Iris: Higher-Order Concurrent Separation Logic

Lecture 9: Concurrency Intro and Invariants

Lars Birkedal

Aarhus University, Denmark

November 21, 2017
Overview

Earlier:
- Operational Semantics of $\lambda_{\text{ref,conc}}$
  - $e, (h, e) \rightsquigarrow (h, e')$, and $(h, \mathcal{E}) \rightarrow (h', \mathcal{E}')$
- Basic Logic of Resources
  - $l \hookrightarrow v, P \ast Q, P \ast\ast Q, \Gamma \mid P \vdash Q$
- Basic Separation Logic
  - $\{P\} e \{v.Q\} : \text{Prop, isList } l \ x s, \text{ADTs, foldr}$
- Later ($\triangleright$) and Persistent ($\Box$) Modalities.

Today:
- Concurrency Intro: $e_1 \parallel e_2$
- Invariants: $P^l$
- Key Points:
  - Thread-local reasoning.
  - Disjoint concurrency rule for $e_1 \parallel e_2$
  - Invariants for sharing of resources among threads $e_1$ and $e_2$ in $e_1 \parallel e_2$. 
To start off with simpler proof rules, we first define a programming language construct for parallel execution of two expressions $e_1$ and $e_2$.

- $e_1 || e_2$ runs $e_1$ and $e_2$ in parallel, waits until both finish, and then returns a pair consisting of the values to which $e_1$ and $e_2$ evaluated.

- Definable using `fork`. First we define `spawn` and `join`.

- Notation: write `None` for $\text{inj}_1 ()$ and `Some x` for $\text{inj}_2 x$. 
Encoding of $e_1 \parallel e_2$

\[
\text{spawn} := \lambda f. \text{let } c = \text{ref}(\text{None}) \text{ in } \text{fork } (c \leftarrow \text{Some}(f(\_))) \; ; \; c
\]

\[
\text{join} := \text{rec } f(c) = \text{match } ! c \text{ with}
\]

\[
\text{Some } x \Rightarrow x
\]

\[
\text{None } \Rightarrow f(c)
\]

end

\[
\text{par} := \lambda f_1 f_2. \text{let } h = \text{spawn} f_1 \text{ in}
\]

\[
\text{let } v_2 = f_2(\_) \text{ in}
\]

\[
\text{let } v_1 = \text{join}(h) \text{ in}
\]

\[
(v_1, v_2)
\]

\[
e_1 \parallel e_2 := \text{par}(\lambda \_ . e_1)(\lambda \_ . e_2)
\]
Thread-Local Reasoning

A Key Point of Concurrent Separation Logic:

- We do not reason about possible interleavings of threads (too many to reason about in a scalable way). See Hans Boehm: You Don't Know Jack About Shared Variables or Memory Models. CACM Vol. 55 No. 2, Pages 48-54.
- We reason about each thread in isolation – thread-local reasoning.
- Important for modular reasoning!
Thread-Local Reasoning

A Key Point of Concurrent Separation Logic:

- We do not reason about possible interleavings of threads (too many to reason about in a scalable way). See Hans Boehm: You Don’t Know Jack About Shared Variables or Memory Models. CACM Vol. 55 No. 2, Pages 48-54.

- We reason about each thread in isolation – thread-local reasoning.

- Important for modular reasoning!

- How?
Thread-Local Reasoning

A Key Point of Concurrent Separation Logic:

▶ We do not reason about possible interleavings of threads (too many to reason about in a scalable way). See Hans Boehm: You Don't Know Jack About Shared Variables or Memory Models. CACM Vol. 55 No. 2, Pages 48-54.

▶ We reason about each thread in isolation – thread-local reasoning.

▶ Important for modular reasoning!

▶ How ?

▶ We

▶ either ensure that there are no interesting interleavings among threads (disjoint concurrency),

▶ or we abstract over how threads may interfere with each other, so that it is still possible to reason thread-locally.

