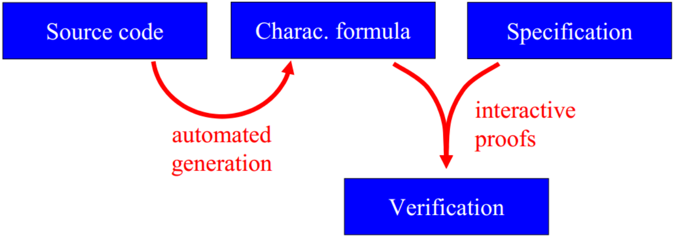
A new, axiom-free implementation of CFML for the verification of imperative programs

Arthur Charguéraud

Inria

2017/10/13

CFML: program verification using characteristic formulae



Old CFML: too large trusted code base, including OCaml parser, OCaml typechecker, and the characteristic formula generator (in OCaml).

Challenges in removing axioms

- CF generation is typed directed.
- Treatment of polymorphism involves quantification over type.
- Lifting of program values into logical values.
- CF generator involves non-structural recursion.
- Nontrivial soundness and completeness proofs.

Armaël's Work on CakeML in HOL

Impressive results:

- Implemented a characteristic formula generator entirely inside HOL.
- Proved its soundness once and for all.
- Generalized the generator to add support for exceptions.

Yet:

- HOL lacks an equivalent of Coq evars, limiting the ability to reproduce CFML tactics for verifying programs in practice.
- Connexion between values from the deep embedding and values from the logic is manual, whereas CFML automatically performs this *lifting* of values.

This work

Tackles the following challenges:

- Develop a characteristic formula generator in Coq.
- Try to simplify the proofs as much as possible.
- Integrate lifting of values directly in the generator.
- Adapt CFML tactics to the new setting.

- Target a language covering both ML-style and C-style programs.
- Add fine-grained ownership of individual fields or cells.

This talk

1. Semantics in Coq.
2. Separation Logic in Coq.
3. Characteristic formulae in Coq.
4. Support for lifting of values.
5. Treatment of records.

Semantics

Syntax of the deep embedding

```
Definition var := nat.
Definition loc := nat.
Definition null : loc := 0%nat.

Inductive prim : Type :=
  | val_get : prim
  | val_set : prim
  | val_ref : prim
  | val_alloc : prim
  | val_eq : prim
  | val_add : prim
  | val_sub : prim
  | val_mul : prim
  | val_div : prim
  | val_ptr_add : prim.
```

```
Inductive val : Type :=
  | val_unit : val
  | val_bool : bool → val
  | val_int : int → val
  | val_loc : loc → val
  | val_prim : prim → val
  | val_fun : var → trm → val
  | val_fix : var → var → trm → val

with trm : Type :=
  | trm_val : val → trm
  | trm_var : var → trm
  | trm_fun : var → trm → trm
  | trm_fix : var → var → trm → trm
  | trm_if : trm → trm → trm → trm
  | trm_seq : trm → trm → trm
  | trm_let : var → trm → trm → trm
  | trm_app : trm → trm → trm
  | trm_while : trm → trm → trm
  | trm_for : var → trm → trm → trm → trm.
```

Syntax with coercions and notations

```
int mlist_length(cell* p) {
  cell* q;
  int n;
  if (p != null) {
    q = p->tl;
    n = mlist_length(q);
    return n+1;
  } else {
    return 0;
  }
}
```

```
Definition val_mlist_length : val :=
  Vars F, P, Q, N in
  ValFix F P :=
    If_ val_neq P null Then (
      Let Q := val_get_tl P in
      Let N := F Q in
      val_add N 1
    ) Else (
      0
    ).
```

Also supports n-ary applications:

```
Context (v:val) (p:loc).
Check (val_push_front v p : trm).
```

