Iris: Higher-Order Concurrent Separation Logic

Lecture 5: Abstract Data Types

Lars Birkedal

Aarhus University, Denmark

November 10, 2017

Overview

Earlier:

- Operational Semantics of $\lambda_{\mathrm{ref,conc}}$
  - e, $(h, e) \rightsquigarrow (h, e')$, and $(h, \mathcal{E}) \rightarrow (h', \mathcal{E}')$
- 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

Today:

- Abstract Data Types
- Key Points:
  - Existential Quantification over Representation Predicate
  - Discussion of Different Styles of Programs and Specs

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):

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

Specification

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

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

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

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

- In this course, 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

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

Stack Specification, v0

Inspiration from the counter spec:

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

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

```
let $s$ = mk_stack() in
let $x1$ = ref(1) in
let $x2$ = ref(2) in
    push($x1$, $s$);
    push($x2$, $s$);
    let $y$ = pop($s$) in
    $y \leftarrow 3$
```

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

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

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).