Iris: Higher-Order Concurrent Separation Logic

Lecture 9: Concurrency Intro and Invariants

Lars Birkedal

Aarhus University, Denmark

November 19, 2020

Overview

Earlier:

- Operational Semantics of $\lambda_{\mathrm{ref,conc}}$
  - e, $(h, e) \rightsquigarrow (h, e')$, and $(h, \mathcal{E}) \to (h', \mathcal{E}')$
- Basic Logic of Resources
  - $l \hookrightarrow v$, $P * Q$, $P \mathrel{-\!\!*} Q$, $\Gamma \mid P \vdash Q$
- Basic Separation Logic
  - $\{P\}\, e\, \{v.Q\}$: Prop, isList l xs, ADTs, foldr
- Later ($\triangleright$) and Persistent ($\square$) Modalities.

Today:

- Concurrency Intro: $e_1 \mid\mid e_2$
- Invariants: $\boxed{P}^{\iota}$
- Key Points:
  - Thread-local reasoning.
  - Disjoint concurrency rule for $e_1 \mid\mid e_2$
  - Invariants for sharing of resources among threads e_1 and e_2 in $e_1 \mid\mid e_2$.

Parallel Composition

To start off with simpler proof rules, we first define a programming language construct for parallel execution of two expresssions e_1 and e_2.

- $e_1 \mid\mid e_2$ 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 $\mathsf{inj}_1\,()$ and Some x for $\mathsf{inj}_2\,x$.

Encoding of $e_1 \mid\mid e_2$

$$\mathsf{spawn} := \lambda f.\mathsf{let}\, c = \mathsf{ref}(\mathsf{None})\,\mathsf{in}\,\mathsf{fork}\,(c \leftarrow \mathsf{Some}(f\,()))\,;\, c$$

$$\begin{aligned}\mathsf{join} := \mathsf{rec}\, f(c) = \ & \mathsf{match}\ !\,c\,\mathsf{with} \\ & \quad \mathsf{Some}\, x \Rightarrow x \\ & \mid \mathsf{None} \quad \Rightarrow f(c) \\ & \mathsf{end}\end{aligned}$$

$$\begin{aligned}\mathsf{par} := \lambda f_1 f_2.\, & \mathsf{let}\, h = \mathsf{spawn}\, f_1\,\mathsf{in} \\ & \mathsf{let}\, v_2 = f_2\,()\,\mathsf{in} \\ & \mathsf{let}\, v_1 = \mathsf{join}(h)\,\mathsf{in} \\ & (v_1, v_2)\end{aligned}$$

$$e_1 \mid\mid e_2 := \mathsf{par}(\lambda\_.e_1)(\lambda\_.e_2)$$

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

$$\textsc{Ht-par}\quad \frac{S \vdash \{P_1\}\, e_1\, \{v.\, Q_1\} \qquad S \vdash \{P_2\}\, e_2\, \{v.\, Q_2\}}{S \vdash \{P_1 * P_2\}\, e_1 \mid\mid e_2\, \{v.\exists v_1 v_2.\, v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}}$$

- 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 $\textsc{Ht-par}$ to show:

$$\{\ell_1 \hookrightarrow n * \ell_2 \hookrightarrow m\}\,(e_1 \mid\mid e_2); {!}\,\ell_1 + {!}\,\ell_2\,\{v.\, v = n + m + 2\}$$

Disjoint Concurrency Example

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

$$\{\ell_1 \hookrightarrow n * \ell_2 \hookrightarrow m\}\,(e_1 \mid\mid e_2);\, !\,\ell_1 + !\,\ell_2\,\{v.v = n + m + 2\}$$

- More realistic example: merge sort.

Non-disjoint Concurrency Example

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

  $$\{\ell \hookrightarrow n\}\,(e \mid\mid e);\,!\,\ell\,\{v.v \geq n\}$$

  where e is the program $\ell \leftarrow !\,\ell + 1$.
- Why?