▶ Hence Hoare triples over individual expressions continue to be the basic entity of program proofs (rather than some kind of Hoare triple over thread pools).
Disjoint Concurrency Rule

\[
\text{HT-PAR} \\
S \vdash \{P_1\} e_1 \{v.Q_1\} \quad S \vdash \{P_2\} e_2 \{v.Q_2\} \\
S \vdash \{P_1 \ast P_2\} e_1 || e_2 \{v.\exists v_1 v_2. \ v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}
\]

- The rule states that we can run \(e_1\) and \(e_2\) in parallel, if they have *disjoint* footprints and that in this case we can verify the two components separately.
- Thus this rule is sometimes also referred to as the *disjoint concurrency rule*. 
Disjoint Concurrency Example

Let $e_i$ be $\ell_i \leftarrow !\ell_i + 1$, for $i \in \{1, 2\}$. Then we can use $\text{HT-PAR}$ to show:

$$\{\ell_1 \leftarrow n \ast \ell_2 \leftarrow m\} (e_1 || e_2); !\ell_1 + !\ell_2 \{v.v = n + m + 2\}$$
Let $e_i$ be $\ell_i \leftarrow !\ell_i + 1$, for $i \in \{1, 2\}$. Then we can use $\text{HT-PAR}$ to show:

$$\{ \ell_1 \leftrightarrow n \ast \ell_2 \leftrightarrow m \} (e_1 || e_2); !\ell_1 + !\ell_2 \{ v. v = n + m + 2 \}$$

More realistic example: merge sort.
Non-disjoint Concurrency Example

- The $H_T$-PAR rule does not suffice to verify a concurrent program which modifies a shared location.
- For instance, we cannot use it to prove:
  \[
  \{\ell \leftarrow n\} (e \parallel e); !\ell \{v.v \geq n\}
  \]
  where $e$ is the program $\ell \leftarrow !\ell + 1$.
- Why?
Non-disjoint Concurrency Example

The $\text{H}_T\text{-PAR}$ rule does not suffice to verify a concurrent program which modifies a shared location.

For instance, we cannot use it to prove

$$\{\ell \leftarrow n\} (e \parallel e); \! \ell \{\nu.\nu \geq n\}$$

where $e$ is the program $\ell \leftarrow !\ell + 1$.

Why?

- We cannot split the $\ell \leftarrow n$ predicate to give to the two subcomputations.
Non-disjoint Concurrency Example

- The $\text{HT-PAR}$ rule does not suffice to verify a concurrent program which modifies a shared location.
- For instance, we cannot use it to prove
  \[
  \{ l \hookrightarrow n \} (e \parallel e); \! l \{ v. v \geq n \}
  \]
  where $e$ is the program $l \leftarrow ! l + 1$.
- Why?
  - We cannot split the $l \hookrightarrow n$ predicate to give to the two subcomputations.
  - We need the ability to share predicate $l \hookrightarrow n$ among the two threads running in parallel.
- That is what invariants enable.
Non-disjoint Concurrency Example

- The $\text{HT-PAR}$ rule does not suffice to verify a concurrent program which modifies a shared location.
- For instance, we cannot use it to prove

$$\{\ell \leftrightarrow n\} (e \parallel e); !\ell \{v. v \geq n\}$$

where $e$ is the program $\ell \leftarrow !\ell + 1$.
- Why?
  - We cannot split the $\ell \leftrightarrow n$ predicate to give to the two subcomputations.
  - We need the ability to share predicate $\ell \leftrightarrow n$ among the two threads running in parallel.
- That is what \textit{invariants} enable.
- Is this even the best spec we can show?
Non-disjoint Concurrency Example

- The $H_T$-par rule does not suffice to verify a concurrent program which modifies a shared location.
- For instance, we cannot use it to prove

\[ \{\ell \leftarrow n\} (e \parallel e); !\ell \{v. v \geq n\} \]

where $e$ is the program $\ell \leftarrow !\ell + 1$.
- Why?
  - We cannot split the $\ell \leftarrow n$ predicate to give to the two subcomputations.
- We need the ability to *share* predicate $\ell \leftarrow n$ among the two threads running in parallel.
- That is what *invariants* enable.
- Is this even the best spec we can show?
  - The best we can hope to prove is:

\[ \{\ell \leftarrow n\} (e \parallel e); !\ell \{v. v = n + 1 \lor v = n + 2\} \]

but that is considerably harder, so won’t do that for now.
Invariants

- Add a type of invariant names InvName to the logic.
- Add new term $\overline{P}_\iota$, to be read as “invariant $P$ named $\iota$”.
- Typing rule:

\[
\Gamma \vdash P : \text{Prop} \quad \Gamma \vdash \iota : \text{InvName} \quad \Gamma \vdash \overline{P}_\iota : \text{Prop}
\]