Semantics of the deep embedding

```
Definition state := fmap loc val.

Inductive red : state → trm → state → val → Prop :=
  | red_val : ∀m v,
      red m v m v
  | red_fun : ∀m x t1,
      red m (trm_fun x t1) m (val_fun x t1)
  | red_fix : ∀m f x t1,
      red m (trm_fix f x t1) m (val_fix f x t1)
  | red_if : ∀m1 m2 m3 b r t0 t1 t2,
      red m1 t0 m2 (val_bool b) →
      red m2 (if b then t1 else t2) m3 r →
      red m1 (trm_if t0 t1 t2) m3 r
  | red_seq : ∀m1 m2 m3 t1 t2 r,
      red m1 t1 m2 val_unit →
      red m2 t2 m3 r →
      red m1 (trm_seq t1 t2) m3 r
  | red_let : ∀m1 m2 m3 x t1 t2 v1 r,
      red m1 t1 m2 v1 →
      red m2 (subst x v1 t2) m3 r →
      red m1 (trm_let x t1 t2) m3 r
  | red_app_arg : ∀m1 m2 m3 m4 t1 t2 v1 v2 r,
      red m1 t1 m2 v1 →
      red m2 t2 m3 v2 →
      red m3 (trm_app v1 v2) m4 r →
      red m1 (trm_app t1 t2) m4 r
  | red_app_fun : ∀m1 m2 v1 v2 x t r,
      v1 = val_fun x t →
      red m1 (subst x v2 t) m2 r →
      red m1 (trm_app v1 v2) m2 r
  | red_app_fix : ∀m1 m2 v1 v2 f x t r,
      v1 = val_fix f x t →
      red m1 (subst f v1 (subst x v2 t)) m2 r →
      red m1 (trm_app v1 v2) m2 r
  | red_while : ∀m1 m2 t1 t2 r,
      red m1 (trm_if t1 (trm_seq t2 (trm_while t1 t2))
             val_unit) m2 r →
      red m1 (trm_while t1 t2) m2 r
  | red_for_arg : ∀m1 m2 m3 m4 v1 v2 x t1 t2 t3 r,
      red m1 t1 m2 v1 →
      red m2 t2 m3 v2 →
      red m3 (trm_for x v1 v2 t3) m4 r →
      red m1 (trm_for x t1 t2 t3) m4 r
  | red_for : ∀m1 m2 x n1 n2 t3 r,
      red m1 (
        If (n1 ⩽ n2)
          then (trm_seq (subst x n1 t3) (trm_for x (n1
                 +1) n2 t3))
          else val_unit) m2 r →
      red m1 (trm_for x n1 n2 t3) m2 r
...
```

Separation Logic

Separation Logic assertions (heap predicates)

```
Definition hprop := state → Prop.

Definition hempty : hprop := (* written \[] *)
  fun h ⇒ h = fmap_empty.

Definition hpure (P:Prop) : hprop := (* written \[P] *)
  fun h ⇒ h = fmap_empty ∧ P.

Definition hsingle (l:loc) (v:val) : hprop := (* written (l↦v) *)
  fun h ⇒ h = fmap_single l v ∧ l ≠null.

Definition hstar (H1 H2:hprop) : hprop := (* written (H1 ⋆H2) *)
  fun h ⇒ ∃h1 h2, H1 h1
              ∧ H2 h2
              ∧ fmap_disjoint h1 h2
              ∧ h = fmap_union h1 h2.

Definition hexists(A:Type) (J:A→hprop) : hprop := (* written (∃ x, H) *)
  fun h ⇒ ∃x, J x h.

Definition htop : hprop := (* written ⊤*)
  fun (h:heap) ⇒ True.
```

Separation Logic triples

```
Definition Hoare_triple (H:hprop) (t:trm) (Q:val→hprop) : Prop :=
  ∀(h:state), H h → ∃(v:val) (h':state), red h t h' v ∧ Q v h'.
```

```
Definition SL_triple (H:hprop) (t:trm) (Q:val→hprop) :=
  ∀(H':hprop), Hoare_triple (H ⋆ H') t (Q ⋆ H').
```

```
Definition triple (t:trm) (H:hprop) (Q:val→hprop) :=
  ∀(H':hprop), Hoare_triple (H ⋆ H') t (Q ⋆ H' ⋆ ⊤).
```

Example: mutable linked lists

```
Lemma rule_mlist_length : ∀(L:list val) (p:loc),
  triple (val_mlist_length p)
    (p ⇝ MList L)
    (fun (r:val) ⇒ \[r = val_int (length L)] ⋆ p ⇝ MList L).

Definition field : Type := nat.
Definition hd : field := 0%nat.
Definition tl : field := 1%nat.

Definition MCell (v:val) (q:val) (p:loc) : hprop := (* written [p ⇝ MCell v q] *)
  ((p+hd) ↦ v) ⋆ ((p+tl) ↦ q).

Fixpoint MList (L:list val) (p:loc) : hprop := (* written [p ⇝ MList L] *)
  match L with
  | nil ⇒ \[p = null]
  | x::L' ⇒ ∃(p':loc), (p ⇝ MCell x p') ⋆ (p' ⇝ MList L')
  end.
```

