An Introduction to Separation Logic, and the Benefits of going Higher-order (A Tutorial)

Lars Birkedal

Logic and Semantics Group
Dept. of Comp. Science, Aarhus University

April 2013
Separation Logic

- Program Logic a la Hoare Logic for reasoning about programs with pointers (or references to shared mutable data) [Reynolds, O’Hearn, . . . , 2000+]
- Main feature: it facilitates modular reasoning, formalized via so-called frame rule, using a connective called separating conjunction.
Hoare Logic - a recap

- Programming Language: imperative while-language
- Assertion Language: first-order logic w. equality
- Specifications for partial correctness:
  - \( \{P\} C \{Q\} \)
  - if \( s \vdash P \) and \( C \), \( s \rightarrow s' \), then \( s' \vdash Q \).
- Rules for deriving specifications (including rules for first-order logic).
Separation Logic

- Programming Language: as before but now with C-like pointers
- Specifications for partial correctness:
  - $\{P\} C \{Q\}$
  - if $s, h \models P$ and $C, s, h \rightarrow s', h'$, then $s', h' \models Q$.
- Assertion Language: first-order logic w. equality + BI connectives:
  - $s, h \models \text{emp}$ iff $h$ is the empty heap
  - $s, h \models x \mapsto 5$ iff $h$ is the singleton heap with one location $s(x)$ with value 5.
  - $s, h \models P \ast Q$ iff $h$ can be split into $h_1$ and $h_2$, with disjoint domains, such that $s, h_1 \models P$ and $s, h_2 \models Q$.
  - $s, h \models P \rightarrow Q$ iff . . .
Examples of “small” axioms

\[
\begin{align*}
\{x \mapsto \bot\} \text{dispose}(x) \{\text{emp}\} \\
\{x \mapsto \bot\} [x] := e \{x \mapsto e\}
\end{align*}
\]

Modular Reasoning via Frame Rule:

\[
\begin{align*}
\{P\} C \{Q\} \\
\{P \star R\} C \{Q \star R\}
\end{align*}
\]

(assuming \(\text{Mod}(C) \cap \text{FV}(R) = \emptyset\)).
Example: in-place list reversal

- Linked list of cons-cells.
- Program: $reverse =

\[ j := \text{nil}; \text{while } i \neq \text{nil} \text{ do } (k := [i+1]; [i+1] := j; j := i; i := k) \]

- Local specification:

\[ \{ \text{list}(\alpha, i) \} \text{ reverse } \{ \text{list}(\text{rev}(\alpha), j) \} \]

- where $\text{list}(\alpha, i)$ is def’ by ind. on sequence $\alpha$:

\[
\begin{align*}
\text{list}([], i) & \overset{\text{def}}{=} i = \text{nil} \land \text{emp} \\
\text{list}(n :: \alpha, i) & \overset{\text{def}}{=} \exists j. i \mapsto (n, j) \ast \text{list}(\alpha, j).
\end{align*}
\]
Example: in-place list reversal, II

Points to notice:

- *Local reasoning*: the precondition \( \text{list}(\alpha, i) \) only describes resources needed by \text{reverse} (the *footprint* of \text{reverse})*
- Having proved local spec, frame rule gives, e.g.:

\[
\{ \text{list}(\alpha, i) \ldots \text{list}(\beta, k) \} \text{ reverse } \{ \text{list}(\text{rev}(\alpha), j) \ldots \text{list}(\beta, k) \}
\]

Hence we can (re-)use the specification in larger contexts.

In summa: standard separation logic makes it easy to reason about pointer programs when we can find some way to separate data structures into *disjoint* parts.
Example: Cheney’s Copying GC

[Torp-Smith, Birkedal, Reynolds, POPL’2004]
Model and Soundness

- A set $\llbracket Val \rrbracket$ interpreting the type $Val$ of values
- A set $\llbracket Loc \rrbracket$ of locations such that $\llbracket Loc \rrbracket \subseteq \llbracket Val \rrbracket$
- A set $H = \llbracket Loc \rrbracket \rightarrow_{fin} \llbracket Val \rrbracket$ of heaps (finite partial functions)
- Partial binary operation $*$ on heaps:

$$ h_1 \ast h_2 = \begin{cases} h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \text{undefined} & \text{otherwise,} \end{cases} $$

where $h_1 \# h_2$ iff $\text{Dom}(h_1) \cap \text{Dom}(h_2) = \emptyset$

- Given stack $s : \text{Var} \rightarrow_{fin} \llbracket Val \rrbracket$, terms interpreted as usual:

$$ \llbracket x \rrbracket s = s(x) $$
$$ \llbracket n \rrbracket s = \llbracket n \rrbracket $$
$$ \llbracket t_1 \pm t_2 \rrbracket s = \llbracket t_1 \rrbracket s \pm \llbracket t_2 \rrbracket s $$

