Iris: Higher-Order Concurrent Separation Logic

Basic Separation Logic: Proving Pointer Programs[1]

Amin Timany

Aarhus University, Denmark

August 13, 2026

[1] Based on slides by Lars Birkedal

Derived rule: Sequencing

- Sequencing $e1; e_2$ is also definable (using let).
- Derivable rule

$$\textsc{Ht-seq}\quad \frac{S \vdash \{P\}\, e_1\, \{v.Q\} \qquad S \vdash \{\exists x.\, Q\}\, e_2\, \{u.R\}}{S \vdash \{P\}\, e_1; e_2\, \{u.R\}}$$

$$\frac{S \vdash \{P\}\, e_1\, \{\_.Q\} \qquad S \vdash \{Q\}\, e_2\, \{u.R\}}{S \vdash \{P\}\, e_1; e_2\, \{u.R\}}$$

where $\_.Q$ means that Q does not mention the return value.

Example: swap (finally!)

Implementation

$$\mathsf{swap} = \lambda x\ y.\mathsf{let}\ z := !\,x\ \mathsf{in}\ x \leftarrow !\,y; y \leftarrow z$$

Possible Specification

$$\{\ell_1 \hookrightarrow v_1 * \ell_2 \hookrightarrow v_2\}\ \mathsf{swap}\ \ell_1\ \ell_2\ \{v.v = () \wedge \ell_1 \hookrightarrow v_2 * \ell_2 \hookrightarrow v_1\}.$$

Proof Outlines

In the literature, you will often see *proof outlines*, which sketch how proofs of programs are done. Tricky to make precise, so not used in the Iris Lecture Notes. Possible proof outline for above proof for swap:

$$
\begin{array}{l}
\{\ell_1 \hookrightarrow v_1 \ * \ell_2 \hookrightarrow v_2\} \\
\mathsf{swap}\, \ell_1 \ell_2 \\
\{\ell_1 \hookrightarrow v_1 \ * \ell_2 \hookrightarrow v_2\} \\
\mathsf{let}\, z := !\, \ell_1 \,\mathsf{in} \\
\{\ell_1 \hookrightarrow v_1 \ * \ell_2 \hookrightarrow v_2 * z = v_1\} \\
\ell_1 \leftarrow !\, \ell_2; \\
\{\ell_1 \hookrightarrow v_2 \ * \ell_2 \hookrightarrow v_2 * z = v_1\} \\
\ell_2 \leftarrow z \\
\{\ell_1 \hookrightarrow v_2 \ * \ell_2 \hookrightarrow v_1 * z = v_1\} \\
\{\ell_1 \hookrightarrow v_2 \ * \ell_2 \hookrightarrow v_1\}
\end{array}
$$

Mutable Shared Data Structures: Linked Lists

- Key Point: We reason about mutable shared data structures by relating them to mathematical models.
- Today:
  - Mutable Shared Data Structure = linked lists
  - Mathematical Model = sequences (aka functional lists)
  - $\mathsf{isList}\ l\ xs$ relates value l to sequence xs
  - Defined by induction on xs

$$\mathsf{isList}\ l\ [] \equiv l = \mathsf{inj}_1()$$
$$\mathsf{isList}\ l\ (x : xs) \equiv \exists hd, l'.\ l = \mathsf{inj}_2(hd) * hd \hookrightarrow (x, l') * \mathsf{isList}\ l'\ xs$$

  - Why do we use $*$ above?

