Iris: Higher-Order Concurrent Separation Logic

Lecture 6: Case Study: foldr

Lars Birkedal

Aarhus University, Denmark

September 27, 2020
Overview

Earlier:
- Operational Semantics of $\lambda_{\text{ref, conc}}$
  - $e, (h, e) \leadsto (h, e')$, and $(h, E) \rightarrow (h', E')$
- Basic Logic of Resources
  - $I \leftrightarrow v, P \preceq Q, P \npreceq Q, \Gamma \mid P \vdash Q$
- Basic Separation Logic
  - $\{P\} e \{v.Q\} : \text{Prop, isList} I \, xs$
- Abstract Data Types

Today:
- Case Study: foldr
- Key Points:
  - Nested triples for specification of higher-order functions.
  - Use a mathematical model of the data structure and prove most properties on that.
  - Test spec with several clients.
Recall the isList predicate, defined by induction on the mathematical sequence \(xs\).

\[
isList \ l[] \equiv \ l = \text{inj}_1()
\]

\[
isList \ l(x : xs) \equiv \exists \hd, \ l'. \ l = \text{inj}_2(\hd) \ast \hd \leftrightarrow (x, \ l') \ast \text{isList} \ l' \ xs
\]
foldr

- Intuitive type:

\[
\text{foldr} : (\alpha \times \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha \ \text{list} \rightarrow \beta
\]

\[
\text{rec foldr}(f \ a \ l) = \text{match } l \ \text{with}
\]

\[
\text{inj}_1 \ x_1 \Rightarrow a
\]

\[
| \ \text{inj}_2 \ x_2 \Rightarrow \text{let } h = \pi_1 ! x_2 \text{ in}
\]

\[
\text{let } t = \pi_2 ! x_2 \text{ in}
\]

\[
f (h, (\text{foldr } f \ a \ t))
\]

end
Specification of foldr

∀P, Inv. ∀f. ∀xs. ∀l. \{ \forall x. \forall a'. \forall ys. \{ P x \ast Inv ys a' \} f (x, a') \{ r. Inv(x : ys) r \} \}

∀P, Inv. ∀f. ∀xs. ∀l. \{ \ast isList l xs \ast allP xs \ast Inv [] a \}

foldr f a l

\{ r. isList l xs \ast Inv xs r \} 

where

allP [] \equiv True

allP (x : xs) \equiv P x \ast allP xs
Remarks about the Specification

- The $\lambda_{\text{ref,conc}}$ value $l$ is related to a mathematical sequence $xs$, which is our model of lists.

- The rest of the spec is formulated in terms of the model, e.g., the invariant $Inv$ has type $Inv : \text{list Val} \rightarrow \text{Val} \rightarrow \text{Prop}$, where $\text{list Val}$ is the type of mathematical sequence of values.

  - Idea: allows most of the reasoning to be done at the math model level, without considering the imperative code.

  - See Esben Clausen's Hash Table Specification (on iris-project.org) for another example.

- We use a nested triple because $\text{foldr}$ is a higher-order function.

- We quantify over $P$ and $Inv$ to allow clients to instantiate those. The idea is that $P$ is a predicate that holds for each element in the given list, and $Inv \ xs \ a$ expresses that $a$ is the result of folding $f$ over $xs$. 
Client: sumList

\[
\text{rec sumList}(l) = \text{let } f = \lambda(x, y).x + y \text{ in foldr } f \ 0 \ l
\]

\[
\forall l. \forall xs. \{\text{isList } l \ xs * \text{allNats } xs\} \ \text{sumList } l \{r. \text{isList } l \ xs * r = \Sigma_{x \in xs} x\}
\]

where

\[
\text{allNats } [] \equiv \text{True}
\]

\[
\text{allNats } (x : xs) \equiv \text{isNat } x * \text{allNats } xs
\]

\[
\text{isNat } x \equiv \begin{cases} 
\text{True} & \text{if } x \in \mathbb{N} \\
\text{False} & \text{otherwise}
\end{cases}
\]
Proof of sumList

