Lecture 6: Case Study: foldr

Lars Birkedal

Aarhus University, Denmark

November 10, 2017
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
  - $l \hookrightarrow v, P \ast Q, P \nind Q, \Gamma \mid P \vdash Q$
- Basic Separation Logic
  - $\{P\} e \{v.Q\} : \text{Prop, isList } l \; xs$
- 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.
Recall the isList predicate, defined by induction on the mathematical sequence \(xs\).

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

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

- Intuitive type:

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

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

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

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

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

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

end
Specification of foldr

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

\text{foldr} \ f \ a \ l  

\{ r.\text{isList} \ l \ xs \ast \text{Inv} \ \ x \ s \ r \}  

where  

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

\text{allP} \ (x : xs) \equiv P \ x \ast \text{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 $\text{Inv}$ has type $\text{Inv} : \text{list Val} \rightarrow \text{Val} \rightarrow \text{Prop}$, 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 $\text{Inv}$ to allow clients to instantiate those. The idea is that $P$ is a predicate that holds for each element in the given list, and $\text{Inv} \; xs \; a$ expresses that $a$ is the result of folding $f$ over $xs$. 
Client: sumList

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

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

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 \texttt{foldr} with

\begin{itemize}
  \item \( P = \text{isNat} \)
  \item \( \text{Inv } ys \ a' = (a' = \mathbb{N} \Sigma_{y \in ys} y) \)
  \item \( f = \lambda(x, y).x + y \) and \( l = l \) and \( xs = xs \)
\end{itemize}

to get

\[
\begin{align*}
\{ & (\forall x, a. \forall ys \{ \text{isNat } x \ast a = \Sigma_{y \in ys} y \} (\lambda(x, y).x + y)(x, a) \{ r.r = \Sigma_{y \in (x:ys)} \} ) \} \\
& \ast \text{isList } l \ xs \ast \text{allNats } xs \ast 0 = \Sigma_{x \in \emptyset} x \\
& \text{foldr } (\lambda(x, y).x + y) \ a \ l \\
\} \text{isList } l \ xs \ast r = \Sigma_{x \in xs} x \\
\]

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

By rule of consequence SFTS

isList \ l \ xs \ * \ allNats \ xs

\Rightarrow

(\forall x, a. \ \forall ys. \ \{isNat \ x * a = \sum_{y \in ys} y\} (\lambda (x, y). x + y)(x, a) \{r. r = \sum_{y \in (x:ys)}\})

* isList \ l \ xs \ * \ allNats \ xs \ * \ 0 = \sum_{x \in \emptyset} x

which is left as exercise.
Client: filter

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

\[
\{ (\forall x. \{ \text{true} \} \; p \; x \; \{ v. \text{isBool} \; v \; \ast \; v = P \; x \} ) \ast \text{isList} \; l \; xs \} \\
\forall P. \forall l. \forall xs. \quad \text{filter} \; p \; l \\
\quad \{ r. \text{isList} \; l \; xs \ast \text{isList} \; r \; (\text{listFilter} \; P \; xs) \} \\
\]

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 $P$)
- $\text{Inv } xs \ a = \text{isList } a \ (\text{listFilter } P \ xs)$
- $f = \lambda (x, y). \text{if } p x \text{ then inj}_2 (\text{ref}(x, xs)) \text{ else xs}$
- $l = l$ and $xs = xs$
Proof of foldr

Recall spec:

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

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

$$\forall x. \forall a'. \forall ys. \{ P \ x * \text{Inv} \ ys \ a' \} \ f \ (x, \ a') \ {r.\text{Inv} \ (x : ys)} \vdash \ \texttt{foldr} \ f \ a \ l$$

Now proceed by the \texttt{REC} rule.

