Characteristic Formulae with Lifting

Arthur Charguéraud

Inria

May 24th, 2023

Overview

Smooth embedding in Coq of Separation Logic for ML programs

1. Quick demo of CFML
2. Lifting: mapping OCaml values to Coq values
3. Unlifted characteristic formulae
4. Lifted characteristic formulae, in a foundational way

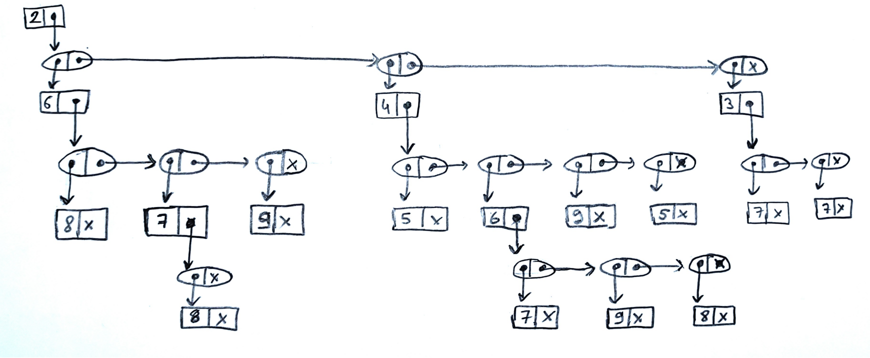
— Quick demo of CFML —

Imperative pairing heaps

```
type heap = contents ref

and contents = | Empty
               | Nonempty of node
```

```
type node = {
  mutable value : int;
  mutable sub : node MList.t }
```



Imperative pairing heaps: OCaml implementation

```
let merge q1 q2 =
  if q1.value < q2.value
    then (MList.push q1.sub q2; q1)
    else (MList.push q2.sub q1; q2)

let insert p x =
  let q2 = {
    value = x;
    sub = MList.create() } in
  match !p with
  | Empty ->
      p := Nonempty q2
  | Nonempty q1 ->
      p := Nonempty (merge q1 q2)

let create () =
  ref Empty
```

```
let rec merge_pairs l =
  let q1 = MList.pop l in
  if MList.is_empty l then q1 else
  let q2 = MList.pop l in
  let q = merge q1 q2 in
  if MList.is_empty l
    then q
    else merge q (merge_pairs l)

let pop_min p =
  match !p with
  | Empty -> assert false
  | Nonempty q ->
    let x = q.value in
    if MList.is_empty q.sub
      then p := Empty
      else p := Nonempty
                  (merge_pairs q.sub);
    x
```

Representation predicates for pairing heaps

```
Inductive node : Type := Node : elem → list node → node. (* Functional tree *)

(* [inv n E] relates a tree node [n] with the multiset [E] of its items *)
Inductive inv : node → elems → Prop :=
  | inv_Node : ∀x ns Es E,
      Forall2 inv ns Es →
      Forall (foreach (is_ge x)) Es →
      E = {x} ∪ (list_union Es) →
      inv (Node x ns) E.

(* [q ⇝ Tree n] relates a pointer [q] on a tree with a functional tree [n].
   [q ⇝ Tree n] is a notation for [Tree n q]. *)
Fixpoint Tree (n:node) (q:loc) : hprop :=
  match n with
  | Node x hs ⇒ ∃(p:loc), q ⇝ { value':=x; sub':=p } ⋆ p ⇝ MListOf Tree hs
  end.

(* [q ⇝ Repr E], relates a pointer [q] on a tree with a multiset [E] *)
Definition Repr (E:elems) (q:loc) : hprop := ∃n, (q ⇝ Tree n) ⋆ [inv n E].

(* [p ⇝ Heap E] relates a pointer [p] on a heap with a multiset [E] *)
Definition Heap (E:elems) (p:heap_) : hprop :=
  ∃c, (p ↪ c) ⋆ match c with | Empty ⇒ [E = ∅]
                             | Nonempty q ⇒ q ⇝ Repr E
                             end.
```