Example: inc on linked lists

Incrementing all values in a linked list of integers.
Implementation

$$
\begin{array}{l}
\mathsf{rec\,inc}\ l := \mathsf{match}\ l\ \mathsf{with} \\
\quad \mathsf{inj}_1\ x_1 \Rightarrow () \\
\quad \mid \mathsf{inj}_2\ x_2 \Rightarrow \mathsf{let}\ h := \pi_1\ !\,x_2\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{let}\ t := \pi_2\ !\,x_2\ \mathsf{in} \\
\qquad\qquad\qquad\quad x_2 \leftarrow (h+1, t); \\
\qquad\qquad\qquad\quad \mathsf{inc}\ t \\
\quad \mathsf{end}
\end{array}
$$

Specification

$$\forall xs.\forall l.\{\mathsf{isList}\ l\ xs\}\ \mathsf{inc}\ l\ \{v.v = () \wedge \mathsf{isList}\ l\ (\mathsf{map}(1+)xs)\}$$

Example: inc on linked lists

Proof

- Proceed by the recursion rule.
- When considering the body of inc, proceed by case analysis of xs.
- Case $xs = []$: TS
  - $\forall l.\{\text{isList } l[]\}$ match l with ... $\{v.v = () \wedge \text{isList } l(\text{map}(1+)[])\}$
- Case $xs = x : xs'$: TS
  - $\forall x, xs'.\forall l.\{\text{isList } l(x : xs')\}$ match l with... $\{v.v = () \wedge \text{isList } l(\text{map}(1+)(x : xs'))\}$
- In both cases, we proceed by the match rule (the isList predicate will tell us which branch is chosen).

Case $xs = []$

To show

- $\forall l.\{\mathsf{isList}\ l\ []\}\ \mathsf{match}\ l\ \mathsf{with}\ \ldots\ \{v.v = () \land \mathsf{isList}\ l\ (\mathsf{map}(1+)[])\}$

Note: $\mathsf{isList}\ l\ [] \equiv l = \mathsf{inj}_1()$. Thus SFTS

$$\{\mathsf{isList}\ l\ []\}\ l\ \{v.v = \mathsf{inj}_1\ () * \mathsf{isList}\ l\ []\}$$

which we do by HT-FRAME and HT-PRE-EQ followed by HT-RET.

Case $xs = x : xs'$

- To show
  - $\forall x, xs'. \forall l. \{\text{isList } l (x : xs')\}$ match l with... $\{v. v = () \wedge \text{isList } l (\text{map}(1+)(x : xs'))\}$
- Note: $\text{isList } l (x : xs) \equiv \exists hd, l'. l = \text{inj}_2\ hd * hd \hookrightarrow (x, l') * \text{isList } l' xs'$. Thus we have

$$\{l = \text{inj}_2\ hd * hd \hookrightarrow (x, l') * \text{isList } l' xs'\}$$
$$l$$
$$\{r. r = \text{inj}_2\ hd * l = \text{inj}_2\ hd * hd \hookrightarrow (x, l') * \text{isList } l' xs'\}$$

  for some l' and hd, using the rule HT-EXIST, the frame rule, the HT-PRE-EQ rule, and the HT-RET rule.
- Hence, the second branch will be taken and by the match rule it suffices to verify the body of the second branch.

Case $xs = x : xs'$, continued

- Using HT-LET-DET and HT-PROJ repeatedly we quickly prove

$$\{l = \mathsf{inj}_2\ hd * hd \hookrightarrow (x, l') * \mathsf{isList}\ l'\ xs'\}$$
$$\mathsf{let}\ h := \pi_1\ !hd\ \mathsf{in}$$
$$\mathsf{let}\ t := \pi_2\ !hd\ \mathsf{in}$$
$$hd \leftarrow (h + 1, t)$$
$$\{l = \mathsf{inj}_2\ hd * hd \hookrightarrow (x + 1, l') * \mathsf{isList}\ l'\ xs' * t = l' * h = x\}$$

- Now, by HT-SEQ and HT-CSQ, we are left with proving

$$\{l = \mathsf{inj}_2\ hd * hd \hookrightarrow (x + 1, t) * \mathsf{isList}\ t\ xs'\}$$
$$f\ t$$
$$\{r.r = () * \mathsf{isList}\ l(\mathsf{map}(+1)(x : xs'))\}$$

which follows from assumption (cf. recursion rule) and definition of the isList predicate.