Separation Logic rules

```
Lemma rule_frame : ∀t H Q H',
  triple t H Q →
  triple t (H ⋆ H') (Q ⋆ H').
```

```
Lemma rule_consequence : ∀t H' Q' H Q,
  H ⊳ H' →
  triple t H' Q' →
  Q' ⊳ Q →
  triple t H Q.
```

```
Lemma rule_htop_post : ∀t H Q,
  triple t H (Q ⋆ ⊤) →
  triple t H Q.
```

```
Lemma rule_extract_hexists : ∀t (A:Type) (J:A→hprop) Q,
  (∀ x, triple t (J x) Q) →
  triple t (hexists J) Q.
```

```
Lemma rule_let : ∀(x:var) t1 t2 H Q Q1,
  triple t1 H Q1 →
  (∀ (X:val), triple (subst x X t2) (Q1 X) Q) →
  triple (trm_let x t1 t2) H Q.
```

```
Lemma rule_if_bool : ∀(b:bool) t1 t2 H Q,
  (b = true → triple t1 H Q) →
  (b = false → triple t2 H Q) →
  triple (trm_if b t1 t2) H Q.
```

```
Lemma rule_apps_fixs : ∀xs f F (Vs:vals) t1 H Q,
  F = (val_fixs f xs t1) →
  var_fixs f (length Vs) xs →
  triple (subst f F (substs xs Vs t1)) H Q →
  triple (trm_apps F Vs) H Q.
```

```
Lemma rule_set : ∀w l v,
  triple (val_set (val_loc l) w)
    (l ↦ v)
    (fun r ⇒ \[r = val_unit] ⋆ l ↦ w).
```

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall X.\ \{Q'\ X\}\ ([x \to X]\,t_2)\ \{Q\}}{\{H\}\ (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)\ \{Q\}}\ \text{LET}$$

Interactive proofs

```
Lemma rule_mlist_length : ∀L p,
  triple (val_mlist_length p)
    (p ⇝ MList L)
    (fun r ⇒ \[r = val_int (length L)] ⋆ p ⇝ MList L).
Proof using.
  intros L. induction_wf: list_sub_wf L. intros p.
  applys rule_app_fix. reflexivity. simpl.
  applys rule_if'. { xapplys rule_neq. }
  simpl. intros X. xpull. intros EX. subst X.
  case_if as C1; case_if as C2; tryfalse.
  { inverts C2. xchange MList_null_inv.
    xpull. intros EL. applys rule_val. hsimpl. subst~. }
  { xchange (MList_not_null_inv_cons p). { auto. }
    xpull. intros x p' L' EL. applys rule_let.
    { xapplys rule_get_tl. }
    { simpl. intros p''. xpull. intros E. subst p''.
      applys rule_let.
      { simpl. xapplys IH. { subst~. } }
      { simpl. intros r. xpull. intros Er. subst r.
        xchange (MList_cons p p' x L').
        xapplys rule_add_int.
        { intros. subst L. hsimpl. }
        { intros. subst. rew_length. fequals. math. } } } }
Qed.
```

Interactive proofs using characteristic formulae

```
Lemma rule_mlist_length' : ∀L p,
  triple (val_mlist_length p)
    (p ⇝ MList L)
    (fun r ⇒ \[r = val_int (length L)] ⋆ p ⇝ MList L).
Proof using.
  intros L. induction_wf: list_sub_wf L. xcf.
  xapps. xif ;⇒ C.
  { xchanges~ (MList_not_null_inv_cons p) ;⇒ x p' L' EL. subst L.
    xapps. xapps~ IH.
    xchange (MList_cons p).
    xapps. hsimpl. isubst. rew_length. fequals. math. }
  { inverts C. xchanges MList_null_inv ;⇒ EL. subst. xvals~. }
Qed.
```

Characteristic formulae

Properties of characteristic formulae

The characteristic formula $[\![t]\!]$ of a term t is a predicate such that:

$$\forall H Q. \quad [\![t]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\; t\; \{Q\}$$

```
Theorem triple_of_cf : ∀t H Q,
  cf t H Q → triple t H Q.
```

Characteristic formula for let-bindings

We want: $[\![t]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\; t\; \{Q\}$.

In particular: $[\![\text{let}\, x = t_1 \,\text{in}\, t_2]\!]\, H\, Q \;\Leftrightarrow\; \{H\}\; (\text{let}\, x = t_1 \,\text{in}\, t_2)\; \{Q\}$.