Example proof of a recursive function

```
let rec merge_pairs l =
  let q1 = MList.pop l in
  if MList.is_empty l then q1 else
  let q2 = MList.pop l in
  let q = merge q1 q2 in
  if MList.is_empty l
     then q
     else merge q (merge_pairs l)
```

```
Lemma Triple_merge_pairs : ∀ns l Es,
  ns ≠nil →
  Forall2 inv ns Es →
  SPEC (merge_pairs l)
    PRE (l ⇝ MListOf Tree ns)
    POST (fun q ⇒ q ⇝ Repr (list_union Es)).

Proof using.
  intros ns. induction_wf IH: list_sub ns; introv N Is.
  xcf. xapp* ;⇒ q1 n1 ns' →. inverts Is as I1 Is. xif ;⇒ C.
  { subst. inverts Is. rew_listx. xval. xchanges* ←Repr_eq. }
  { xapp* ;⇒ q2 n2 ns'' →. inverts Is as I2 Is.
    do 2 xchange* ←Repr_eq. xapp ;⇒ r. xif ;⇒ C'.
    { subst. inverts Is. rew_listx. xval. xsimpl. }
    { xapp* ;⇒ r'. xapp ;⇒ r''. rew_listx. xsimpl*. } }
Qed.
```

Example proof obligation

```
let q2 = MList.pop l in
let q = merge q1 q2 in
if MList.is_empty l
   then q
   else merge q (merge_pairs l)
```

```
IH : ∀(y : list node), list_sub y (n1 :: ns') → ∀l Es, y ≠nil →
     Forall2 inv y Es → SPEC (merge_pairs l)
                        PRE (l ⇝ MListOf Tree y)
                        POST (fun q ⇒ q ⇝ Repr (list_union Es))
N : n1 :: ns' ≠nil
I1 : inv n1 y
Is : Forall2 inv ns' Es'
C : ns' ≠nil
______________________________________(1/1)
PRE (l ⇝ MListOf Tree ns' ⋆ q1 ⇝ Tree n1)
CODE (Let q2 := App MList_ml.pop l in
      Let q := App merge q1 q2 in
      Let x1__ := App MList_ml.is_empty l in
      If_ x1__ Then
        Val q
      Else
        Let x2__ := App merge_pairs l in
        App merge q x2__)
POST (fun q ⇒ q ⇝ (Repr (list_union (y :: Es'))) ⋆ ⊤)
```

— Lifting —

Motivation #1: lifted postconditions

Instead of $\{[\,]\}\ (\mathsf{ref}\, v)\ \{\lambda(r : \mathsf{val}).\ \exists(p : \mathsf{loc}).\ [r = \mathsf{val\_loc}\, p] \star (p \hookrightarrow v)\}$

we write $\{[\,]\}\ (\mathsf{ref}\, v)\ \{\lambda(p : \mathsf{loc}).\ (p \hookrightarrow v)\}$

Likewise for any type, e.g.: $\{...\}\ ...\ \{\lambda(t : \mathsf{tree}\, A).\ ...\}$

Implementation of encoders

```
Class Enc (A:Type) : Type :=
  { enc : A → val;
    enc_inj : injective enc }.

Instance Enc_bool : Enc bool := {| enc := val_bool; ... |}.
Instance Enc_int : Enc int := {| enc := val_int; ... |}.
Instance Enc_list : ∀(A:Type) {EA:Enc A}, Enc (list A) := ...

Definition Post (A:Type) {EA:Enc A} (Q:A→hprop) : val→hprop :=
  fun (v:val) ⇒ ∃(V:A), [v = enc V] ⋆ Q V.

Definition Triple (t:trm) A {EA:Enc A} (H:hprop) (Q:A→hprop) : Prop :=
  triple t H (Post Q).
```

Motivation #2: lifted let-binding rule

Quantify program variables directly at the appropriate Coq type.

```
Lemma Triple_let :
  ∀X t1 t2 H,
  ∀(A:Type) (EA:Enc A) (Q:A→hprop),
  ∀(T:Type) {EB:Enc T} (Q1:T→hprop),
  Triple t1 H Q1 →
  (∀ (X:T), Triple (subst x (enc X) t2) (Q1 X) Q) →
  Triple (trm_let x t1 t2) H Q.
```

Lifted WP

```
let b : bool = f a in ..
```

```
WP (f a) (fun (b:bool) ⇒ ...)
```

Reasoning rules apply to an untyped deeply embedded language.
CFML's approach: input typed OCaml code and output a *lifted* WP.