...
Interpretation of assertions

$s, h \models \phi$, where $FV(\phi) \subseteq Dom(s)$:

$s, h \models t_1 = t_2$ iff $\llbracket t_1 \rrbracket s = \llbracket t_2 \rrbracket s$

$s, h \models t_1 \leftrightarrow t_2$ iff $Dom(h) = \{ \llbracket t_1 \rrbracket s \}$ and $h(\llbracket t_1 \rrbracket s) = \llbracket t_2 \rrbracket s$

$s, h \models \text{emp}$ iff $h = \emptyset$

$s, h \models \top$ always

$s, h \models \bot$ never

$s, h \models \phi \ast \psi$ iff there exists $h_1, h_2 \in H$ such that $h_1 \ast h_2 = h$ and $s, h_1 \models \phi$ and $s, h_2 \models \psi$

$s, h \models \phi \rightarrow \ast \psi$ iff for all $h', h' \neq h$ and $s, h' \models \phi$ implies $s, h \ast h' \models \psi$

$s, h \models \phi \lor \psi$ iff $s, h \models \phi$ or $s, h \models \psi$

$s, h \models \phi \land \psi$ iff $s, h \models \phi$ and $s, h \models \psi$

$s, h \models \phi \rightarrow \psi$ iff $s, h \models \phi$ implies $s, h \models \psi$

$s, h \models \forall x. \phi$ iff for all $v \in \llbracket \text{Val} \rrbracket$, $s[x \mapsto v], h \models \phi$

$s, h \models \exists x. \phi$ iff there exists $v \in \llbracket \text{Val} \rrbracket$, such that $s[x \mapsto v], h \models \phi$
Sound Proof Rules

Standard predicate logic + rules from bunched implications:

\[
(\phi \ast \psi) \ast \theta \vdash_\Gamma \phi \ast (\psi \ast \theta) \quad \phi \ast (\psi \ast \theta) \vdash_\Gamma (\phi \ast \psi) \ast \theta
\]

\[
\vdash_\Gamma \phi \leftrightarrow \phi \ast \text{emp} \quad \phi \ast \psi \vdash_\Gamma \psi \ast \phi
\]

\[
\phi \vdash_\Gamma \psi \quad \theta \vdash_\Gamma \omega \\
\phi \ast \theta \vdash_\Gamma \psi \ast \omega
\]

\[
\phi \ast \psi \vdash_\Gamma \theta \\
\phi \vdash_\Gamma \psi \rightarrow \ast \theta
\]

- BI, a substructural logic, related to linear logic.
- See [O’Hearn, Pym: The Logic of Bunched Implications, BSL 1999]
Specification logic

\[
\begin{align*}
\lbrack \Delta_l; \Delta_p \vdash \{ P \} \ c \ \{ Q \} \rbrack(s_l) & \iff \\
\forall s_p \in \lbrack \Delta_p \rbrack. \forall h \in \lbrack \Delta_l, \Delta_p \vdash P \rbrack(s_l, s_p). \\
(c, s_p, h) \ is \ safe \land \\
(c, s_p, h) \Downarrow (s'_p, h') \Rightarrow h' \in \lbrack \Delta \vdash Q \rbrack(s_l, s'_p)
\end{align*}
\]

\[
\lbrack \Delta_l; \Delta_p \vdash \forall x : \tau. \ delta \rbrack(s_l) \iff \\
\lbrack \Delta_l; \Delta_p \vdash \delta \rbrack(s_l)\lbrack x \mapsto v \rbrack \ for \ all \ v \in \lbrack \tau \rbrack.
\]

\[
\ldots
\]
Proof Rules

Judgements of form $\Delta_l; \Delta_p \vdash \Gamma \vdash \delta$, with logical variables $\Delta_l$, program variables $\Delta_p$, specification logic assumptions $\Gamma$, and specification logic formula $\delta$.