Separation Logic rule:

$$\frac{\{H\}\; t_1\; \{Q'\} \qquad \forall X.\; \{Q'\, X\}\; ([x \to X]\, t_2)\; \{Q\}}{\{H\}\; (\text{let}\, x = t_1 \,\text{in}\, t_2)\; \{Q\}}$$

Characteristic formula:

$$\begin{aligned}[\![\text{let}\, x = t_1 \,\text{in}\, t_2]\!] \;\equiv\; \lambda HQ.\; \exists Q'.\quad & [\![t_1]\!]\, H\, Q' \\ & \wedge\; \forall X.\; [\![([x \to X]\, t_2)]\!]\, (Q'\, X)\, Q\end{aligned}$$

Characteristic formulae generator in Coq

$$[\![\text{let}\, x = t_1 \,\text{in}\, t_2]\!] \;\equiv\; \lambda H Q.\ \exists Q'.\quad [\![t_1]\!]\, H\, Q' \\ \wedge\ \forall X.\ [\![([x \to X]\, t_2)]\!]\, (Q'\, X)\, Q$$

```
Definition formula := hprop → (val → hprop) → Prop.

Fixpoint cf (t:trm) : formula := (* simplified *)
  match t with
  | trm_let x t1 t2 ⇒ fun H Q ⇒ ∃Q', cf t1 H Q' ∧ ∀(X:val), cf (subst x X t2) (Q' X) Q
  ...
  end.
```

```
Definition cf_let (F1:formula) (F2of:val→ formula) : formula := fun H Q ⇒
  ∃Q', F1 H Q' ∧ ∀X, (F2of X) (Q' X) Q.

Fixpoint cf (t:trm) : formula := (* simplified *)
  match t with
  | trm_let x t1 t2 ⇒ cf_let (cf t1) (fun X ⇒ cf (subst x X t2))
  ...
  end.
```

Integration of structural rules using local

$$\frac{H \rhd H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \rhd Q \star \top}{\{H\}\ t\ \{Q\}}\ \text{FRAME-CONSEQUENCE-GC}$$

$$\frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{Q\}}\ \text{EXISTS} \qquad \frac{P\ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \text{PROP}$$

All structural rules can be simulated using this predicate transformer:

```
Definition local (F:(hprop→(val→hprop)→Prop)) : formula := fun H Q ⇒
   ∀h, H h → ∃H1 H2 Q1,
      (H1 ⋆ H2) h
    ∧ F H1 Q1
    ∧ Q1 ⋆ H2 ⊳ Q ⋆ ⊤ .
```

Characteristic formulae generator in Coq

```
Fixpoint cf (t:trm) : formula := (* simplified *)
  match t with
  | trm_val v ⇒ local (cf_val v)
  | trm_var x ⇒ local (cf_fail) (* unbound variable *)
  | trm_fun x t1 ⇒ local (cf_val (val_fun x t1))
  | trm_fix f x t1 ⇒ local (cf_val (val_fix f x t1))
  | trm_if t0 t1 t2 ⇒ local (cf_if (cf t0) (cf t1) (cf t2))
  | trm_seq t1 t2 ⇒ local (cf_seq (cf t1) (cf t2))
  | trm_let x t1 t2 ⇒ local (cf_let (cf t1) (fun X ⇒ cf (subst x X t2)))
  | trm_app t1 t2 ⇒ local (triple t)
  | trm_while t1 t2 ⇒ local (cf_while (cf t1) (cf t2))
  | trm_for x t1 t2 t3 ⇒ local (
      match t1, t2 with
      | trm_val v1, trm_val v2 ⇒ cf_for v1 v2 (fun X ⇒ cf (subst x X t3))
      | _, _ ⇒ cf_fail (* not in A-normal form *)
      end)
  end.
```

```
Definition cf_fail : formula := fun H Q ⇒ False.
```

The optimal fixed point combinator at work

```
Definition CF cf (t:trm) := (* define the functional *)
  match t with
  | trm_let x t1 t2 ⇒ local (cf_let (cf t1) (fun X ⇒ cf (subst x X t2)))
  ...
  end.

Definition cf := FixFun CF. (* apply the optimal fixed point combinator *)

Lemma cf_unfold : ∀t, cf t = CF cf t. (* fixed point equation *)

Fixpoint func_iter n A B (F:(A→B)→(A→B)) (f:A→B) (x:A) : B :=
  match n with
  | O ⇒ f x
  | S n' ⇒ F (func_iter n' F f) x
  end.

Lemma cf_unfold_iter : ∀n t, (* iterated fixed point equation *)
  cf t = func_iter n CF cf t.
(* 4 lines of proof *)
```