Motivation #3: lifted points-to predicates

`p ↪ v` where `v` is a Coq value is defined as `hprop_single p (enc v)`

Example: mutable list

```
type 'a mlist = ('a mcell) ref
and 'a mcell = Nil | Cons of 'a * 'a mcell
```

Generated definitions

```
Definition mlist (A : Type) : Type := loc.

Inductive mcell (A : Type) : Type :=
  | Nil : mcell A
  | Cons : A → mlist A → mcell A.

Global Instance Enc_mcell (A:Type) {EA:Enc A} : Enc (mcell A) :=
  { enc :=
      fun (v : mcell A) ⇒
        match v with
        | Nil ⇒ val_constr "Nil" [ ]
        | Cons x r ⇒ val_constr "Cons" [ enc x; enc r ]
        end;
    enc_inj := ... }.
```

Motivation #4: lifted representation predicates

	$\mathsf{Mlist}\,(\mathsf{val\_int}\,1 :: \mathsf{val\_int}\,2 :: \mathsf{val\_int}\,3 :: \mathsf{nil})\,p$
or	$\mathsf{Mlist}\,(\mathsf{List.map}\,\mathsf{val\_int}\,(1 :: 2 :: 3 :: \mathsf{nil}))\,p$
or	$\mathsf{MlistOf}\,\mathsf{Int}\,(1 :: 2 :: 3 :: \mathsf{nil}))\,p$
we can write	$\mathsf{Mlist}\,(1 :: 2 :: 3 :: \mathsf{nil})\,p$

C-style mutable lists

```
Fixpoint MList A {EA:Enc A} (L:list A) (p:loc) : hprop :=
  match L with
  | nil ⇒ [p = null]
  | x::L' ⇒ ∃p', p ↪ { head := enc x; tail := p' } ⋆ (MList L' p')
  end.
```

OCaml-style mutable lists

```
Fixpoint MList A {EA:Enc A} (L:list A) (p:loc) : hprop :=
  ∃(v:mcell A), p ↪ v ⋆
  match L with
  | nil ⇒ [v = Nil]
  | x::L' ⇒ ∃p', [v = Cons x p'] ⋆ (MList L' p')
  end.
```

Motivation #5: lifted pattern matching

```
type color = Red | Black
type rbtree = Empty | Node of color * rbtree * int * rbtree
match ts with
| (Black, Node (Red, Node (Red, a, x, b), y, c), z, d) -> ..
```

Without lifting

```
∀(a x b y c z d : val),
  [ts = val_constr "tuple" [
        val_constr "Black" [ ];
        val_constr "Node" [
          val_constr "Red" [ ];
          val_constr "Node" (val_constr "Red" [a; x; b]);
          y; c];
        z; d] ] ⋆...
```

With lifting

```
Inductive color_ : Type := Red : color_ | Black : color_.
Inductive rbtree_ : Type :=
  Empty : rbtree_ | Node : color_ → rbtree_ → Z → rbtree_ → rbtree_.

 ∀(a : rbtree_) (x : int) (b : rbtree_) (y : int) (c : rbtree_) (z : int) (d : rbtree_),
   [ts = (Black, Node Red (Node Red a x b) y c, z, d)] ⋆...
```

— Characteristic formulae without lifting —

Principle of characteristic formulae

The *characteristic formula* of a term t is a logic formula "$\mathsf{cf}\,t$" such that:

$$t_1 \approx_{\mathsf{obs}} t_2 \quad\Longleftrightarrow\quad \mathsf{cf}\,t_1 =_{\mathsf{logic}} \mathsf{cf}\,t_2$$

Enables reasoning about programs *without* referring to program syntax.