Non-disjoint Concurrency Example

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

  $$\{\ell \hookrightarrow n\}\,(e \,||\, e);\,!\,\ell\,\{v.v \geq n\}$$

  where e is the program $\ell \leftarrow !\,\ell + 1$.
- Why?
  - We cannot split the $\ell \hookrightarrow n$ predicate to give to the two subcomputations.

Non-disjoint Concurrency Example

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

$$\{\ell \hookrightarrow n\}\,(e \,||\, e);!\,\ell\,\{v.v \geq n\}$$

  where e is the program $\ell \leftarrow !\,\ell + 1$.
- Why?
  - We cannot split the $\ell \hookrightarrow n$ predicate to give to the two subcomputations.
- We need the ability to *share* predicate $\ell \hookrightarrow n$ among the two threads running in parallel.
- That is what *invariants* enable.

Non-disjoint Concurrency Example

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

$$\{\ell \hookrightarrow n\}\,(e \,||\, e);\,!\,\ell\,\{v.v \geq n\}$$

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

Non-disjoint Concurrency Example

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

  $$\{\ell \hookrightarrow n\}\,(e \mid\mid e);\,!\,\ell\,\{v.v \geq n\}$$

  where e is the program $\ell \leftarrow !\,\ell + 1$.
- Why?
  - We cannot split the $\ell \hookrightarrow n$ predicate to give to the two subcomputations.
- We need the ability to *share* predicate $\ell \hookrightarrow n$ 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:

    $$\{\ell \hookrightarrow n\}\,(e \mid\mid e);\,!\,\ell\,\{v.v = n + 1 \vee v = n + 2\}$$

    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 $\boxed{P}^{\iota}$, to be read as "invariant P *named* ι".
- Typing rule:

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash \iota : \mathsf{InvName}}{\Gamma \vdash \boxed{P}^{\iota} : \mathsf{Prop}}$$

- Note that there are *no restrictions on* P. In particular, we are also allowed to form *nested invariants*, *e.g.*, terms of the form $\boxed{\boxed{P}^{\iota}}^{\iota'}$.

Intuition of memory

- $\ell_1 \leftarrow {!}\,\ell_1 + {!}\,\ell \mid\mid \ell_2 \leftarrow {!}\,\ell_2 + {!}\,\ell$.
- ℓ_1 owned by the first expression, ℓ_2 by the second, ℓ shared.

Invariant Names on Hoare Triples

- 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 $\ell \hookrightarrow -$ 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 $\mathcal{E} = \mathsf{InvName}$, 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:

$$\frac{\begin{array}{c}\textsc{Ht-mask-weaken}\\ S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}_1} \qquad \mathcal{E}_1 \subseteq \mathcal{E}_2\end{array}}{S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}_2}}$$

- 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.

Rules for Invariants: Persistence and Allocation

- A key point of invariants is that they can be shared. Hence invariants are persistent:

$$\textsc{Inv-persistent}\quad \frac{}{\boxed{P}^{\iota} \vdash \Box \boxed{P}^{\iota}}$$

- Invariant allocation rule:

$$\textsc{Ht-inv-alloc}\quad \frac{\mathcal{E} \text{ infinite} \qquad S \wedge \exists \iota \in \mathcal{E}. \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E}}}{S \vdash \{\triangleright P * Q\}\, e\, \{v.R\}_{\mathcal{E}}}$$

Rules for Invariants: Invariant Opening Rule

- The invariant opening rule

$$\frac{\begin{array}{cc}\textsc{Ht-inv-open} & \\ e \text{ is an atomic expression} & S \wedge \boxed{P}^{\iota} \vdash \{\triangleright P * Q\}\, e\, \{v.\, \triangleright P * R\}_{\mathcal{E}}\end{array}}{S \wedge \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E} \uplus \{\iota\}}}$$

