Iris: Higher-Order Concurrent Separation Logic

Lecture 6: Case Study: foldr

Lars Birkedal

Aarhus University, Denmark

November 10, 2017

Overview

Earlier:

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

isList

- Recall the isList predicate, defined by induction on the mathematical sequence xs.

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

foldr

- Intuitive type:

$$\mathsf{foldr} : (\alpha \times \beta \to \beta) \to \beta \to \alpha \text{ list} \to \beta$$

$$\begin{aligned}
\mathsf{rec\,foldr}(f\ \ a\ \ l) = \ & \mathsf{match}\ l\ \mathsf{with} \\
& \ \ \mathsf{inj}_1\, x_1 \Rightarrow a \\
& \mid \mathsf{inj}_2\, x_2 \Rightarrow \mathsf{let}\ h = \pi_1\ !\, x_2\ \mathsf{in} \\
& \qquad\qquad\quad\ \mathsf{let}\ t = \pi_2\ !\, x_2\ \mathsf{in} \\
& \qquad\qquad\quad\ f\ \ (h\ , (\mathsf{foldr}\ \ f\ \ a\ \ t)) \\
& \mathsf{end}
\end{aligned}$$

Specification of foldr

$$\forall P, Inv. \forall f. \forall xs. \forall l. \left\{ \begin{array}{l} (\forall x. \forall a'. \forall ys. \{Px * Inv\ \ ys\ \ a'\}\, f\ \ (x,\ \ a')\, \{r. Inv(x : ys) r\}) \\ * \operatorname{isList}\ \ l\ \ xs * \operatorname{allP}\ \ xs * Inv\ \ []\ \ a \end{array} \right\} \\ \quad \texttt{foldr}\ f\ \ a\ \ l \\ \{r. \operatorname{isList}\ \ l\ \ xs * Inv\ \ xs\ \ r\}$$

where

$$\begin{aligned} \operatorname{allP}\ [] &\equiv \operatorname{True} \\ \operatorname{allP}\ (x : xs) &\equiv P\ \ x * \operatorname{allP}\ \ xs \end{aligned}$$

Remarks about the Specification

- The $\lambda_{\mathrm{ref,conc}}$ 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 $Inv : \mathrm{list}\ Val \to Val \to \mathrm{Prop}$, where $\mathrm{list}\,Val$ 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 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

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

$$\forall l.\, \forall xs.\, \{\mathsf{isList}\ \ l\ xs * \mathsf{allNats}\ \ xs\}\, \mathsf{sumList}\ \ l\, \{r.\, \mathsf{isList}\ \ l\ xs * r = \Sigma_{x \in xs} x\}$$

where

$$\begin{aligned} \mathsf{allNats}\ [] &\equiv \mathsf{True} \\ \mathsf{allNats}\ (x : xs) &\equiv \mathsf{isNat}\ \ x * \mathsf{allNats}\ \ xs \\ \mathsf{isNat}\ \ x &\equiv \begin{cases} \mathsf{True} & \text{if } x \in \mathbb{N} \\ \mathsf{False} & \text{otherwise} \end{cases} \end{aligned}$$

Proof of sumList

Let l and xs be arbitrary. Instantiate spec for foldr with

- $P = \text{isNat}$
- $Inv\ ys\ a' = (a' =_{\mathbb{N}} \Sigma_{y \in ys} y)$
- $f = \lambda(x, y).x + y$ and $l = l$ and $xs = xs$

to get

$$\left\{ \begin{array}{l} \left(\forall x, a.\, \forall ys.\, \{\text{isNat}\ \ x * a = \Sigma_{y \in ys} y\}\, (\lambda(x, y).x + y)(x,\ \ a)\, \{r.r = \Sigma_{y \in (x:ys)}\}\right) \\ * \,\text{isList}\ \ l\ xs * \text{allNats}\ \ xs * 0 = \Sigma_{x \in \emptyset} x \end{array} \right\}$$

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

$$\{r.\, \text{isList}\ \ l\ xs * r = \Sigma_{x \in xs} x\}$$

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

Proof of sumList

By rule of consequence SFTS

$$\text{isList}\ \ l\ xs * \text{allNats}\ \ xs$$
$$\Rightarrow$$
$$\left(\forall x, a. \forall ys. \{\text{isNat}\ \ x * a = \Sigma_{y \in ys} y\}\,(\lambda(x, y).x + y)(x,\ \ a)\,\{r.r = \Sigma_{y \in (x:ys)}\}\right)$$
$$* \,\text{isList}\ \ l\ xs * \text{allNats}\ \ xs * 0 = \Sigma_{x \in \emptyset} x$$

which is left as exercise.

Client: filter

$$
\begin{aligned}
\mathsf{rec\ filter}(p\ \ l) = \quad & \mathsf{let}\ f = (\lambda(x, xs). \quad \mathsf{if}\ p\ \ x \\
& \qquad\qquad\qquad\qquad\quad \mathsf{then\ inj}_2\,(\mathsf{ref}(x, xs)) \\
& \qquad\qquad\qquad\qquad\quad \mathsf{else}\ xs) \\
& \ \mathsf{in} \\
& \mathsf{foldr}\ \ f\ \ []\ \ l
\end{aligned}
$$