History of characteristic formulae (CF)

1985 Hennessy-Milner: CF for a process calculus

2005 Berger-Honda-Yoshida: CF in an ad-hoc first-order Hoare logic

2010 CFML: CF for OCaml with lifting, in higher-order SL;
Foundational CF for *Imp* without lifting

2015 Guéneau et al.: foundational CF for CakeML without lifting

2022 CFML: foundational CF for OCaml with lifting, in WP-style

From WP to characteristic formulae

$$\{H\}\ t\ \{Q\} \quad\iff\quad H \vdash \mathsf{wp}\, t\, Q$$

- wp viewed as a predicate applied to a piece of syntax (Iris)
- wp computed for a term annotated with invariants (VCgen)
- wp computed for a term without invariants (characteristic formula)

Soundness of characteristic formulae:

$$\mathsf{cf}\, t\, Q \vdash \mathsf{wp}\, t\, Q$$

Characteristic formula generator

$\mathsf{cf}_E\, t\, Q$ computes the WP of t in A-normal form with respect to a postcondition Q, and E binds free program variables to Coq variables.

$$\begin{array}{lcl}
\mathsf{cf}_E\, x & \equiv & \mathsf{framed}\,(\lambda Q.\, \mathsf{If}\ (x \in \mathsf{dom}\, E)\ \mathsf{then}\ Q\,(E[x])\ \mathsf{else}\ \bot) \\
\mathsf{cf}_E\, v & \equiv & \mathsf{framed}\,(\lambda Q.\, Q\, v) \\
\mathsf{cf}_E\,(t_1\, t_2) & \equiv & \mathsf{framed}\,(\lambda Q.\, \mathsf{wp}\,(\mathsf{subst}\, E\,(t_1\, t_2))\, Q) \\
\mathsf{cf}_E\,(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) & \equiv & \mathsf{framed}\,(\lambda Q.\, \mathsf{cf}_E\, t_1 \big(\lambda X.\ \mathsf{cf}_{(x,X)::E}\, t_2\, Q\big)) \\
\mathsf{cf}_E\,(\mu f.\lambda x.t) & \equiv & \mathsf{framed}\,(\lambda Q.\, \forall F.\ [\mathcal{H}] \mathrel{-\!\!\star} Q\, F) \\
 & & \qquad \mathcal{H} = \forall X Q'.\ \mathsf{cf}_{(f,F)::(x,X)::E}\, t\, Q' \vdash \mathsf{wp}\,(F\, X)\, Q'
\end{array}$$

The predicate transformer framed enables applications of the frame rule.

$$\mathsf{framed}\,\mathcal{F} \;\equiv\; \lambda Q.\;\; \exists Q'.\ (\mathcal{F}\, Q') \star (Q' \mathrel{-\!\!\star} Q)$$

Formalization and soundness

```
Fixpoint cf (E:ctx) (t:trm) : (val→hprop)→hprop :=
  framed (match t with
  | trm_var x ⇒ match lookup x E with
                | None ⇒ fun Q ⇒ [False]
                | Some v ⇒ fun Q ⇒ Q v
                end
  | trm_val v ⇒ fun Q ⇒ Q v
  | trm_fix f x t1 ⇒ fun Q ⇒
      let H := (∀ X Q', cf ((f,F)::(x,X)::E) t1 Q' ⊢ wp (trm_app F X) Q') in
      ∀F, [H] ⋆ Q F
  | trm_app t1 t2 ⇒ wp (isubst E t)
  | trm_let x t1 t2 ⇒ fun Q ⇒ (cf E t1) (fun v ⇒ cf ((x,v)::E) t2 Q)
  end).
```

Soundness of characteristic formulae

```
Theorem cf_sound : ∀t Q,
  cf nil t Q ⊢ wp t Q.
```

— Characteristic formulae with lifting —

Constructors for lifted characteristic formulae

Without lifting

```
Definition formula : Type := (val → hprop) → hprop.
Definition wp (t:trm) : formula := ...

Definition framed (f:formula) : formula := ...
Definition cf_let (f1:formula) (f2of:val→ formula) : formula :=
  framed (fun (Q:val→ hprop) ⇒ f1 (fun (X:val) ⇒ f2of X Q)).
```