Rules for commands, e.g.,

$$\Delta_l; \Delta_p \vdash \{E \mapsto -\} [E] := E'[E \mapsto E']$$

Rule of Consequence:

$$\Delta_l; \Delta_p \vdash P \rightarrow P' \quad \Delta_l; \Delta_p \vdash \{P'\} c \{Q'\} \quad \Delta_l; \Delta_p \vdash Q' \rightarrow Q$$

$$\Delta_l; \Delta_p \vdash \{P\} c \{Q\}$$

Frame Rule:

$$\Delta_l; \Delta_p \vdash \{P\} c\{Q\}$$

$$\Delta_l; \Delta_p \vdash \{P * R\} c \{Q * R\} \quad \text{Mod}(c) \cap \text{FV}(R) = \emptyset$$

Forall Rule:

$$\Delta_l, x:\tau; \Delta_p \vdash \Gamma \vdash \delta$$

$$\Delta_l; \Delta_p \vdash \forall x:\tau. \delta \quad x \notin \text{FV}(\Gamma)$$
Soundness of proof rules

In traditional separation logic:

- Mostly straightforward.
- Frame rule relies on properties of operational semantics
  
  **Safety Monotonicity** For all $c, s, h$, if $(c, s, h)$ is safe, then for all heaps $h'$ s.t. $h' \neq h$, $(c, s, h \ast h')$ is also safe.
  
  **Frame Property** For all $c, s, h$, if $(c, s, h)$ is safe and $h' \neq h$, then $(c, s, h \ast h') \Downarrow (s', h'')$, implies that there is $h_0$ disjoint from $h'$ such that $h'' = h_0 \ast h'$ and $(c, s, h) \Downarrow (s', h_0)$. (relies on nondeterministic memory allocation).
Variations

- Intuitionistic models (assertions upwards closed wrt. heap extension) for languages with no explicit deallocation.
- Permission models (fractions, etc.) for more fine-grained notions of separation, allowing multiple readers of same location.
- ...
Modular Reasoning for Modules

- Let us call separation logic as above for \textit{first-order separation logic} and the frame rule for \textit{first-order frame rule}.
- As we have seen, first-order separation logic provides modular reasoning for first-order programs.
- What about second-order or higher-order programs? (programming languages with some kind of module facility).
- Example: a program that uses a stack module.
- Ideally, a client of a stack module should not know about how the stack module is implemented.
- So seek logic that supports reasoning about clients without revealing info about module implementation.
- Two lines of development
  - higher-order separation logic supporting data abstraction
  - separation logic with higher-order frame rules for hiding
Higher-Order Separation Logic Example

Stack ADT

\[
\text{stackspec} = \\
\exists \alpha : \text{Type}. \exists \text{inv} \in \alpha \times \mathbb{N} \text{list} \to \text{Prop}.
\]

\[
\{ \text{emp} \} \text{new()} \{ s : \alpha. \text{inv}(s, []) \} \times \\
\forall s : \alpha. \forall x : \mathbb{N}. \forall l : \mathbb{N} \text{list} \\
\{ \text{inv}(s, l) \} \text{push}(s, x) \{ \text{inv}(s, x :: l) \} \times \\
\forall s : \alpha. \forall x : \mathbb{N}. \forall l : \mathbb{N} \text{list} \\
\{ \text{inv}(s, x :: l) \} \text{pop}(s) \{ y : \mathbb{N}. \text{inv}(s, l) \land y = x \}.
\]

- Modularity: clients can use the spec without knowing anything about how the stack is implemented (since abstract in the inv predicate).
- Different stack implementations can meet this spec.
- Presented at ESOP 2005 [Biering, Birkedal, Torp-Smith] (second-order (“abstract predicates”) version independently by Parkinson-Biermann, POPL 2005, for OO language)
Models of HOSL: BI-Hyperdoctrines

- Focus first on extending the assertion logic to higher-order logic.
- As suggested by the example, have in mind many-typed formulation with

\[ \tau ::= \text{Val} \mid \ldots \mid 1 \mid \tau \times \tau \mid \tau \to \tau \mid \text{Prop} \]

- Seek an approach that covers the many variations needed in applications.
- So let’s pause and think about abstract models of higher-order logic.
BI Hyperdoctrines — Overview