Let \( l \) and \( xs \) be arbitrary. Instantiate spec for \( \text{foldr} \) with

\[
\begin{align*}
\text{P} &= \text{isNat} \\
\text{Inv } ys a' &= (a' = \mathbb{N} \sum_{y \in ys} y) \\
f &= \lambda(x, y). x + y \quad \text{and} \quad l = l \quad \text{and} \quad xs = xs
\end{align*}
\]

to get

\[
\left\{ \begin{aligned}
(\forall x, a. \forall ys. \{ \text{isNat } x \ast a = \sum_{y \in ys} y \} (\lambda(x, y). x + y)(x, a) \{ r. r = \sum_{y \in (x:ys)} y \}) \\
* \text{isList } l xs * \text{allNats } xs * 0 = \sum_{x \in \emptyset} x
\end{aligned} \right.
\]

\[
\text{foldr} (\lambda(x, y). x + y) a l
\]

\[
\{ r. \text{isList } l xs * r = \sum_{x \in xs} x \}
\]

which is almost what we want, the difference being the precondition.
Proof of sumList

By rule of consequence SFTS

\[ \text{isList } l \; xs \ast \text{allNats } xs \]

\[ \Rightarrow \]

\[ (\forall x, a. \; \forall ys. \{ \text{isNat } x \ast a = \sum_{y \in ys} y \} \; (\lambda (x, y). x + y)(x, a) \{ r. r = \sum_{y \in (x:ys)} \}) \ast \text{isList } l \; xs \ast \text{allNats } xs \ast 0 = \sum_{x \in \emptyset} x \]

which is left as exercise.
Client: filter

\[
\text{rec } \text{filter}(p \ l) = \ \text{let } f = (\lambda(x, xs). \ \text{if } p \ x \\
\quad \text{then } \text{inj}_2(\text{ref}(x, xs)) \\
\quad \text{else } xs) \\
\quad \text{in} \\
\text{foldr } f \ [\ ] \ l
\]
Specification of filter

\[
((\forall x. \{ true \} p x \{ v. isBool \ v \star v = P x \}) \star isList \ l \ xs) \]

\[
\forall P. \forall l. \forall xs. \quad \text{filter } p \ l \\
\{ r. isList \ l \ xs \star isList \ r (\text{listFilter } P \ xs) \}
\]

where

\[
\text{listFilter } P \ [] \equiv []
\]

\[
\text{listFilter } P \ (x : xs) \equiv \begin{cases} 
(x : (\text{listFilter } P \ xs)) & \text{if } P \ x \\
\text{listFilter } P \ xs & \text{otherwise}
\end{cases}
\]
Proof of filter

Let $P$, $l$ and $xs$ be given. Instantiate spec for foldr with

- $P = \lambda x. \text{true}$ (note: this is the instantiation of the $P$ in the spec for foldr, not to be confused with the parameter $P$)
- $\text{Inv } xs \ a = \text{isList } a \ (\text{listFilter } P \ xs)$
- $f = \lambda (x, y). \text{if } p \ x \ \text{then } \text{inj2} \ (\text{ref}(x, xs)) \ \text{else } xs$
- $l = l$ and $xs = xs$
Proof of foldr

Recall spec:

$$\forall P, Inv. \forall f. \forall xs. \forall l. \{ (\forall x. \forall a'. \forall ys. \{ \forall x. \forall a'. \forall ys. \{ P \land \text{inv } ys \land a' \} \land f(x, a') \} \land r. \text{inv}(x : ys) r \} \land \text{isList } l \land \text{allP } xs \land \text{inv } [] a \land \text{foldr } f a l \land r. \text{isList } l \land \text{inv } xs \land r \}$$
Proof of foldr

Idea: foldr defined by recursion, so we wish to use the \( \text{REC} \) rule. Move the nested triple into the context: we know that we can move triples in-and-out of preconditions; it also holds for quantified triples (Ch. 6). Thus SFTS:

\[
\forall x. \forall a'. \forall ys. \{ P \times \ast \text{Inv} ys a' \} f (x, a') \{ r.\text{Inv} (x : ys) \} \vdash \text{foldr} f a l \{ \text{isList} l xs \ast \text{allP} xs \ast \text{Inv} [] a \} \\
\{ r.\text{isList} l xs \ast \text{Inv} xs r \}
\]