- Note that there are no restrictions on $P$. In particular, we are also allowed to form nested invariants, e.g., terms of the form $\overline{P}_{\iota'}$. 
There will be rules allowing us to temporarily *open* invariants, and, conceptually, get local ownership over the resources described by the invariant, so that we may operate on those resources.

Of course, it does not make sense to get local ownership of some resource twice (if we “∗ on” a resource $\ell \leftrightarrow -$ twice, then we get false).

Hence we need to ensure that we do not open invariants more than once.

Hence we index Hoare triples with infinite set of invariant names $\mathcal{E}$:

$$S \vdash \{P\} e \{v.\ Q\}_{\mathcal{E}}$$

This set identifies the invariants we are allowed to use.

If there is no annotation on the Hoare triple then $\mathcal{E} = \text{InvName}$, the set of all invariant names. With this convention all the previous rules are still valid.
Invariant Names on Hoare Triples

- Just one new rule for relating Hoare triples with different sets of invariant names:

\[
\text{HT-MASK-WEAKEN} \\
S \vdash \{P\} e \{v.Q\}_{\mathcal{E}_1} \quad \mathcal{E}_1 \subseteq \mathcal{E}_2 \\
\hline
S \vdash \{P\} e \{v.Q\}_{\mathcal{E}_2}
\]

- Intuitively sound: if we can show the triple while being allowed to open \(\mathcal{E}_1\) invariants, then we can, of course, also show the triple if we are allowed to open more invariants.
A key point of invariants is that they can be shared. Hence invariants are persistent:

$$\text{INV-PERSISTENT}$$

$$P^t \vdash \Box P^t$$

Invariant allocation rule:

$$\text{HT-INV-ALLOC}$$

$$E \text{ infinite} \quad S \land \exists \ell \in E. \ [P]^t \vdash \{Q\} e \{v.R\} E$$

$$S \vdash \{\triangleright P \ast Q\} e \{v.R\} E$$
Rules for Invariants: Invariant Opening Rule

The invariant opening rule

\[ \text{HT-INV-OPEN} \]

\[ S \land [P]^\iota \vdash \{ \triangleright P \ast Q \} e \{ v.\triangleright P \ast R \} \varepsilon \]

\[ S \land [P]^\iota \vdash \{ Q \} e \{ v.R \} \varepsilon \uplus \{ \iota \} \]

is the only way to get access to the resources governed by an invariant.
Rules for Invariants: Invariant Opening Rule

▶ The invariant opening rule

\[ \text{HT-INV-OPEN} \]
\[ \begin{align*} 
S \land [P] & \vdash \{\triangleright P \ast Q\} e \{v.\triangleright P \ast R\} e \\
S \land [P] & \vdash \{Q\} e \{v.R\} e \cup \{i\}
\end{align*} \]

is the only way to get access to the resources governed by an invariant.

▶ Thus if we know an invariant \([P]\) exists, we can temporarily, for one atomic step, get access to the resources.

▶ An expression is atomic if it reduces to a value in one reduction step.
Rules for Invariants: Invariant Opening Rule

- The invariant opening rule

\[
\text{HT-INV-OPEN} \\
S \land [P]^\ell \vdash \{\triangleright P \ast Q\} e \{v.\triangleright P \ast R\} \varepsilon \\
S \land [P]^\ell \vdash \{Q\} e \{v.R\} \varepsilon \cup \{\ell\}
\]

is the only way to get access to the resources governed by an invariant.

- Thus if we know an invariant $[P]^\ell$ exists, we can \textit{temporarily}, for one atomic step, get access to the resources.
  - An expression is \textit{atomic} if it reduces to a value in \textit{one} reduction step.