\[
\{\text{isList} \ l \ xs * \text{allP} \ xs * \text{Inv} \ [] \ a\}
\]

\[
\forall x. \forall a'. \forall ys. \{ P \ x * \text{Inv} \ ys \ a' \} \ f \ (x, \ a') \ {r.\text{Inv} \ (x : ys)} \vdash \ \texttt{foldr} \ f \ a \ l
\]

\[
\{r.\text{isList} \ l \ xs * \text{Inv} \ xs \ r\}
\]
Fixpoint is_list (hd : val) (xs : list val) : iProp Σ :=
match xs with
| [] ⇒ ⌜hd = NONE\r
| x :: xs ⇒ ∃ l hd', ⌜hd = SOME #l\r
  l↦→(x,hd') * is_list hd' xs
end
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 \( f \ a \ hd : \text{val} \) \( e_f \ e_a \ e_{\text{hd}} : \text{expr} \) \( xs : \text{list val} \) :

to_val e_f = Some f >
to_val e_a = Some a >
to_val e_{\text{hd}} = Some hd >

\[
\begin{aligned}
&\{\forall (x \ a' : \text{val}) (ys : \text{list val}),
&\quad \{\{ P x \star I \ ys \ a'\}\}\}
&\quad e_f (x, a')
&\quad \{\{r, \text{RET} \ r; I (x::ys) \ r\}\}\}
&\quad \ast \text{is\_list} \ hd \ xs
&\quad \ast ([\ast \text{list}] \ x \in \ xs, P x)
&\quad \ast I [] \ a
\end{aligned}
\]

foldr e_f e_a e_{\text{hd}}

\[
\begin{aligned}
&\{\}
&\quad r, \text{RET} \ r; \text{is\_list} \ hd \ xs
&\quad \ast I \ xs \ r
\end{aligned}
\]
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) "(#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 ("f with "[H_free 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
{RET v; \( v = LitV (LitInt (fold_right Z.add 0 xs)) \)}.

Proof.
iIntros (Φ) "H_is_list HLater".
wp_rec. wp_let.
iApply (foldr_spec_PI
  (fun x ⇒ (∃ (n : Z), \( x = \#n \))%I)
  (fun xs' acc ⇒ (∃ ys, \( acc = #(fold_right Z.add 0 ys) \))%
  \( x \in \) xs',∃ (n' : Z), \( x = \#n' \))%I
  with "[H_is_list] [HLater]").
iSplitR.
iIntros (x a' ys). iAlways. iIntros (Φ') "(H1 & H2) H3".
do 5 (wp_let _).
iDestruct "H2" as (zs) "(\% & \% & H_list)".
iDestruct "H1" as (n2) "\%". iSimplifyEq. wp_binop.
iApply "H3". iExists (n2::zs). repeat (iSplit; try done).
by iExists _.
iNext. iIntros (Φ) "(H1 & H2)".
iApply "HLater". iDestruct "H2" as (ys) "(\% & \% & H_list)".
iSimplifyEq. rewrite (map_injective xs ys (\n : Z, \#n)); try done.
unfold inj. intros x y H_xy. by inversion H_xy.
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, \forall r = LitV (LitBool b) \implies \exists b = P x)}}}}}
filter p hd
{{{v, RET v; is_list hd xs
    * is_list v (List.filter P xs)}}}.

Proof.
iIntros (\Phi) "[H_isList \#H_p] H_\Phi".
do 3 (wpPure _).
iApply (foldr_spec_PI (fun x \Rightarrow True)I
  (fun xs' acc \Rightarrow is_list acc (List.filter P xs'))I
   with "[\$H_isList] [H_\Phi]").
iSplitL.
iIntros "** !#" (\Phi').
iIntros "[._ H_isList] H_\Phi'."
  repeat (wpPure _). wp_bind (p x). iApply "H_p"; first done.
iNext. iIntros (r) "H". iSimplifyEq destruct (P x); wp_if.
  * unfold cons. repeat (wpPure _).
    wp_alloc l. iApply "H_\Phi".
    iExists l, a'. by iFrame.
    * by iApply "H_\Phi'".
iSplit; last done.
  rewrite big_sepL_forall. eauto.
iNext. iApply "H_\Phi".
Qed.