Now proceed by the \( \text{REC} \) rule.
Formalization in Coq, using Iris Proof Mode

\[
\text{Fixpoint } \text{is_list} \ (\text{hd} : \text{val}) \ (\text{xs} : \text{list val}) : \text{iProp } \Sigma := \\
\quad \begin{array}{l}
\text{match } \text{xs} \text{ with} \\
\quad | \ [\ ] \Rightarrow \exists \ \text{hd} = \text{NONEV} \\
\quad | \ x :: \ \text{xs} \Rightarrow \exists \ l \ \text{hd'}, \ \exists \ \text{hd} = \text{SOMEV} \ #l \ \Rightarrow \ l \mapsto (x,\text{hd'}) \ * \ \text{is_list} \ \text{hd'} \ \text{xs}
\end{array}
\]
Definition inc : val :=
  rec: "inc" "hd" :=
    match: "hd" with
      NONE ⇒ #()
    | SOME "l" ⇒
        let: "tmp1" := Fst !"l" in
        let: "tmp2" := Snd !"l" in
        "l" ← ("tmp1" + #1), "tmp2");
      inc "tmp2"  
end.

Lemma inc_wp hd xs :
{ {{ is_list_nat hd xs }}
  inc hd
{ { w, RET w; [w = #() \* is_list_nat hd (map Z.succ xs) ]}}.
Proof.
iIntros (Φ) "Hxs H".
iLöb as "IH" forall (hd xs Φ). wp_rec. destruct xs as [|x xs]; iSimplifyEq.
  − wp_match. iApply "H". done.
  − iDestruct "Hxs" as (1 hd') "(\& Hx & Hxs)". iSimplifyEq.
    wp_match. do 2 (wp_load; wp_proj; wp_let). wp_op.
    wp_store. iApply ("IH" with "Hxs").
    iNext. iIntros. iApply "H". iDestruct "~" as "[Hw Hislist]".
    iFrame. iExists l, hd'. iFrame. done.
Qed.
Definition foldr : val :=
    rec: "foldr" "f" "a" "l" :=
        match: "l" with
        NONE ⇒ "a"
        | SOME "p" ⇒
            let: "hd" := Fst !"p" in
            let: "t" := Snd !"p" in
            "f" ("hd", ("foldr" "f" "a" "t"))
        end.
Lemma foldr_spec_PI P I (f a hd : val ) (e_f e_a e_hd : expr) (xs : list val) :
  to_val e_f = Some f →
  to_val e_a = Some a →
  to_val e_hd = Some hd →
  \{\{ (∀ (x a' : val) (ys : list val),
    \{\{ P x *I ys a'\}\}
    e_f (x, a')
    \{\{r, RET r; I (x::ys) r \}\}\}
  * is_list hd xs
  * ([* list] x ∈ xs, P x)
  * I [] a
\}\}.
foldr e_f e_a e_hd
  \{\{r, RET r; is_list hd xs
   * I xs r
\}\}.
Proof.
apply of_to_val in Hef as ←.
apply of_to_val in Hea as ←.
apply of_to_val in Hehd as ←.
iIntros (Φ) "(#H_f & H_isList & H_Px & H_Iempty) H_inv".
iInduction xs as [ [x xs'] "IH" forall (Φ a hd); wp_rec; do 2 wpLet; iSimplifyEq.
— wp_match. iApply "H_inv". eauto.
— iDestruct "H_isList" as (1 hd') "[% [H_l H_isList]]".
  iSimplifyEq.
wp_match. do 2 (wp_load; wp_proj; wpLet).
wp_bind (((foldr f) a) hd').
iDestruct "H_Px" as "(H_Px & H_Pxs')".
iApply ("IH" with "H_isList H_Pxs' H_Iempty [H_l H_Px H_inv]").
iNext. iIntros (r) "(H_isListxs' & H_Ixs')".
iApply ("H_f" with "[H_jxs'H_Px] [H_inv H_isListxs' H_l]").
iNext. iIntros (r') "H_inv'". iApply "H_inv". iFrame.
iExists l, hd' by iFrame.
Qed.
Lemma sum_spec (hd: val) (xs: list Z) :
{{ is_list hd (map (fun n ⇒ LitV (LitInt n)) xs) }}
sum_list hd
{{ v, RET v; v = LitV (LitInt (fold_right Z.add 0 xs)) }}).