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

Rules for Invariants: Invariant Opening Rule

- The invariant opening rule

$$\textsc{Ht-inv-open}\quad \frac{e \text{ is an atomic expression} \qquad S \wedge \boxed{P}^{\iota} \vdash \{\triangleright P * Q\}\, e\, \{v.\, \triangleright P * R\}_{\mathcal{E}}}{S \wedge \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E} \uplus \{\iota\}}}$$

  is the only way to get access to the resources governed by an invariant.
- Thus if we know an invariant $\boxed{P}^{\iota}$ 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

$$\textsc{Ht-inv-open}\;\frac{e \text{ is an atomic expression} \qquad S \wedge \boxed{P}^{\iota} \vdash \{\triangleright P * Q\}\, e\, \{v.\triangleright P * R\}_{\mathcal{E}}}{S \wedge \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E} \uplus \{\iota\}}}$$

  is the only way to get access to the resources governed by an invariant.
- Thus if we know an invariant $\boxed{P}^{\iota}$ 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.
- This rule is the reason we need to annotate the Hoare triples with sets of invariant names $\mathcal{E}$.

Regarding $\triangleright$ in the Invariant Opening Rule

- Note: we only get access to the resources *later* ($\triangleright$).
- This is essential, logic would be inconsistent otherwise; proof not covered in this course, see `https://iris-project.org/pdfs/2016-icfp-iris2-final.pdf`
- There is a wide class of *timeless* propositions for which it does not matter.
- Timeless propositions include most ordinary propositions, but not those involving a later modality, an update modality, or a general predicate variable.
- Many concrete examples will thus not need the general rule with later above.
- We therefore did consider leaving it out of this course.
- But it is a key feature of Iris that invariants can contain general predicates (not just timeless ones), in particular predicate variables.
- This is important for giving modular specs, see, e.g., the specification for a lock next week.
- And for other advanced applications: models of type systems.

Stronger Frame Rule

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

$$\textsc{Ht-frame-atomic}\quad \frac{e \text{ is an atomic expression} \qquad S \vdash \{P\}\, e\, \{v.\, Q\}}{S \vdash \{P * \triangleright R\}\, e\, \{v.\, Q * R\}}$$

- (We will see an example application of this rule later.)

Remark: Footprint Reading of Hoare Triples

- Earlier "minimal footpring" 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.

Example

- Recall the example we cannot prove with disjoint concurrency rule:

$$\{\ell \hookrightarrow n\}\,(e \mid\mid e); !\,\ell\,\{v.v \geq n\}$$

  where e is the program $\ell \leftarrow !\,\ell + 1$.
- Let's prove it now!
- We start by allocating invariant

$$I = \exists m.\, m \geq n \wedge \ell \hookrightarrow m$$

  using $\textsc{Ht-inv-alloc}$ rule. This is possible by rule of consequence, since $\ell \hookrightarrow n$ implies I and hence $\triangleright I$.

Example proof

- Thus we have to prove

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,(e \mid\mid e);\,!\,\ell\,\{v.v \geq n\} \tag{1}$$

  for some ι.
- Using the derived sequencing rule HT-SEQ SFTS the following two triples

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,(e \mid\mid e)\,\{\_.\mathsf{True}\}.$$
$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\;!\,\ell\,\{v.v \geq n\}.$$

- We show the first one; during the proof of that we will need to show the second triple as well.
- Using HT-PAR, SFTS

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,e\,\{\_.\mathsf{True}\}$$

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

Example proof

- Using the bind rule we first show

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\ !\,\ell\,\{v.v \geq n\}.$$

- Note that this is exactly the second premise of the sequencing rule mentioned above.
- By invariant opening rule Ht-inv-open SFTS

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

- Using rule Ht-frame-atomic together with Ht-load and structural rules we have

$$\{\triangleright I\}\ !\,\ell\,\{v.v = m \land m \geq n \land \ell \hookrightarrow m\}_{\mathsf{InvName}\setminus\{\iota\}}.$$

From this we easily derive the needed triple.

Example proof

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

$$\boxed{I}^{\iota} \vdash \forall m.\, \{m \geq n\}\, \ell \leftarrow (m+1)\, \{\_.\,\mathsf{True}\}.$$

- To show this we again use the invariant opening rule and $\textsc{Ht-frame-atomic}$ (exercise!).