With lifting

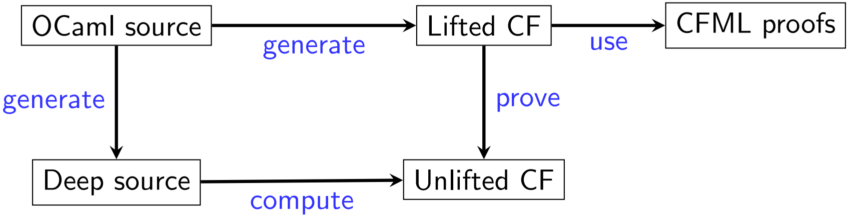
```
Definition Formula : Type := ∀(A:Type) (EA:Enc A), (A → hprop) → hprop.

Definition Wp (t:trm) : Formula :=
  fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒ wp t (Post Q).

Definition Framed (F:Formula) : Formula :=
  fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒ framed (@F A EA) Q.

Definition CF_let (F1:Formula) (A1:Type) {EA1:Enc A1} (G2:A1→ Formula) :Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒ F1 _ _ (fun (X:A1) ⇒ (G2 X) _ _ Q)).
```

CFML workflow



Imperative pairing heaps: generated definitions

```
Definition node_ : Type := loc.

Definition value' : field := (0)%nat.
Definition sub' : field := (1)%nat.

Inductive contents_ : Type :=
  | Empty : contents_
  | Nonempty : node_ → contents_.

Definition heap_ : Type := loc.

Definition merge_pairs : val :=
  val_fixs "merge_pairs" ("l" :: nil)
    (trm_let "q1" (trm_apps MList_ml.pop ("l" :: nil))
      (trm_let "x0__" (trm_apps MList_ml.is_empty ("l" :: nil))
        (trm_if "x0__" "q1"
          (trm_let "q2" (trm_apps MList_ml.pop ("l" :: nil))
            (trm_let "q" (trm_apps merge ("q1" :: "q2" :: nil))
              (trm_let "x1__" (trm_apps MList_ml.is_empty ("l" :: nil))
                (trm_if "x1__" "q"
                  (trm_let "x2__" (trm_apps "merge_pairs" ("l" :: nil))
                    (trm_apps merge ("q" :: "x2__" :: nil))))))))).
```

Imperative pairing heaps: characteristic formula

```
Definition merge_pairs_cf_def__ : Prop :=
  Wpgen_body
    (∀ (l : loc) (H : hprop) (A : Type) (EA : Enc A) (Q : A → hprop),
     himpl H
       (Wptag (Wpgen_let_trm
              (Wptag
                 (Wpgen_app node_ MList_ml.pop
                    (dyn_make l :: nil)))
              (fun q1 : node_ ⇒
               Wptag (Wpgen_let_trm
                    (Wptag
                       (Wpgen_app bool MList_ml.is_empty
                          (dyn_make l :: nil))) (fun x0__ : bool ⇒ ...

Lemma merge_pairs_cf__ : merge_pairs_cf_def__.
Proof using. (* Proof script remains to be automated *)
  cf_main. cf_app. cf_app. cf_if.
  { cf_val. }
  { cf_app. cf_app. cf_app. cf_if.
    { cf_val. }
    { cf_app. cf_app. } }
Qed.
```

Summary

Characteristic formulae: no more deeply embedded terms

- avoid reduction contexts during proofs
- avoid simplifying substitutions during proofs
- enable the lifting technique

Lifting technique: no more deeply embedded values

- quantifies program variables at the corresponding Coq type
- simplifies the statement of postconditions
- enables representation predicates such as `MList` $(1::2::3::\texttt{nil})$ `p`
- simplifies reasoning on pattern matching

Characteristic formulae with lifting

- hides the deep embedding from the user
- leads to more concise proof scripts
- can be justified in a foundational way

Characteristic formulae and lifting in Iris?

Pointers

→ A modern eye on separation logic for sequential programs

`http://www.chargueraud.org/research/2023/hdr/chargueraud_hdr.pdf`

→ Foundations of Separation Logic, Volume 6 of Software Foundations

→ Example case studies

- *Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits.*
  Charguéraud and Pottier, JAR'19.
- *Verifying a hash table and its iterators in higher-order separation logic*
  Pottier, CPP'17
- *Formal proof and analysis of an incremental cycle detection algorithm*
  Guéneau, Jourdan, Charguéraud, and Pottier, ITP'19
- *Specification and verification of a transient stack.*
  Moine, Charguéraud, Pottier, CPP'22

Thanks!