- A hyperdoctrine is a categorical formalization of a model of predicate logic [Lawvere 1969]. Sound and complete for IHOL.
- Toposes also sound and complete for IHOL.
- BI Hyperdoctrines sound and complete for IHOL + BI
Let \( \mathcal{C} \) be a category with finite products. A \textit{first-order hyperdoctrine} \( \mathcal{P} \) over \( \mathcal{C} \) is a contravariant functor \( \mathcal{P} : \mathcal{C}^{\text{op}} \rightarrow \text{Poset} \) s.t.:

- Each \( \mathcal{P}(X) \) is a Heyting algebra.
- Each \( \mathcal{P}(f) : \mathcal{P}(Y) \rightarrow \mathcal{P}(X) \) is a Heyting algebra homomorphism.
- There is an element \( =_X \) of \( \mathcal{P}(X \times X) \) satisfying that for all \( A \in \mathcal{P}(X \times X) \),

\[
\top \leq \mathcal{P}(\Delta_X)(A) \quad \text{iff} \quad =_X \leq A.
\]
For each product projection \( \pi : \Gamma \times X \to \Gamma \) in \( C \),
\( \mathcal{P}(\pi) : \mathcal{P}(\Gamma) \to \mathcal{P}(\Gamma \times X) \) has both a left adjoint \( (\exists X)_{\Gamma} \) and a right adjoint \( (\forall X)_{\Gamma} \):

\[
A \leq \mathcal{P}(\pi)(A') \quad \text{if and only if} \quad (\exists X)_{\Gamma}(A) \leq A'
\]

\[
\mathcal{P}(\pi)(A') \leq A \quad \text{if and only if} \quad A' \leq (\forall X)_{\Gamma}(A).
\]

Natural in \( \Gamma \).
Types and terms interpreted by objects and morphisms of $C$

Each formula $\phi$ with free variables in $\Gamma$ is interpreted as a $\mathcal{P}$-predicate $\llbracket \phi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket)$ by induction on the structure of $\phi$ using defining properties of hyperdoctrine.

A formula $\phi$ with free variables in $\Gamma$ is satisfied if $\llbracket \phi \rrbracket$ is $\top$ in $\mathcal{P}(\llbracket \Gamma \rrbracket)$.

Sound and complete for intuitionistic predicate logic.

A first-order hyperdoctrine is sound for classical predicate logic in case all the fibres $\mathcal{P}(X)$ are Boolean algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean algebra homomorphisms.
Hyperdoctrines

A (general) *hyperdoctrine* is a first-order hyperdoctrine with the following additional properties:

- $\mathcal{C}$ is cartesian closed; and
- there is $H \in \mathcal{C}$ and a natural bijection $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, H)$.

Cartesian closure interprets higher types. Type of propositions is interpreted by $H$.

Basic Example:

- $\mathcal{C} = \text{Set}$
- $\mathcal{P}(X) = \text{Sub}(X)$.
- $H = 2$
Recall: A *BI algebra* is a Heyting algebra, which has an additional symmetric monoidal closed structure \((\text{emp}, \ast, \neg\ast)\) (i.e., satisfying the rules shown earlier).

Define: A first-order hyperdoctrine \(\mathcal{P}\) over \(\mathcal{C}\) is a *first-order BI hyperdoctrine* in case
- all the fibres \(\mathcal{P}(X)\) are BI algebras, and
- all the reindexing functions \(\mathcal{P}(f)\) are BI algebra homomorphisms

Likewise for general BI hyperdoctrines.
First-Order Predicate BI

- Predicate logic with equality extended with \( \text{emp}, \phi \ast \psi, \phi \rightarrow \ast \psi \) satisfying the rules shown earlier.
- **Theorem** The interpretation of first-order predicate BI is sound and complete.
- Also for classical predicate BI, of course
- Higher-order Predicate BI
  - Higher-order predicate logic extended with BI as above.
  - BI hyperdoctrines sound and complete class of models.
Example of BI hyperdoctrine

Let $B$ be a complete BI algebra. Define Set-indexed BI hyperdoctrine:

- $\mathcal{P}(X) = B^X$, functions from $X$ to $B$, ordered pointwise
- For $f : X \to Y$, $\mathcal{P}(f) : B^Y \to B^X$ is comp. with $f$.
- $=_X (x, x')$ is $\top$ if $x = x'$, otherwise $\bot$.
- Quantification: for $A \in B^{\Gamma \times X}$

\[
(\exists X)_\Gamma (A) \overset{\text{def}}{=} \lambda i \in \Gamma. \bigvee_{x \in X} A(i, x)
\]

\[
(\forall X)_\Gamma (A) \overset{\text{def}}{=} \lambda i \in \Gamma. \bigwedge_{x \in X} A(i, x)
\]

in $B^\Gamma$. 
Earlier work showed how to use some toposes to model propositional BI ($\text{Sub}_E(1)$ is a BI-algebra, for certain $E$).