Iris: Higher-Order Concurrent Separation Logic

Abstract Data Types[2]

Amin Timany

Aarhus University, Denmark

August 13, 2026

[2]Based on slides by Lars Birkedal

Counter Module, v1

Implementation

$$\begin{aligned} \text{mk\_counter} &= \lambda\_.\text{ref}\,(0) \\ \text{inc\_counter} &= \lambda x.x \leftarrow !\,x + 1 \\ \text{read\_counter} &= \lambda x.!\,x \end{aligned}$$

Specification

$$\begin{aligned} &\{\text{True}\}\ \text{mk\_counter}()\,\{v.v \hookrightarrow 0\} \\ &\{\ell \hookrightarrow n\}\ \text{inc\_counter}\,\ell\,\{v.v = () \wedge \ell \hookrightarrow n + 1\} \\ &\{\ell \hookrightarrow n\}\ \text{read\_counter}\,\ell\,\{v.v = n \wedge \ell \hookrightarrow n\}. \end{aligned}$$

Problems with v1

- Exposes internals of the counter, so not modular:
- If we verify a client relative to this spec and then change the data representation, then we will have to re-verify the client

Idea

- Existentially quantify over the "counter representation predicate" C
- thus hiding the fact that the return value is a location.

Counter Module, v2

Implementation

$$\begin{aligned}\text{mk\_counter} &= \lambda\_.\text{ref}\,(0)\\ \text{inc\_counter} &= \lambda x.x \leftarrow\, !\,x + 1\\ \text{read\_counter} &= !\,x\end{aligned}$$

Specification

$$\begin{aligned}&\exists C : \mathit{Val} \to \mathbb{N} \to \text{Prop}.\\ &\{\text{True}\}\ \text{mk\_counter}()\,\{c.C(c,0)\}\wedge\\ &\forall c.\,\{C(c,n)\}\ \text{inc\_counter}\,c\,\{v.v = () \wedge C(c,n+1)\}\wedge\\ &\forall c.\,\{C(c,n)\}\ \text{read\_counter}\,c\,\{v.v = n \wedge C(c,n)\}.\end{aligned}$$

Problems with v2

- Implementation code does not provide any abstraction, so:
- Client of the code may directly modify the reference cell representing the counter

Idea

- Data abstraction in typed language: by using some form of module types / existential types
- In our untyped language: use local state encapsulation

Counter Module, v3

Implementation (only exposes the increment and read methods):

$$\text{counter} = \lambda\_.\text{let } x := \text{ref}\,(0) \text{ in } (\lambda\_.x \leftarrow !\,x + 1,\ \ \lambda\_.\,!\,x)$$

Specification

$$\{\text{True}\}\ \text{counter}()\left\{ v.\exists \ell.\ \begin{array}{l} \ell \hookrightarrow 0 * \\ \forall n.\,\{\ell \hookrightarrow n\}\ v.\,\text{inc}()\,\{u.u = () \wedge \ell \hookrightarrow (n+1)\} * \\ \forall n.\,\{\ell \hookrightarrow n\}\ v.\,\text{read}()\,\{u.u = n \wedge \ell \hookrightarrow n\} \end{array} \right\}$$

where we write $v.\,\text{inc}$ in place of $\pi_1\ v$ and $v.\,\text{read}$ in place of π_2

Problems with v3

- Exposes that the internal state is a single location.

Idea

- Combine with existential quantification of representation predicate.

Counter Module, v4

$$\{\mathsf{True}\}\ \mathsf{counter}()\left\{v.\exists C : \mathbb{N} \to \mathsf{Prop}.\ \begin{array}{l} C(0)* \\ \forall n.\, \{C(n)\}\ v.\, \mathsf{inc}()\, \{u.u = () \wedge C(n+1)\}* \\ \forall n.\, \{C(n)\}\ v.\, \mathsf{read}()\, \{u.u = n \wedge C(n)\} \end{array}\right\}$$

Issues with v4

Pros

- Modular!

