

They are always passed down, never returned.

In fact, in the model, we will see that, in a triple $\{H\}\ t\ \{Q\}$:

- the postcondition applies only to some read-write fragment of the final heap;
- the read-only part of the heap must be preserved anyway,
  so there is no need for the postcondition to describe it.

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.f and h.r.

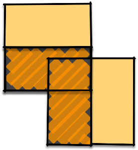
- h.f represents the locations to which we have full access;
- h.r represents the locations to which we have read-only access.

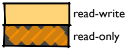
An assertion, or permission, is a predicate over heaps (or: a set of heaps).

Heap composition & separating conjunction

The combination of two heaps:

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



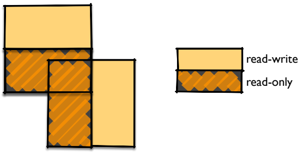


is defined only if:

Heap composition & separating conjunction

The combination of two heaps:

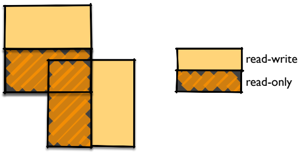
$$h_1 + h_2 = (h_1.\mathrm{f} \uplus h_2.\mathrm{f},\ h_1.\mathrm{r} \cup h_2.\mathrm{r})$$



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:

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

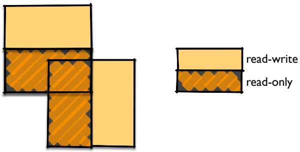
is defined only if:

- the read-write components $h_1.\mathrm{f}$ and $h_2.\mathrm{f}$ are disjoint;
- the read-only components $h_1.\mathrm{r}$ and $h_2.\mathrm{r}$ agree where they overlap;

Heap composition & separating conjunction

The combination of two heaps:

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

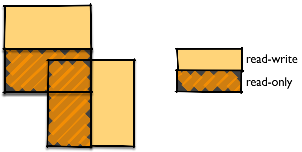


is defined only if:

- 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;
- the read-write component h_1.f is disjoint with the read-only component h_2.r, and vice-versa.

Heap composition & separating conjunction

The combination of two heaps:



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

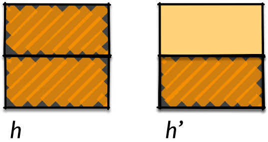
is defined only if:

- 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:

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

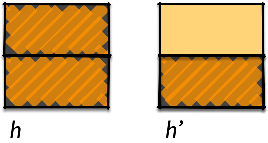
The read-only modality



RO(H) is interpreted as follows:

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

The read-only modality



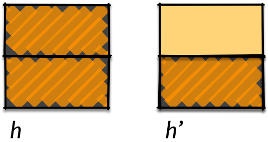
RO(H) is interpreted as follows:

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

This means:

- we have write access to nothing.

The read-only modality



RO(H) is interpreted as follows:

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

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

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

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.

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

Soundness

Theorem

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

Proof.

“Straightforward”. Machine-checked. □

We propose:

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

We believe that temporary read-only permissions sometimes help state more concise, accurate, useful specifications, and lead to simpler proofs.

Possible future work: an implementation in CFML (Charguéraud).

Amnesia (1/2)

Suppose *population* has this “RO” specification:

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

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

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

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

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

If “RO” 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 $\text{let } foo\ h =$

2 $\quad \{h \leadsto \text{HashTable } M\}$ — *foo*'s precondition

3 $\quad \{h \leadsto \{data = a; \ldots\} \star a \leadsto \text{Array } L \star \ldots\}$ — by unfolding

4 $\quad \text{let } d = h.data \text{ in}$

5 $\quad \{h \leadsto \{data = a; \ldots\} \star a \leadsto \text{Array } L \star \ldots \star [d = a]\}$ — by reading

6 $\quad \{h \leadsto \text{HashTable } M \star [d = a]\}$ — by folding

7 $\quad \text{let } p = population\ h \text{ in}$ — we have to fold

8 $\quad \{h \leadsto \text{HashTable } M \star [d = a] \star [p = \#M]\}$

9 $\quad \ldots$

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

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