Toposes model (higher-order) predicate logic, since $\text{Sub}_E$ is a hyperdoctrine.

But, surprise, we cannot interpret predicate BI in toposes:

**Theorem** Let $E$ be a topos and suppose $\text{Sub}_E : E^{\text{op}} \to \text{Poset}$ is a BI hyperdoctrine. Then the BI structure on each lattice $\text{Sub}_E(X)$ is trivial, i.e., for all $\varphi, \psi \in \text{Sub}_E(X)$, $\varphi \star \psi \leftrightarrow \varphi \land \psi$. 
Separation Logic as a BI Hyp.

- $\mathcal{P}(H)$ is a complete Boolean BI algebra, ordered by inclusion.
- Let $S$ be the BI hyperdoctrine induced by the complete Boolean BI algebra.
- **Theorem** $h \in \llbracket \phi \rrbracket(v_1, \ldots, v_n)$ iff $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$, $h \models \phi$.
- (also works for other models of separation logic, e.g., intuitionistic and permissions models)
- The BI hyperdoctrine $S$ also gives a model of higher-order separation logic, with $\mathcal{P}(H)$ the set of truth values.
- Really, we have only talked about the assertion logic of separation logic.
- But, since types and terms in the specification logic are interpreted as sets, and $\mathcal{P}(H)$ is a set, we can also quantify over $\text{Prop}$ in the specification logic, as used in the stack example.
Applications of HOSL

- Data abstraction, cf. stack example from above.
- Formalization of separation logic
  - applications (e.g., proof of Cheney GC) used various extensions of separation logic, with relations, trees, etc.
  - point is that they are all definable in higher-order logic, no need for ad-hoc extensions:
  - Let $2 = \{\bot, \top\}$. There is a canonical map $\iota : 2 \to \mathcal{P}(H)$. Say $\phi : X \to \mathcal{P}(H)$ is pure if there is a map $\chi_\phi : X \to 2$ s.t. $\phi = \iota \circ \chi_\phi$.
  - The sub-logic of pure predicates is simply the standard classical higher-order logic of Set.
  - Allows to use classical higher-order logic for defining lists, trees, etc.
  - In particular, recursive definitions of predicates, earlier done at the meta-level, can now be done inside the higher-order logic itself.
Logical characterizations of classes of formulas, e.g.,

Traditional definition of a *precise*: $q$ is precise iff, for $s, h$, there is at most one subheap $h_0$ of $h$ such that $s, h_0 \models q$.

**Prop.** $q$ is precise iff

$$\forall p_1, p_2 : Prop. (p_1 \ast q) \land (p_2 \ast q) \rightarrow (p_1 \land p_2) \ast q$$

is valid in the BI hyperdoctrine $S$.

Thus: can make *logical* proofs about precise formulas.
(Returning to program proving applications)

- General (generic / polymorphic) specifications and proofs of polymorphic programs
- Parameterized list predicate:

  $$\text{plist}(P, [], i) \overset{\text{def}}{=} i = \text{nil} \land \text{emp}$$

  $$\text{plist}(P, x :: \beta, i) \overset{\text{def}}{=} \exists j. i \mapsto (x, j) \ast P(x) \ast \text{plist}(P, \beta, j)$$

- Generic specification:

  $$\beta : \text{seqInt} \vdash \forall P : \text{Prop}^{\text{Int}}. \{\text{plist}(P, \beta, i)\} \text{ reverse } \{\text{plist}(P, \beta^\dagger, j)\},$$

- Point: one general spec (hence one proof of implementation code) that can be instantiated at will by clients.
Higher-Order Frame Rules (a sketch)

*Hiding* the resource invariant.