Soundness of the generator

```
Theorem triple_of_cf : ∀t H Q,
  cf t H Q → triple t H Q.
Proof using.
  intros t. induction_wf: trm_size t. rewrite cf_unfold. destruct t; simpl;
   try (applys sound_for_local; intros H Q P).
  { unfolds in P. applys~ rule_val. hchanges~ P. }
  { false. }
  { unfolds in P. applys rule_fun. hchanges~ P. }
  { unfolds in P. applys rule_fix. hchanges~ P. }
  { destruct P as (Q1&P1&P2). applys rule_if.
    { applys* IH. }
    { intros v. specializes P2 v. applys sound_for_local (rm P2).
      clears H Q Q1. intros H Q (b&P1'&P2'&P3'). inverts P1'.
      case_if; applys* IH. }
    { intros v N. specializes P2 v. applys local_extract_false P2.
      intros H' Q' (b&E&S1&S2). subst. applys N. hnfs*. } }
  { destruct P as (H1&P1&P2). applys rule_seq' H1.
    { applys~ IH. }
    { intros X. applys~ IH. } }
  { destruct P as (Q1&P1&P2). applys rule_let Q1.
    { applys~ IH. }
    { intros X. applys~ IH. } }
  { applys P. }
  ... (* plus proof cases for while-loops and for-loops *)
Qed.
```

Proofs using characteristic formulae

```
Notation "'`Val' v" := (local (cf_val v)).
Notation "'`Let' x ':=' F1 'in' F2" := (local (cf_let F1 (fun x => F2))).
Notation "'`If' v 'Then' F1 'Else' F2" := (local (cf_if_val v F1 F2)).
Notation "'`LetIf' F0 'Then' F1 'Else' F2" := (local (cf_if F0 F1 F2)).
...

  L : list val
  p : loc
  ______________________________________________________(1/1)
  (`LetIf (`App ((val_neq p) null))
        Then `Let X := `App (val_get_tl p) in
             `Let X0 := `App (val_mlist_length X) in
              `App ((val_add X0) 1)
        Else (`Val 0))
  (p ~> MList L)
  (fun r : val => \[r = length L] \* p ~> MList L)
```

Lifting of values

Lifted specifications

Post-condition of length:

$$\texttt{fun}\ (\texttt{r}:\texttt{val}) \Rightarrow \backslash[\texttt{r} = \texttt{val\_int}\ (\texttt{length L})] \star (\texttt{p} \rightsquigarrow \texttt{MList L})$$

Cake-ML presentation:

$$\texttt{fun}\ (\texttt{r}:\texttt{val}) \Rightarrow \backslash[\texttt{INT r}\ (\texttt{length L})] \star (\texttt{p} \rightsquigarrow \texttt{MList L})$$

Specifications using lifted values:

$$\texttt{fun}\ (\texttt{r}:\texttt{int}) \Rightarrow \backslash[\texttt{r} = \texttt{length L}] \star (\texttt{p} \rightsquigarrow \texttt{MList L})$$

Lifted specifications, cont.

Post-condition of incr:

$$\texttt{fun}\ (\texttt{r}:\texttt{val}) \Rightarrow \backslash[\texttt{r} = \texttt{val\_unit}] \star (\texttt{p} \leadsto \texttt{val\_int}\ (\texttt{n}+1))$$

Same, with lifting:

$$\texttt{fun}\ (\texttt{\_}:\texttt{unit}) \Rightarrow (\texttt{p} \leadsto (\texttt{n}+1))$$

Typeclasses for lifted values

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

Notation "‘ V" := (enc V).

Global Instance Enc_int : Enc int.
Proof using. constructor. applys val_int. Defined.

Check (‘5 : val).