Cons

- Code and Spec a bit more convoluted (to be expected that there is a price to pay for modularity).
- A bit cumbersome to work with in Coq

Conclusion

- We will typically use v2 implementation and specification style, and sacrifice abstraction at the programming level (which is arguably ok, since we verify programs relative to their specification, not their implementation).

Exercise (jointly, on the board)

- Give a specification for a stack, with create, push, and pop methods.

Possible client

$$
\begin{aligned}
&\textsf{let } s := \mathsf{mk\_stack}() \textsf{ in} \\
&\qquad \mathsf{push}(1, s); \\
&\qquad \mathsf{push}(2, s); \\
&\qquad \textsf{let } y := \mathsf{pop}(s) \textsf{ in } y
\end{aligned}
$$

Stack Specification, v0

Inspiration from the counter spec:

$$\exists \mathrm{isStack} : \mathit{Val} \to \mathrm{list}\,\mathit{Val} \to \mathsf{Prop}.$$
$$\{\mathsf{True}\}\ \mathsf{mk\_stack}()\ \{s.\mathrm{isStack}(s, [])\} \wedge$$
$$\forall s. \forall xs. \{\mathrm{isStack}(s, xs)\}\ \mathsf{push}(x, s)\ \{v. v = () \wedge \mathrm{isStack}(s, x :: xs)\} \wedge$$
$$\forall s. \forall x, xs. \{\mathrm{isStack}(s, x :: xs)\}\ \mathsf{pop}(s)\ \{v. v = x \wedge \mathrm{isStack}(s, xs)\}$$

Problems with Stack Specification v0

- Suffices to show that the client above returns 2.
- But can we also use it for pushing other data, not just numbers ?

Another possible client

$$
\begin{array}{l}
\textsf{let } s := \mathsf{mk\_stack}() \textsf{ in} \\
\textsf{let } x1 := \textsf{ref } (1) \textsf{ in} \\
\textsf{let } x2 := \textsf{ref } (2) \textsf{ in} \\
\mathsf{push}(x1, s); \\
\mathsf{push}(x2, s); \\
\textsf{let } y := \mathsf{pop}(s) \textsf{ in} \\
y \leftarrow 3
\end{array}
$$

A More General Spec ?

Desiderata:

- A spec that works for both clients (and other clients) — think “generics”.
- A spec that allows us to transfer ownership of mutable data when we push and pop elements to / from the stack.
- Transfer ownership from client to module when pushing an element, transfer ownership from module to client when popping an element.
- To ensure that client cannot accidentally mutate data that has been pushed to the stack

Stack Specification

$$\exists \mathrm{isStack} : \mathit{Val} \to \mathrm{list}\,\mathit{Val} \to (\mathit{Val} \to \mathrm{Prop}) \to \mathrm{Prop}.$$
$$\forall P : \mathit{Val} \to \mathrm{Prop}.$$
$$\{\mathrm{True}\}\ \mathrm{mk\_stack}()\ \{s.\mathrm{isStack}(s, [], P)\} \wedge$$
$$\forall s. \forall xs. \{\mathrm{isStack}(s, xs, P) * P(x)\}\ \mathrm{push}(x, s)\ \{v. v = () \wedge \mathrm{isStack}(s, x : xs, P)\} \wedge$$
$$\forall s. \forall x, xs. \{\mathrm{isStack}(s, x : xs, P)\}\ \mathrm{pop}(s)\ \{v. v = x \wedge \mathrm{isStack}(s, xs, P) * P(x)\}$$

Notes:

- Universal quantification over P to capture generic resource to be transferred back and forth.
- Think about how to instantiate P for each of the two clients before.
- Now $\mathrm{isStack}$ is a higher-order predicate! (Recall that Iris is a higher-order logic).