Specification of filter

$$\forall P.\, \forall l.\, \forall xs. \quad \begin{array}{l} \{(\forall x.\, \{\mathsf{true}\}\; p\; x\; \{v.\, \mathrm{isBool}\;\; v * v = P\;\; x\}) * \mathrm{isList}\;\; l\;\; xs\} \\ \quad \mathsf{filter}\; p\;\; l \\ \{r.\, \mathrm{isList}\;\; l\;\; xs * \mathrm{isList}\;\; r\, (\mathrm{listFilter}\;\; P\;\; xs)\} \end{array}$$

where

$$\begin{aligned} \mathrm{listFilter}\;\; P\; [] &\equiv\;\; [] \\ \mathrm{listFilter}\;\; P\; (x : xs) &\equiv \begin{cases} (x : (\mathrm{listFilter}\;\; P\;\; xs)) & \text{if}\;\; P\;\; x \\ \mathrm{listFilter}\;\; P\;\; xs & \text{otherwise} \end{cases} \end{aligned}$$

Proof of filter

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

- $P = \lambda x.\mathsf{true}$ (note: this is the instantiation of the P in the spec for foldr, not to be confused with the parameter P)
- $Inv\ xs\ a = \text{isList}\ \ a\ \ (\text{listFilter}\ \ P\ xs)$
- $f = \lambda(x, y).\mathsf{if}\, p\ x\, \mathsf{then}\, \mathsf{inj}_2\, (\mathsf{ref}(x, xs))\, \mathsf{else}\, xs$
- $l = l$ and $xs = xs$

Proof of foldr

Recall spec:

$$\forall P, Inv. \forall f. \forall xs. \forall l. \left\{ \begin{array}{l} (\forall x. \forall a'. \forall ys. \{Px * Inv\ ys\ a'\}\, f\ (x,\ a')\, \{r. Inv(x : ys) r\}) \\ * \, \mathrm{isList}\ \ l\ xs * \mathrm{allP}\ \ xs * Inv\ []\ a \end{array} \right\}$$
$$\textcolor{blue}{\mathrm{foldr}}\ f\ \ a\ \ l$$
$$\{r.\, \mathrm{isList}\ \ l\ xs * Inv\ \ xs\ \ r\}$$

Proof of foldr

Idea: foldr defined by recursion, so we wish to use the $\textsc{Rec}$ 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:

$$\forall x.\, \forall a'.\, \forall ys.\, \{P\ x * Inv\ ys\ a'\}\, f\ (x,\ a')\, \{r. Inv\ (x : ys)\} \vdash \begin{array}{l} \{\text{isList}\ \ l\ xs * \text{allP}\ \ xs * Inv\ []\ a\} \\ \quad \text{foldr}\ f\ a\ l \\ \{r.\, \text{isList}\ \ l\ xs * Inv\ xs\ r\} \end{array}$$

Now proceed by the $\textsc{Rec}$ rule.

Formalization in Coq, using Iris Proof Mode

```
Fixpoint is_list (hd : val) (xs : list val) : iProp Σ :=
  match xs with
  | [] ⇒ ⌜hd = NONEV⌝
  | x :: xs ⇒ ∃ l hd', ⌜hd = SOMEV #l⌝ ∗ l ↦ (x,hd') ∗ is_list hd' xs
  end
```

inc from last week

```
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 (l 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.
```

foldr

```
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.
```

foldr

```
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
  }}}.
```

foldr proof

```
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 wp_let; iSimplifyEq.
    wp_match. iApply "H_inv". eauto.
    iDestruct "H_isList" as (l hd') "[% [H_l H_isList]]".
    iSimplifyEq.
    wp_match. do 2 (wp_load; wp_proj; wp_let).
    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_l xs' H_Px] [H_inv H_isListxs' H_l]").
    iNext. iIntros (r') "H_inv'". iApply "H_inv". iFrame.
    iExists l, hd'. by iFrame.
Qed.
```

sumList

```
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.
  iApply (foldr_spec_PI
            (fun x ⇒ (∃ (n : Z), ⌜x = #n⌝)%I)
            (fun xs' acc ⇒ ∃ ys,
                 ⌜acc = #(fold_right Z.add 0 ys)⌝
               ∗ ⌜xs' = map (fun (n : Z) ⇒ #n) ys⌝
               ∗ ([∗ list] x ∈ xs',∃ (n' : Z), ⌜x = #n'⌝))%I
            with "[$ H_is_list] [H_later]").
    iSplitR.
    + iIntros (x a' ys). iAlways. iIntros (Φ') "(H1 & H2) H3".
      do 5 (wp_pure _).
      iDestruct "H2" as (zs) "(% & % & H_list)".
      iDestruct "H1" as (n2) "%". iSimplifyEq. wp_binop.
      iApply "H3". iExists (n2::zs). repeat (iSplit; try done).
      by iExists _.
    + iSplit.
      ∗ induction xs; iSimplifyEq; first done.
        iSplit; [iExists a; done | apply IHxs].
      ∗ iExists []. eauto.
    iNext. iIntros (r) "(H1 & H2)".
    iApply "H_later". iDestruct "H2" as (ys) "(% & % & H_list)".
    iSimplifyEq. rewrite (map_injective xs ys (λ n : Z, #n)); try done.
    unfold inj. intros x y H_xy. by inversion H_xy.
Qed.
```

filter

```
Lemma filter_spec (hd p : val) (xs : list val) P :
  {{{ is_list hd xs
      ∗ (∀ x : val , {{{ True }}}
           p x
           {{{r, RET r; ∃ b, ⌜ r = LitV (LitBool b) ⌝ ∗  ⌜ 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)%I
                        (fun xs' acc ⇒ is_list acc (List.filter P xs'))%I
                  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.
```