Fixpoint MList A ‘{EA:Enc A} (L:list A) (p:loc) : hprop :=
  match L with
  | nil ⇒ \[p = null]
  | x::L' ⇒ ∃(p':loc), (p ⇝ Record‘{ hd := x; tl := p' })
                     ⋆ (p' ⇝ MList L')
  end.
```

Lifted Separation Logic

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

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

Lemma Rule_let : ∀x t1 t2 H,
  ∀A `{EA:Enc A} (Q:A→hprop) A1 `{EA1:Enc A1} (Q1:A1→hprop),
  Triple t1 H Q1 →
  (∀ (X:A1), Triple (subst x `X t2) (Q1 X) Q) →
  Triple (trm_let x t1 t2) H Q.
```

Lifted CF generator

```
Definition Formula := ∀‘{Enc A}, hprop → (A → hprop) → Prop.

Notation "’ F H Q" := ((F:Formula) _ _ H Q).

Definition Cf_let (F1 : Formula) (F2of : ∀‘{EA1:Enc A1}, A1 → Formula) : Formula :=
  fun ‘{Enc A} H (Q:A→ hprop) ⇒
    ∃(A1:Type) (EA1:Enc A1) (Q1:A1→ hprop),
        ’F1 H Q1
     ∧ (∀ (X:A1), ’(F2of X) (Q1 X) Q).

Definition Cf_def Cf (t:trm) : Formula :=
  match t with
  | trm_let y t1 t2 ⇒ Local (Cf_let (Cf t1)
                              (fun ‘{EA:Enc A} (X:A) ⇒ Cf (subst y ‘X t2)))
  ...
  end.

Theorem Triple_of_Cf : ∀(t:trm) A ‘{EA:Enc A} H (Q:A→ hprop),
  ’(Cf t) H Q → Triple t H Q.
```

Proofs using lifted CF

```
Lemma Rule_mlist_length : ∀A `{EA:Enc A} (L:list A) (p:loc),
  Triple (val_mlist_length `p)
    PRE (p ⇝ MList L)
    POST (fun (r:int) ⇒ \[r = length L] ⋆ p ⇝ MList L).
Proof using.
  intros. gen p. induction_wf: list_sub_wf L; intros. xcf.
  xapps~. xif ;⇒ C.
  { xchanges~ (MList_not_null_inv_cons p) ;⇒ x p' L' EL.
    xapps. xapps~ IH. xchange (MList_cons p).
    xapps. hsimpl. isubst. auto. }
  { subst. xchanges MList_null_inv ;⇒ EL. xvals~. }
Qed.
```

Other examples

- Other functions on lists such as reverse.
- List traversals using a while loop.
- Higher-order iteration on lists.
- Mutable queue with in-place merge, using linked cells.
- Union-Find (simpler version without union by rank).

`https://gitlab.inria.fr/charguer/cfml/tree/master/model`

Representation of records

Generic representation predicate for records

```
Context A `{EA:Enc A} (v:A) (p:loc) (q:loc).
Check (p ⇝ Record`{ hd := v; tl := q } : hprop).

Notation "`{ f1 := x1 ; f2 := x2 }" :=
  ((f1, Dyn x1)::(f2, Dyn x2)::nil)

Definition Record_fields : Type := list (field * dyn).

Record dyn := Dyn { dyn_type : Type;
                    dyn_enc : Enc dyn_type;
                    dyn_value : dyn_type }.

Fixpoint Record (L:Record_fields) (p:loc) : hprop :=
  match L with
  | nil ⇒ \[]
  | (f, Dyn V)::L' ⇒ ((p+f) ↦ (enc V)) ⋆ (p ⇝ Record L')
  end.
```

Computing access specifications in Coq

```
Lemma Rule_set_field : ∀`{EA:Enc A} (V1 V2:A) (p:loc) (f:field),
  Triple (val_set_field f p `V2)
    PRE (p `.` f ↦ V1)
    POST (fun _ ⇒ p `.` f ↦ V2).
```

Generate on the fly:

```
  Triple (val_set_field hd p `V2)
    PRE (p ⇝ Record`{ hd := `V1; tl := q })
    POST (fun _ ⇒ p ⇝ Record`{ hd := `V2; tl := q }).
```

Computing access specifications in Coq

```
(* Generate the specification of field update on a record representation *)
Definition set_spec (f:field) `{EA:Enc A} (W:A) (L:Record_fields) : option Prop :=
  match list_assoc_update_or_none f (Dyn W) L with
  | None ⇒ None
  | Some L' ⇒ Some (∀ p,
      Triple (val_set_field f `p `W) (p ⇝ Record L) (fun (_:unit) ⇒ p ⇝ Record L'))
  end.

(* Proof of correctness of the generated specification *)
Lemma set_spec_correct : ∀(f:field) `{EA:Enc A} (W:A) (L:Record_fields) (P:Prop),
  (set_spec f W L = Some P) → P.
```

Thanks!

`https://gitlab.inria.fr/charguer/cfml/tree/master/model`