- Note: we only get access to the resources \textit{later} ($\triangleright$).
  - This is essential, logic would be inconsistent otherwise (proof not covered in this course)
The invariant opening rule

\[
\text{HT-INV-OPEN}
\]

\[
S \land [P] \vdash \{ \triangleright P \ast Q \} \ e \ \{ \triangleright v. P \ast R \} \ E
\]

\[
S \land [P] \vdash \{ Q \} \ e \ \{ v. R \} \ E \cup \{ \iota \}
\]

is the only way to get access to the resources governed by an invariant.

Thus if we know an invariant \([P] \) exists, we can temporarily, for one atomic step, get access to the resources.

- An expression is atomic if it reduces to a value in one reduction step.

Note: we only get access to the resources later (\(\triangleright\)).

- This is essential, logic would be inconsistent otherwise (proof not covered in this course)

This rule is the reason we need to annotate the Hoare triples with sets of invariant names \(E\).
Stronger Frame Rule

- Stronger frame rule which allows to remove $\triangleright$ from frame:

\[
\begin{align*}
\text{HT-FRAME-ATOMIC} \\
\text{e is an atomic expression} & \quad S \vdash \{P\} e \{v.Q\} \\
S \vdash \{P \triangleright R\} e \{v.Q \ast R\}
\end{align*}
\]

- (We will see an example application of this rule later.)
Remark: Footprint Reading of Hoare Triples

- Earlier “minimal footprint” reading must be refined now.
- Given triple $\{P\} e \{v.Q\}$, the resources required for running $e$ can
  - either be in the precondition $P$,
  - or be governed by one or more invariants.
- For example, may prove triples of the form $\{\text{True}\} e \{v.Q\}$, for some $Q$, where $e$ accesses shared state governed by an invariant.
Recall the example we cannot prove with disjoint concurrency rule:

\[ \{ \ell \gets n \} (e || e); ! \ell \{ v. v \geq n \} \]

where \( e \) is the program \( \ell \gets ! \ell + 1 \).

Let’s prove it now!

We start by allocating invariant

\[ l = \exists m. m \geq n \land \ell \gets m \]

using \texttt{HT-INV-ALLOC} rule. This is possible by rule of consequence, since \( \ell \gets n \) implies \( l \) and hence \( \triangleright l \).
Example proof

Thus we have to prove

\[ I^i \vdash \{ \text{True} \} (e \parallel e); ! \ell \{ v.v \geq n \} \]  \hspace{1cm} (1)

for some \( i \).

Using the derived sequencing rule \( \text{HT-SEQ} \) SFTS the following two triples

\[ I^i \vdash \{ \text{True} \} (e \parallel e) \{ \_ . \text{True} \}. \]

\[ I^i \vdash \{ \text{True} \} ! \ell \{ v.v \geq n + 1 \}. \]

We show the first one; during the proof of that we will need to show the second triple as well.

Using \( \text{HT-PAR} \), SFTS

\[ I^i \vdash \{ \text{True} \} e \{ \_ . \text{True} \} \]

(Note that we cannot open the invariant now since the expression \( e \) is not atomic.)
Example proof

- Using the bind rule we first show

$$I^\ell \vdash \{\text{True}\} ! \ell \{v.v \geq n\}.$$ 

- Note that this is exactly the second premise of the sequencing rule mentioned above.

- By invariant opening rule $\text{HT-INV-OPEN} \ SFTS$

$$\{\triangleright I\} ! \ell \{v.v \geq n \land \triangleright I\}_{\text{InvName}\{\iota\}}.$$ 

- Using rule $\text{HT-FRAME-ATOMIC}$ together with $\text{HT-LOAD}$ and structural rules we have

$$\{\triangleright I\} ! \ell \{v.v = m \land m \geq n \land \ell \mapsto m\}_{\text{InvName}\{\iota\}}.$$ 

From this we easily derive the needed triple.
Example proof

- To show the second premise of the bind rule, SFTS

\[ \{ \} \vdash \forall m. \{ m \geq n \} \ell \leftarrow (m + 1) \{ \text{True} \} \].

- To show this we again use the invariant opening rule and HT-FRAME-ATOMIC (exercise!).