- **Abstract stack spec:**

\[
\begin{align*}
\text{stackspec} &= \\
&\forall P : \mathbb{N} \rightarrow \text{Prop}.
\forall x : \mathbb{N}. \{ P(x) \} \text{push}(x) \{ \text{emp} \} \land \\
&\{ \text{emp} \} \text{pop}(). \{ r : 1 + \mathbb{N}. \ (\exists n. r = \text{inr}(n) \rightarrow P(n)) \land \\
&(r = \text{inl}(\ast) \rightarrow \text{emp}) \}.\end{align*}
\]

- **Proof of client relative to abstract spec:**

\[
\text{stackspec} \vdash \text{client}
\]
Apply HO-frame rule:

\[
(stackspec \vdash \text{client}) \otimes inv
\]

Infer

\[
stackspec \otimes inv \vdash \text{client} \otimes inv
\]

and then distribute \(inv\) over pre- and post-conditions in stackspec to get

\[
\{ P(x) \ast inv \} \text{push}(x) \{ emp \ast inv \} \wedge \\
\{ emp \ast inv \}_{\text{pop}}(r : 1 + N. \ (\exists n. r = \text{inr}(n) \rightarrow P(n) \ast inv) \wedge \} \\
(r = \text{inl}(*) \rightarrow emp \ast inv)\}.
\]

that matches an implementation with resource invariant \(inv\).
Analogy and references

- HOSL approach: polymorphic types
- HOFR approach: subtyping
- (analogy not formalized)
- Second-order frame rule [O’Hearn, Yang, Reynolds, POPL 2004]
- Higher-order frame rules [Birkedal, Torp-Smith, Yang, LICS 2005]
- For recent account for higher-order language, with bells-and-whistles, see [Schwinghammer, Birkedal, Pottier, Reus, Stovring, Yang: A step-indexed Kripke model of hidden state. Mathematical Structures in Computer Science 23(1): 1-54 (2013)]
- HOFR also useful for structuring reasoning about assembly code, see [Jensen, Benton, Kennedy: High-level Separation Logic for Low-level Code, POPL-2013]
What is involved in models of HOFR?

- Two technical points regarding models of HOFR that have proved useful elsewhere.

1. For prog. lang. with higher-order function: how to express that programs behave “locally” (safety monotonicity and frame property)?
   - We do not, we just require that all proved programs satisfy the frame rule (bake it in to the interpretation). Analogous to how parametric models of System F are constructed. First done in [LICS 2005] paper mentioned above.

2. Hiding / HOFR rule done via Kripke model (world consists of the hidden resources that programs should preserve).
   - With higher-order store, the set of worlds is recursively defined in CBUlt (see [Birkedal et. al.: Step-Indexed Kripke Models over Recursive Worlds, POPL’2011] and [MSCS-2013] paper mentioned above).
Recursion / Step-Indexing

- Need some mathematical technique to establish soundness of proof rules for recursive functions.
- For instance, domain theory or step-indexing.
- Omitted today (step-indexing will be discussed in talk on Thursday).
Views

(a glimpse)

- General framework for compositional reasoning about concurrency
- Framing and compositionality for thread-local reasoning captured via \( \ast \).
- Instances cover (concurrent) separation logic, rely-guarantee, combinations of separation logic and rely-guarantee, e.g., concurrent abstract predicates. Also Kripke models of type systems.
Separation Logic instance of Views
(disregarding stacks for notational simplicity):

- **View** = \( P(\Sigma) \), \( \Sigma = H \) separation algebra.
- Concrete machine states \( S = H \), heaps.
- Reification map: \( I : \text{View} \to P(S), I = h \mapsto \{h\} \).
- Semantics of triples \( \{p\} \ c \ \{q\} \):
  \[
  \forall r \in \text{View}. \forall x \in p \ast r. \forall s \in I(x). \\
  c, s \downarrow s' \Rightarrow \\
  \exists y \in q \ast r. s' \in I(y)
  \]

- Observe:
  - Quantification over frames ("baking-in" the frame rule).
  - Standard operational semantics.

- Instrumented states in Views as needed for logical reasoning.

- Reference: [Dinsdale-Young, Birkedal, Gardner, Parkinson, Yang: Views: Compositional Reasoning for Concurrency. POPL-2013]
Summary

- First-order standard separation logic for modular reasoning “in the small”
- Higher-order separation / higher-order frame rules for modular reasoning “in the large” (for programs composed of modules)
- Mostly an overview
- But some details about how to construct models, which can be of help when looking at the literature, both “old style” and “new style”.
Thank you for your attention.