Proof.
iIntros (Φ) "H_is_list H_later".
wp_rec. wp_let.
\[\begin{align*}
  & \text{iApply (foldr_spec_PI} \\
  & \quad (\text{fun x ⇒ } (\exists (n : Z), \text{⌜}x = \# n\text{⌟}\%I) \\
  & \quad (\text{fun xs'} \ acc ⇒ \exists ys, \\
  & \quad \quad \text{⌜}acc = \#(\text{fold_right Z.add 0 ys)}\text{⌟} \\
  & \quad \quad * \ \text{⌜}xs' = \text{map (fun (n : Z) ⇒ \# n) ys \text{⌟} \\
  & \quad \quad * (\text{⌜ list} \ x ∈ xs', \exists (n' : Z), \text{⌜}x = \# n'\text{⌟})\%I} \\
  & \quad \text{with } "[\text{H_is_list}] [\text{H_later}]". \\
  & \text{− iSplitR.} \\
  & \quad + \text{iIntros (x a' ys). iAlways. iIntros (Φ') }"(H1 \ & \ H2) \ H3". \\
  & \quad \text{do 5 (wpPure _).} \\
  & \quad \text{iDestruct "H2" as (zs) }"(\% \ & \ % \ & \ H_list)". \\
  & \quad \text{iDestruct "H1" as (n2) }"\%". \text{iSimplifyEq. wp_binop.} \\
  & \quad \text{iApply } "H3". \text{iExists (n2::zs). repeat (iSplit; try done).} \\
  & \quad \text{by iExists _}. \\
  & \quad + \text{iSplit.} \\
  & \quad \quad * \text{induction xs; iSimplifyEq; first done.} \\
  & \quad \quad \text{iSplit; [iExists a; done | apply IHxs].} \\
  & \quad \quad * \text{iExists []}. \text{eauto.} \\
  & \text{− iNext. iIntros (r) }"(H1 \ & \ H2)". \\
  & \quad \text{iApply } "H_later". \text{iDestruct "H2" as (ys) }"(\% \ & \ % \ & \ H_list)". \\
  & \quad \text{iSimplifyEq. rewrite (map_injective xs ys (λ n : Z, \# n)); try done.} \\
  & \quad \text{unfold inj. intros x y H_xy. by inversion H_xy.} \\
\end{align*}\]
Qed.
Lemma filter_spec (hd p : val) (xs : list val) P :
{{is_list hd xs
  * (\forall x : val, {{True}})
    P x
    {{(r, RET r; \exists b, \[r = LitV \langle LitBool b \rangle \land \[b = P \ x \]\)}}
}}
filter p hd
{{(v, RET v; is_list hd xs
  * is_list v (List.filter P xs)
)}}.

Proof.
iIntros (Φ) "[H_isList \#H_p] H_Φ".
do 3 (wp Pure _).
iApply (foldr_spec_PI (fun x ⇒ True)\%
  (fun xs' acc ⇒ is_list acc (List.filter P xs'))\%
  with "[\$H_isList] [H_Φ]").
  - iSplitL.
    + iIntros "** !#" (Φ'). iIntros "[\_ H_isList] H_Φ'".
      repeat (wp Pure _). wp_bind (p x). iApply "H_p": first done.
      iNext. iIntros (r "H". iSimplifyEq. destruct (P x): wp_if.
      * unfold cons. repeat (wp Pure _). wp_alloc l. iApply "H_Φ".
        iExists l, a'. by iFrame.
        * by iApply "H_Φ'".
      + iSplit; last done.
        rewrite big_sepl_forall. eauto.
    - iNext. iApply "H_Φ".
Qed.